Konrad-Zuse-Zentrum für Informationstechnik Berlin
Heilbronner Str. 10, D-10711 Berlin-Wilmersdorf

Detlev Stalling Hans-Christian Hege

Fast and Resolution Independent Line Integral Convolution

Preprint SC-94-37 (June 1995)

Fast and Resolution Independent Line Integral Convolution

Detlev Stalling   Hans-Christian Hege

Abstract

Line Integral Convolution (LIC) is a powerful technique for generating striking images and animations from vector data. Introduced in 1993, the method has rapidly found many application areas, ranging from computer arts to scientific visualization. Based upon locally filtering an input texture along a curved stream line segment in a vector field, it is able to depict directional information at high spatial resolutions.

We present a new method for computing LIC images. It employs simple box filter kernels only and minimizes the total number of stream lines to be computed. Thereby it reduces computational costs by an order of magnitude compared to the original algorithm. Our method utilizes fast, error-controlled numerical integrators. Decoupling the characteristic lengths in vector field grid, input texture and output image, it allows computation of filtered images at arbitrary resolution. This feature is of significance in computer animation as well as in scientific visualization, where it can be used to explore vector data by smoothly enlarging structure of details.

We also present methods for improved texture animation, again employing box filter kernels only. To obtain an optimal motion effect, spatial decay of correlation between intensities of distant pixels in the output image has to be controlled. This is achieved by blending different phase-shifted box filter animations and by adaptively rescaling the contrast of the output frames.

CR Categories: I.3.3 [Computer Graphics]: Picture/Image generation; I.3.6 [Computer Graphics]: Methodology and Techniques; I.4.3 [Image Processing]: Enhancement

Additional Keywords: vector field visualization, texture synthesis, periodic motion filtering

Contents

1 Introduction

Generation of textured images from various kinds of vector fields has become an important issue in scientific visualization as well as in animation and special effects. In 1993 Cabral and Leedom presented a powerful technique for imaging vector data called *line integral convolution* [1]. Their algorithm has been used as a general tool for visualizing vector fields. Additionally it has broad applications for image enhancement. A major drawback of the original algorithm, however, is its high computational expense and its restriction to a fixed spatial resolution.

In this paper we present an improved algorithm for line integral convolution, in which computation of streamlines is algorithmically separated from that of convolution. This allows us to exploit economies and to provide wider functionalism in each of the computational steps. The new algorithm

- is *about an order of magnitude faster* than original line integral convolution, making interactive data exploration possible
- is *more accurate* by employing an adaptive, error-controlled streamline integration technique
- is *resolution independent*, enabling the user to investigate image details by smooth detail enlargement (zooming)
- *improves texture animation* using shifted box filter kernels together with a simple blending technique.

In recent years a number of methods for artificially generating textures have been suggested. These methods cover a variety of applications. In the field of scientific visualization texture-based methods are of special interest because they allow the display of vector fields in an unrivaled spatial resolution. Traditionally, vector data has been represented by small arrows or other symbols indicating vector magnitude and direction. This approach is restricted to a rather coarse spatial resolution. More sophisticated methods include the display of stream lines [8], stream surfaces [10], flow volumes [14], as well as various particle tracing techniques [19, 9, 11]. These methods are well suited for revealing characteristic features of vector fields. However, they strongly depend on the proper choice of seed points. Experience shows that interesting details of the field may easily be missed.

Texture-based methods are not affected by such problems. They depict *all* parts of the vector field and thus are not susceptible to missing characteristic data features. In addition they achieve a much higher spatial resolution, which in some sense can be viewed as the *maximum possible* resolution since the minimum possible feature size of a textured image is a single pixel. In an early method introduced by van Wijk [18] a random texture is convolved along a straight line segment oriented parallel to the local vector direction. Line integral convolution (LIC) [1] modifies this method, so that convolution takes place along curved stream line segments. In this way field structure can be represented much more clearly. Forssell [5] describes another extension that allows her to map flat LIC images onto curvilinear surfaces in three dimensions.

Vector fields are not only of relevance in science and engineering. Many objects of our natural environment exhibit characteristic directional features which are naturally represented by vector data. Consequently algorithms for turning such data into pictorial information are of great importance for synthetic image generation, image post-processing, and computer arts [6, 16]. The variety of directional filters offered by commercial image processing software is just one evidence for this.

The remainder of the paper is organized as follows. Section 2 provides mathematical background and fixes notation. The basic ideas of the new algorithm are outlined in section 3. In the following three sections we present algorithms for fast and accurate streamline integration, discuss some optimization issues, and sketch strategies for fast texture map sampling. We then discuss periodic motion filtering and smooth detail enlargement. Finally we present some results and give an outlook concerning various aspects of LIC methods.

2 Background

Before looking at line integral convolution, let us introduce vector fields formally, define some characteristic features and fix notation. For more detailed expositions on vector fields see standard texts on vector analysis, e.g. [13]. Restricting ourselves to the simplest case, we consider a stationary *vector field* defined by a map $\boldsymbol{v}:\mathbb{R}^2 \to \mathbb{R}^2, \boldsymbol{x} \mapsto \boldsymbol{v}(\boldsymbol{x})$.

The directional structure of $\boldsymbol{v}$ can be graphically depicted by its *integral curves*, also denoted *flow lines* or *stream lines*[1]. An integral curve is a path $\boldsymbol{\sigma}(u)$ whose tangent vectors coincide with the vector field:

$$\frac{d}{du}\,\boldsymbol{\sigma}(u) = \boldsymbol{v}(\boldsymbol{\sigma}(u)) \tag{1}$$

Like any path, $\boldsymbol{\sigma}(u)$ can be reparametrized by a continous, strictly increasing function without changing its shape and orientation. For our purpose it is convenient to use arc-length s. Noting that $ds/du = |\boldsymbol{v}(\boldsymbol{\sigma}(u))|$ we have

$$\frac{d}{ds}\,\boldsymbol{\sigma}(s) = \frac{d\boldsymbol{\sigma}}{du}\frac{du}{ds} = \frac{\boldsymbol{v}}{|\boldsymbol{v}|} \equiv \boldsymbol{f}(\boldsymbol{\sigma}(s)). \tag{2}$$

[1] The image of integral curves (“lines of force”) and their graphical representation played a crucial role in Faradays development of the field concept during 1820-1850 [15].

Of course, this reparametrization is only valid in regions of non-vanishing $|\boldsymbol{v}|$, i.e. for non-degenerate curves $\boldsymbol{\sigma}$. To find a stream line through $\boldsymbol{x}$ the ordinary differential equation (2) has to be solved with the initial condition $\boldsymbol{\sigma}(0) = \boldsymbol{x}$. It can be proved that there is a *unique* solution if the right hand side $\boldsymbol{f}$ locally obeys a Lipschitz-condition. In particular this condition is fulfilled for any function with continuous first derivative. Otherwise, there may exist *multiple* solutions at a single point $\boldsymbol{x}$, i.e. multiple stream lines may start at that point. A typical example are point sources in an electrostatic field. Numerical integrators used in LIC have to be robust enough to handle such cases. Beside isolated singularities also discontinuities occur quite often in vector fields. Usually these are encountered across the boundaries of distinctly characterized field regions, e.g. regions with different electromagnetic properties. An example of this is shown in Fig. 1.

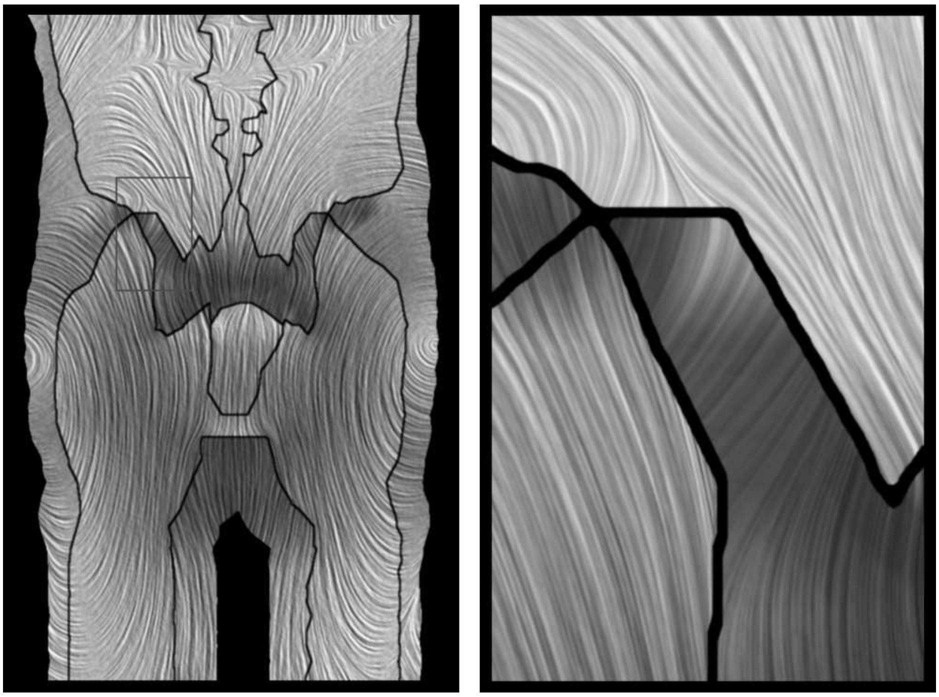


Figure 1: LIC image of a vector field (electrical field) containing discontinuities. Field strength $|\boldsymbol{v}|$ is indicated by color.

Given a stream line $\boldsymbol{\sigma}$, line integral convolution consists in calculating the intensity for a pixel located at $\boldsymbol{x}_0 = \boldsymbol{\sigma}(s_0)$ by

$$I(\boldsymbol{x}_0) = \int_{s_0-L}^{s_0+L} k(s - s_0)\, T(\boldsymbol{\sigma}(s))\, ds. \tag{3}$$

Here T denotes an input texture, usually some sort of random image like white noise. The filter kernel k is assumed to be normalized to unity. The convolution operation (3) causes pixel intensities to be *highly correlated* along individual stream lines, but independent in directions perpendicular to them. In the resulting images the directional structure of the vector field is clearly visible. Usually good results are obtained by choosing filter length $2L$ to be 1/10th of the image width. It is possible to simultaneously visualize field strength $|\boldsymbol{v}|$ by coloring or animating LIC images.

3 Making Line Integral Convolution Fast

In traditional LIC for each pixel in the output image a separate stream line segment and a separate convolution integral are computed. There are two types of redundancies in this approach. First, a single stream line usually covers lots of image pixels. Therefore in traditional LIC large parts of a stream line are recomputed very frequently. Second, for a *constant* filter kernel k very similar convolution integrals occur for pixels covered by the same stream line. This is not exploited by traditional LIC. Consider two points located on the same stream line, $\boldsymbol{x}_1 = \boldsymbol{\sigma}(s_1)$ and $\boldsymbol{x}_2 = \boldsymbol{\sigma}(s_2)$. Assume, both points are separated by a small distance $\Delta s = s_2 - s_1$. Then for a constant filter kernel k obviously

$$I(\boldsymbol{x}_2) = I(\boldsymbol{x}_1) \ - \ k\int_{s_1-L}^{s_1-L+\Delta s} T(\boldsymbol{\sigma}(s))\, ds \ + \ k\int_{s_1+L}^{s_1+L+\Delta s} T(\boldsymbol{\sigma}(s))\, ds. \tag{4}$$

The intensities differ by only two small correction terms that are rapidly computed by a numerical integrator. By calculating long stream line segments that cover many pixels and by restricting to a constant filter kernel we avoid both types of redundancies being present in traditional LIC.

To design a fast LIC algorithm, we have taken an approach which relies on computing the convolution integral by sampling the input texture T at evenly spaced locations $\boldsymbol{x}_i$ along a pre-computed stream line $\boldsymbol{\sigma}(s)$. For the moment we assume that input texture and output image are of the same size, like in traditional LIC. The distance between different sample points is denoted by h_t. We initiate

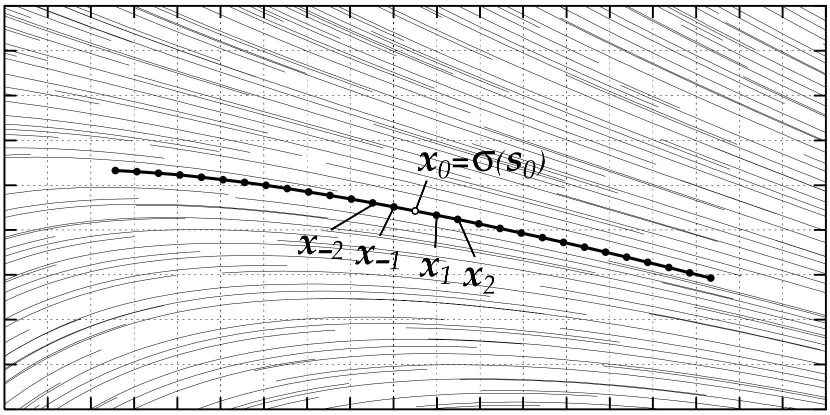


Figure 2: The input texture is sampled at evenly spaced locations $\boldsymbol{x}_i$ along a stream line $\boldsymbol{\sigma}$. For each location the convolution integral $I(\boldsymbol{x}_i)$ is added to the pixel containing $\boldsymbol{x}_i$. A new stream line is computed only for those pixels where the number of samples does not already exceed a user-defined limit.

stream line computation for some location $\boldsymbol{x}_0 = \boldsymbol{\sigma}(s_0)$ (see Fig. 2). The convolution integral for this location is approximated as

$$I(\boldsymbol{x}_0) = k \sum_{i=-n}^{n} T(\boldsymbol{x}_i), \text{ with } \boldsymbol{x}_i = \boldsymbol{\sigma}(s_0 + ih_t). \tag{5}$$

To ensure normalization we take $k = 1/(2n+1)$. The resulting intensity is added to the output image pixel containing $\boldsymbol{x}_0$. Calculation of more accurate trapezoidal sums instead of Riemann sums is nearly as fast, but does not pay in terms of the visual effect. After having computed $I(\boldsymbol{x}_0)$, we step in both directions along the current stream line, thereby updating the convolution integrals as follows

$$\begin{aligned} I(\boldsymbol{x}_{m+1}) &= I(\boldsymbol{x}_m) + k\,[T(\boldsymbol{x}_{m+1+n}) - T(\boldsymbol{x}_{m-n})] \\ I(\boldsymbol{x}_{m-1}) &= I(\boldsymbol{x}_m) + k\,[T(\boldsymbol{x}_{m-1-n}) - T(\boldsymbol{x}_{m+n})]. \end{aligned} \tag{6}$$

For each sample point the corresponding output image pixel is determined and the current intensity is added to that pixel. In this way we efficiently obtain intensities for many pixels covered by the same stream line. The probability for an output image pixel to be hit by a sample point is proportional to the length of the line segment covering that pixel. This can be used to set up some sort of quality control. Running through all output image pixels, we require the total number of hits already occurred in each pixel to be larger than some minimum. If the number of hits is smaller than the minimum, a new stream line computation is initiated. Otherwise that pixel is skipped. At the end accumulated intensities for all pixels have to be normalized against the number of individual hits. Basically our algorithm (referenced as 'fast-LIC' hereafter) can be described by the following pseudocode:

```
for each pixel p
    if (numHits(p) < minNumHits) then
        initiate stream line computation with x_0 = center of p
        compute convolution I(x_0)
        add result to pixel p
        set m = 1
        while m < some limit M
            update convolution to obtain I(x_m) and I(x_{-m})
            add results to pixels containing x_m and x_{-m}
            set m = m + 1
for each pixel p
    normalize intensity according to numHits(p)
```

There are a number of remarks necessary at this point. First, if stream line segments were computed for each pixel separately, the discrete sampling approach would be tainted with major aliasing problems, unless h_t is chosen much smaller than the width of a texture cell. However, if a single stream line is used for many pixels, correlation of pixel intensities along the stream line is guaranteed because exactly the same sampling points are used for convolution. We found a step size of $h_t = 0.5$ times the width of a texture cell to be completely sufficient. Although we have assumed input texture and output image to be of the same size, the fast-LIC algorithm can easily be generalized to set these sizes independently. This is necessary for smooth detail enlargement as discussed in Sect. 8.

The order in which all the output image pixels are processed is of some importance for the efficiency of the algorithm. The goal is to hit as many uncovered pixels with each new stream line as possible. Some optimization strategies are discussed in Sect. 5.

In our algorithm the computation of stream line segments can be performed without referencing input texture or output image. This allows us to utilize powerful, adaptive numerical integration methods. We have implemented several different integrators, which are discussed in Sect. 4. These methods not only accelerate stream line tracking significantly in homogeneous regions, but also ensure high accuracy necessary for resolving small details. Accuracy is especially important in fast-LIC because multiple stream lines determine the intensity of a single pixel. If these lines are incorrectly computed, the LIC pattern gets disturbed. This is most evident near the center of a vortex in the vector field.

Accurate stream line integration also offers new opportunities for texture animation using shifted filter kernels, cf. Sect. 7. For animation we need a full sized convolution range. Therefore, when a stream line leaves the domain of $\boldsymbol{v}$, we continue the path in the current direction. For texture sampling all points are remapped to fall somewhere into the input texture. We continue stream lines in a similar way if $|\boldsymbol{v}|$ vanishes or if a singularity was encountered. Of course, artificially continued stream line segments can not be used to determine intensities of the underlying pixels.

4 Streamline Integration

Usually the vector field will not be available in functional form. For sake of simplicity we assume $\boldsymbol{v}$ to be given at discrete locations on an uniform grid. Vector values at intermediate locations have to be computed by interpolation. We use bilinear interpolation. Of course, better interpolation schemes can be employed if more information is available about the field. Sometimes global field properties are known, e.g. the existance of closed stream lines. In general these properties are not retained, when a local interpolation scheme like bilinear interpolation is used. In particular closed stream lines in the true vector field may no longer be closed in the interpolated field [12]. However, in practice errors due to interpolation are usually much smaller than errors caused by a poor numerical integrator, unless $\boldsymbol{v}$ is given on a very coarse grid.

Bilinear interpolation results in a representation of the field that is not differentiable across the boundaries of grid cells. Therefore, to integrate Eq. (2) in general we can't rely on sophisticated algorithms like extrapolation methods or predictor-corrector schemes, which require a very smooth right hand side. Instead we have employed traditional Runge-Kutta methods. Accompanied with modern error monitoring and adaptive step size control these methods are quite competitive [17, 7, 3]. We also have to take into account that in many applications vector fields arise that are very rough or even discontinuous. In such cases stream line integration is confronted with the potential risk of missing small details embedded in homogeneous regions. This problem can be tackled by delimiting the maximum allowed step size of an adaptive numerical integrator. At the extreme, a really safe method would require stepping from cell to cell in the $\boldsymbol{v}$-grid.

A fast and accurate general-purpose stream line integrator can be built up from the well-known classical fourth-order Runge-Kutta formula. This formula requires four evaluations of the right hand side to proceed from some point $\boldsymbol{x}$ to some other point $\hat{\phi}^h\boldsymbol{x}$ located a step size h ahead on the same stream line:

$$\begin{aligned} \boldsymbol{k}_1 &= h\boldsymbol{f}(\boldsymbol{x}) & \boldsymbol{k}_3 &= h\boldsymbol{f}(\boldsymbol{x}+\tfrac{1}{2}\boldsymbol{k}_2) \\ \boldsymbol{k}_2 &= h\boldsymbol{f}(\boldsymbol{x}+\tfrac{1}{2}\boldsymbol{k}_1) & \boldsymbol{k}_4 &= h\boldsymbol{f}(\boldsymbol{x}+\boldsymbol{k}_3) \end{aligned}$$

$$\hat{\phi}^h\boldsymbol{x} = \boldsymbol{x} + \frac{\boldsymbol{k}_1}{6} + \frac{\boldsymbol{k}_2}{3} + \frac{\boldsymbol{k}_3}{3} + \frac{\boldsymbol{k}_4}{6} + O(h^5) \tag{7}$$

The equation is called fourth-order because it resembles the true solution up to a power of h^4. However, an integration method is rather useless without any means for estimating the actual value of the error term. It turns out that an independent third-order approximation $\bar{\phi}^h\boldsymbol{x}$ can be computed by reusing some of the intermediate steps in (7), namely

$$\bar{\phi}^h\boldsymbol{x} = \boldsymbol{x} + \frac{\boldsymbol{k}_1}{6} + \frac{\boldsymbol{k}_2}{3} + \frac{\boldsymbol{k}_3}{3} + \frac{h\boldsymbol{f}(\hat{\phi}^h\boldsymbol{x})}{6} + O(h^4). \tag{8}$$

The difference between both methods simply equals to

$$\Delta = \hat{\phi}^h \boldsymbol{x} - \bar{\phi}^h \boldsymbol{x} = \frac{1}{6}(\boldsymbol{k}_4 - h\boldsymbol{f}(\hat{\phi}^h \boldsymbol{x})). \tag{9}$$

This term is an estimate of the error of the less accurate formula. However, it can be shown [3] that in many cases this estimate can be safely used to control the step size of the more accurate method, too.

The idea of adaptive step size control is to choose h as large as possible while observing a user-defined error tolerance TOL. For p-th order integration methods the error term scales as h^{p+1}. Therefore if a step size h results in some error Δ, an optimized step size h^* can be obtained by

$$h^* = h \sqrt[p+1]{\rho \cdot \mathrm{TOL}/\Delta}, \tag{10}$$

with a safety factor $\rho < 1$. With this equation a control mechanism can be set up as follows. We ask the integrator to step forward by h and compute Δ from Eq. (9). If Δ is bigger than TOL, we repeat the current step with $h = h^*$. Otherwise, we proceed and take $h = \min(h^*, h_{\text{max}})$ for the next iteration, where h_{max} is the maximum allowed step size. If h becomes much smaller than the grid spacing, we assume that a singularity was encountered and terminate stream line integration. The resulting adaptive numerical integrator, denoted as RK4(3) hereafter, turns out to be very robust and well suited for our application.

We have also implemented two fifth order methods with fourth order error estimation. The first method due to Dormand and Prince [4] requires five $\boldsymbol{f}$-evaluations per iteration. The other due to Cash and Karp [2] requires six. In our case, where the right hand side $\boldsymbol{f}$ is obtained by bilinearly interpolating between discrete grid points, the higher order methods usually will not be significantly superior to RK4(3), except for smooth vector fields sampled at high resolution. However, experience shows that they will never be significantly inferior either.

5 Selecting Streamlines

For the fast-LIC algorithm it is not only important to quickly compute single stream lines, but also to process the output image pixels in such an order that the total number of stream line computations is minimized. For instance it is not a good idea to process pixels in scanline order, because it would be quite probable that new stream lines hit pixels already being covered by other lines. Instead of looking for the optimal pixel to be processed next, we simply subdivide the image into smaller blocks, taking the first pixel of each block, then the second, and so on. With this method the number of computed stream lines is typically about 2% of the number of image pixels. It is possible to incorporate some more sophisticated schemes here like Sobol quasi-random sequences [17], which may be combined with methods for finding areas in the image not covered by stream lines so far.

To obtain an approximately equal stream line density in the image, we stop following an individual line after some distance Mh_t (cf. pseudocode in Sect. 3). Ideally, this length should be adjusted automatically. If lots of previously covered pixels are encountered, computation should be terminated. However, currently we are using a much simpler scheme which nevertheless works reasonably well. We use a fixed M until a certain percentage of pixels is hit. For the remaining pixels we simply compute a short stream line segment and the corresponding convolution integral, but do not traverse the stream line further. Usually a covering limit of 90% and a value Mh_t of about 50-100 pixel widths yield optimal run times, but these values are not that critical for overall performance.

A simple way to compensate for a non-optimal stream line selection strategy is to decrease the minimum number of hits required for a pixel. Even with a low limit the total number of hits for each pixel may be large due to stream lines which are computed later. In fact, for all images in this paper we have taken a limit of only a single hit. Despite this low value, each pixel usually will be covered by several stream lines, as may be seen from Fig. 2.

6 Texture Map Convolution

The ODE solvers discussed in Sect. 4 are able to quickly compute long stream lines at guaranteed high accuracy. However, the actual step sizes used by these integrators are usually much bigger than the distance h_t needed for texture sampling. Therefore we have to interpolate between every two neighbouring locations returned by the ODE solver. The distance between these locations and the curvature of the stream line may easily take values that prohibit the use of a simple linear interpolation scheme. This is illustrated in Fig. 3. Average increments from 10 to 30 times the spacing of the $\boldsymbol{v}$-grid are quite common in practice.

A much better approximation of stream lines can be obtained using cubic Hermite-interpolation, for convenience with a rescaled parameter $u \in [0, 1]$,

$$\boldsymbol{p}(u) = \boldsymbol{a}u^3 + \boldsymbol{b}u^2 + \boldsymbol{c}u + \boldsymbol{d}, \qquad u = \frac{s - s_n}{s_{n+1} - s_n} \tag{11}$$

with coefficients

$$\begin{aligned}
\boldsymbol{a} &= 2\boldsymbol{p}(0) - 2\boldsymbol{p}(1) + \boldsymbol{p}'(0) + \boldsymbol{p}'(1) \\
\boldsymbol{b} &= -3\boldsymbol{p}(0) + 3\boldsymbol{p}(1) - 2\boldsymbol{p}'(0) - \boldsymbol{p}'(1) \\
\boldsymbol{c} &= \boldsymbol{p}'(0) \\
\boldsymbol{d} &= \boldsymbol{p}(0).
\end{aligned}$$

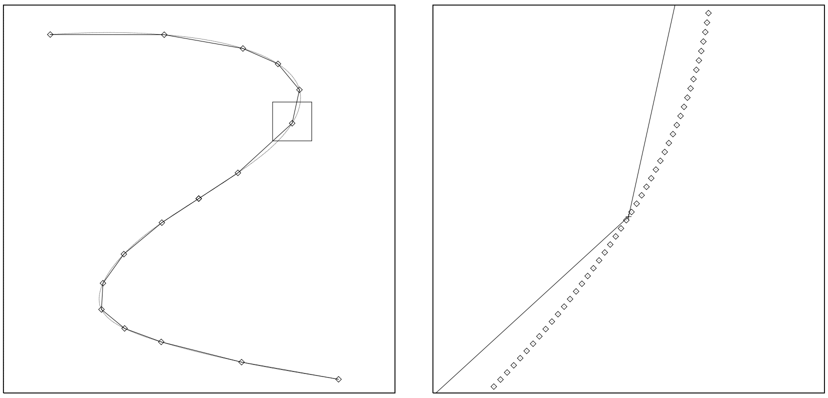


Figure 3: Distances between stream line points as returned by the adaptive numerical integrators are usually so large that cubic interpolation is necessary to track the path for texture sampling.

$\boldsymbol{p}$ and $\boldsymbol{p}'$ expressed in terms of stream line position and orientation are

$$\begin{aligned} \boldsymbol{p}(0) &= \boldsymbol{x}_n & \boldsymbol{p}'(0) &= (s_{n+1} - s_n)\, \boldsymbol{f}(x_n) \\ \boldsymbol{p}(1) &= \boldsymbol{x}_{n+1} & \boldsymbol{p}'(1) &= (s_{n+1} - s_n)\, \boldsymbol{f}(x_{n+1}). \end{aligned}$$

This ensures that the first derivative at the boundaries of the interpolation interval is represented correctly. Since we need to evaluate the cubic interpolation polynomial at evenly spaced sample points only, a forward difference scheme can be employed for stream line tracking. Forward differences are defined by

$$\begin{aligned} \Delta^1 \boldsymbol{p}(u) &= \boldsymbol{p}(u + \delta) - \boldsymbol{p}(u) \\ &= 3\boldsymbol{a}\delta u^2 + (3\boldsymbol{a}\delta^2 + 2\boldsymbol{b}\delta)u + \boldsymbol{a}\delta^3 + \boldsymbol{b}\delta^2 + \boldsymbol{c}\delta \\ \Delta^2 \boldsymbol{p}(u) &= \Delta^1 \boldsymbol{p}(u + \delta) - \Delta^1 \boldsymbol{p}(u) \\ &= 6\boldsymbol{a}\delta^2 u + 6\boldsymbol{a}\delta^3 + 2\boldsymbol{b}\delta^2 \\ \Delta^3 \boldsymbol{p}(u) &= \Delta^2 \boldsymbol{p}(u + h) - \Delta^2 \boldsymbol{p}(u) \\ &= 6\boldsymbol{a}\delta^3 = \text{const.} \end{aligned}$$

To step along the curve with constant increment $\delta = h_t/(s_{n+1} - s_n)$ we first have to compute $\Delta^1 \boldsymbol{p}(u_0)$, $\Delta^2 \boldsymbol{p}(u_0)$, and $\Delta^3 \boldsymbol{p}(u_0)$; then intermediate positions are obtained by using the recursive relationships

$$\begin{aligned} \boldsymbol{p}(u_{k+1}) &= \boldsymbol{p}(u_k) + \Delta^1 \boldsymbol{p}(u_k) \\ \Delta^1 \boldsymbol{p}(u_{k+1}) &= \Delta^1 \boldsymbol{p}(u_k) + \Delta^2 \boldsymbol{p}(u_k) \\ \Delta^2 \boldsymbol{p}(u_{k+1}) &= \Delta^2 \boldsymbol{p}(u_k) + \Delta^3 \boldsymbol{p}(u_k). \end{aligned}$$

After initialization, forward differences require just three additions per component to evaluate the polynomial, instead of three additions and three multiplications required by Horner's rule.

Note, that we cannot assume u_0 to be zero, because in general the distance between two neighbouring positions returned by the integration algorithm will not be a multiple of h_t. Instead, the remainder of h_t which just doesn't fit into the previous interval anymore will serve as the initial offset for the next interval.

It should be noted that we do not necessarily need to keep interpolation separate from stream line integration. As an interesting alternative so-called continous integration methods might be considered [7]. These provide dense output, i.e. solution values at intermediate points $\tilde{x} = x_i + \theta h$ with $0 < \theta \leq 1$. The trick is to gather appropriate information during integration to constitute an interpolation polynomial that can be evaluated without much additional cost. For the 5-th order method of Dormand and Prince a 4-th order continous extension is possible without an extra function evaluation; the solution becomes the fifth-order solution for $\theta = 1$ [7].

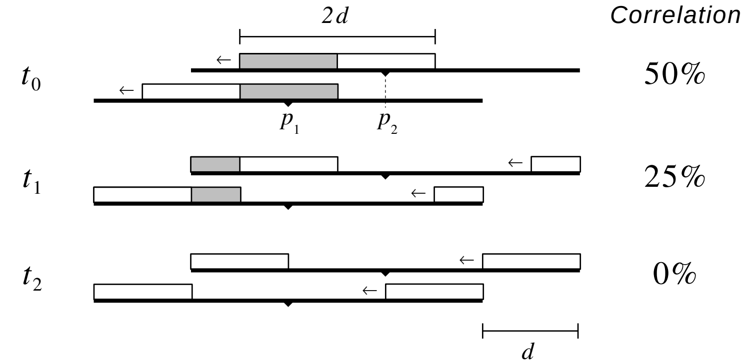
7 Periodic Motion Filters

LIC images can be animated by changing the shape and location of the filter kernel k over time. The apparent motion is well suited to envision vector field *direction* in addition to the pure tangential information contained in static images. In previous work [1] specially designed periodic filter kernels have been used to achieve a motion effect. On first sight it might appear difficult to combine texture animation with the fast-LIC algorithm, since the latter is restricted to constant filter kernels, i.e. box filters. However, this is not the case. In the following we will first introduce the notion of intensity correlation. We will then present a simple blending technique that keeps intensity correlation constant over time and thereby achieves high quality animations.

7.1 Intensity Correlation

Using box filters, an obvious method to animate LIC images is to cycle the boxes through some interval along the stream lines. If this is done with equal velocity for all pixels, a periodic sequence arises. Cycling a box filter can be easily accomplished with the fast-LIC algorithm. Essentially we just have to add some periodic offset function to the limits of the convolution sum in Eq. (5).

It turns out that this naive approach is not well suited for animation since noticeable artifacts are introduced when the boxes reenter the interval. To see this, consider two points p_1 and p_2 on a single stream line that are half a filter length apart. The corresponding pixel intensities initially have a 50% correlation because half of the texture cells being convolved are covered by both filter boxes. When the filter boxes reenter the interval, correlation suddenly drops to zero, as depicted in the following figure:



An intensity correlation function ξ measuring the amount of overlap between filter kernels k for two points separated by a distance d may be defined as

$$\xi(d,t) = \frac{\int \min(k(s,t), k(s+d,t))\, ds}{\int k(s,t)\, ds} \tag{12}$$

for each frame t. For a cycled box filter a plot of this function is shown in Fig. 4a. The length of the filter box was chosen to be 0.5 times the length of the interval. Reduced correlation results in a smaller feature size in the resulting LIC images. This is perceived as a disturbing artifact in animation. Note, that at the same time distant points temporarily become correlated.

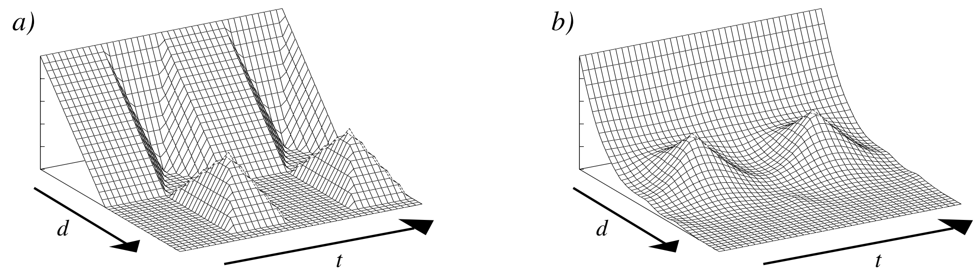


Figure 4: Intensity correlation between two points on a single stream line for different motion filter kernels: box filter (a) and Hanning filter (b). Two periods are shown in t-direction.

To achieve a smoother motion Cabral and Leedom [1] suggested to employ a weighted filter kernel made up of two so-called Hanning filters.

$$k(s,t) = \frac{1+\cos(\kappa s)}{2} \times \frac{1+\cos(n\kappa s + \omega t)}{2}$$

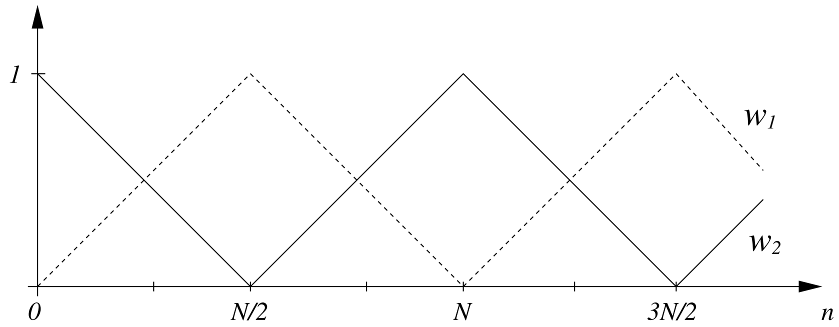
with $\kappa = 2\pi/2L$. For $n = 2$ the corresponding correlation is depicted in Fig. 4b. This function varies significantly less over time than the correlation for the cycled box filter. However, it cannot be used in conjunction with the fast-LIC algorithm. Fortunately, there is a simple method capable of generating periodic animation sequences that can be used in fast-LIC. With this method *no* artifacts at all occur due to reentering filter boxes.

7.2 Frame Blending

Consider an image sequence B_n, $n = 0, 1, ..., N$–1, with a filter box running along some stream line segment, but *not* reentering at the beginning. Obviously, such a sequence is not periodic anymore, but it will exhibit a constant intensity correlation over time. We have simply discarded all frames associated with the peaks in Fig. 4a. A periodic sequence A of length $N/2$ may be obtained by smoothly blending between phase-shifted B-frames, namely

$$A_n = w_1(n)\, B_{n \bmod N} + w_2(n)\, B_{(n+\frac{1}{2}N) \bmod N} \tag{13}$$

with the weights w_1 and w_2 chosen as follows:



This means that frames get less and less weighted as their filter boxes get closer to the extreme positions. Whenever w_i equals one, the middle frame of B will be visible. For each pixel both intensity contributions are completely independent, provided that filter boxes do not overlap. In this case averaging multiple LIC images is statistically equivalent to computing the convolution integral from a modified input texture given as the weighted average of two textures distributed in the original way. While effective filter length L remains the same, averaging multiple frames causes the contrast of the resulting image to be reduced. This has to be compensated.

In raw LIC images intensity I of a single pixel usually is given by convolving a large number of independent texture cells. Therefore the central limit theorem of statistics applies and I can assumed to be gaussian distributed, that is

$$\psi(I) = \text{const.} \exp(-\frac{(I-\mu)^2}{2\sigma^2}). \tag{14}$$

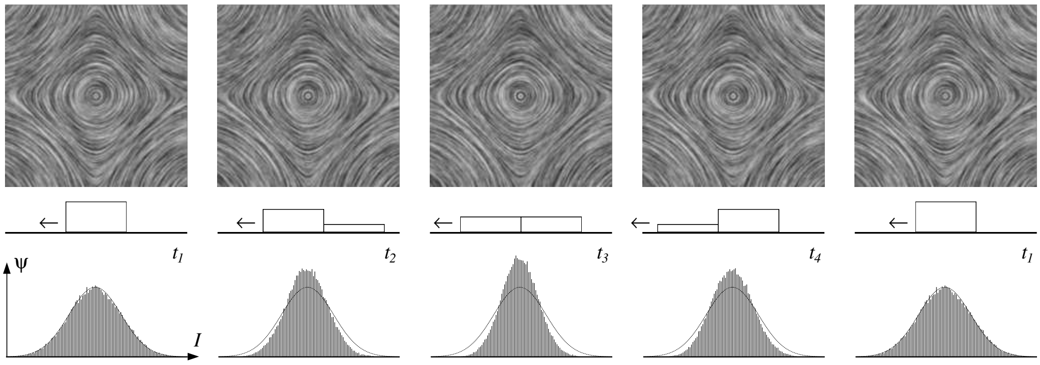


Figure 5: Snapshots from a periodic LIC animation obtained by frame blending. The first and the last image are identical. The figure contains a schematic view of the differently weighted filter boxes moving along the stream line. In the lower part intensity histograms of the blended images are shown. To keep contrast constant, intensity has to be rescaled to fit the original gaussian distribution.

Here μ and σ^2 denote average and variance of the intensity distribution ψ, respectively. Any linear combination of independent gaussian distributed quantities will again be distributed gaussian. The resulting variance is given by $\sigma^2_{\text{res}} = \sum w_i^2 \sigma_i^2$. Consequently, after averaging multiple LIC images of equal μ and σ^2, the original intensity distribution and therefore also contrast can be restored by a simple linear scaling,

$$I \leftarrow \frac{I - \mu}{\sqrt{w_1^2 + w_2^2}} + \mu. \tag{15}$$

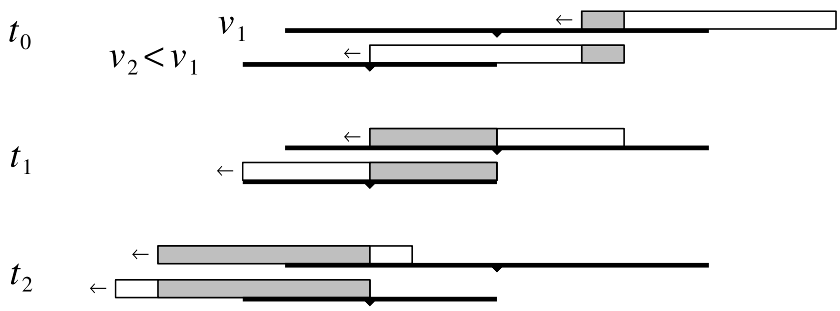
Figure 5 summarizes the process of frame blending and intensity rescaling. Note, that for Eq. (15) to be valid intensities need to be statistically independent. This is guaranteed if the filter boxes in the frames being averaged do not overlap, i.e. if filter length does not exceed 0.5 times the length of the interval. As an alternative we may also use two image sequences computed from completely different input textures. In this case a periodic sequence of length N would be obtained.

7.3 Variable Velocities

The simple blending technique described above comes to its real value when the texture is to be animated with *variable* velocities for each pixel. Such animations are useful to display not only vector direction and orientation, but also to give an impression of vector magnitude $|\boldsymbol{v}|$.

For variable velocities the standard filter cycling approach will not yield periodic sequences anymore. Forsell [5] describes a technique for endlessly playing back a variable motion movie from a fixed number of pre-computed constant speed images. The final intensity for a pixel is computed by interpolating the pixel intensities from those two images, where the filter kernel phase approximately resembles the actual value. However, there still remains a major problem. With ongoing time, filter kernel phases for neighbouring pixels will lose any correlation. Drastic spatio-temporal aliasing effects are introduced. For example the texture may appear to move in the opposite direction in some areas.

To avoid these effects we build up a variable speed animation from only such frames, where the filter kernel phases are correlated. Correlated frames can be produced by letting filter boxes move some variable distance proportional to their velocity, as depicted in the following picture:



To generate a periodic sequence we would like to use the blending technique described above again. However, in general the intensities being averaged are not independent because filter boxes overlap in regions of low velocity. Therefore Eq. (15) is not valid anymore. It is also not a good idea to use two image sequences computed from different input textures. This would cause the LIC pattern to change over time in regions of low velocity. Although no flow would be perceived, blending between different patterns is somehow irritating. Instead, we have to rescale intensity *locally* according to the actual amount of filter box overlap. Overlap is inversely proportional to velocity and may be described by a number $u \in [0, 1]$. An expression for the resulting local variance can be derived by splitting blended intensity into three independent contributions, one due to the overlapping part and two due to the non-overlapping parts of the individual boxes:

$$\sigma^2_{\text{res}} = \Big((w_1^2 + w_2^2)(1-u)^2 + u^2\Big)\,\sigma^2. \tag{16}$$

With this equation we are able to rescale intensity of every pixel so that the original σ^2 is restored. In this way a high quality animation sequence is obtained.

It should be noted that building up animation sequences from shifted box filter convolutions requires accurate stream line computation, because highly unsymmetric convolution ranges can occur. These will emphasize errors due to poor numerical integration. For example, circular stream lines may be falsely depicted as spirals. Artifacts of this kind are usually disguised by a symmetric filter kernel [18, 1]. They do not occur if stream line integration is accurate.

8 Smooth Detail Enlargement

For many applications it is useful to adjust the size of a LIC input texture, so that a single texture cell is covered by lots of output image pixels. This can be easily accomplished with the fast-LIC algorithm. As before we are using Eq. (5) to compute the convolution integral for some initial point $\boldsymbol{x}_0$. It is sufficient to sample the input texture at increments $h_t = 0.5$ times the width of a texture cell. However, when stepping along the stream line and updating the integral according to Eq. (6), we use a smaller step size in order to ensure that we hit as many pixels covered by the stream line as before. Of course, using a smaller step size means that the value of k in Eq. (6) has to be adjusted, too. The ability to choose the sizes of input texture and output image independently can be exploited in several ways.

First, in LIC images created from high frequency input textures, such as white noise, these high frequencies are retained in directions perpendicular to the field direction. This is caused by the one-dimensional nature of the filter kernel. The resulting images often look quite busy. Problems arise if the images are to be processed by lower bandwidth filters like video tape recorders or image compression algorithms. The usual remedy is to use a low-pass filtered input texture or to blur the final LIC images afterwards. With our algorithm convolutions over long distances L can easily be computed. Therefore a better approach is to simply scale up the size of a texture cell as well as convolution length L in terms of pixel width.

With traditional LIC it is hard to generate exactly the same image at different resolutions. It would require to use both a resampled input texture as well as a resampled vector field. This approach is tedious and will unnecessarily introduce errors. However, often it is important to create several versions of a single image at different resolutions, e.g. adopted to various output devices, or for use in animations that require distance dependent texture resolution. This can be easily accomplished with fast-LIC since the size of the output image can be chosen independently of vector field resolution and the input texture.

A slightly different utilization of this feature is the computation of smooth zooms into the vector data field to enlarge interesting details. As an example some close-ups of details in a vector field are shown in Fig. 6, where linear magnification extends up to a factor 100.

If the zoom is to be played back in a sequence, care has to be taken for low magnification factors. If stream line integration is unconditionally started at the center of output image pixels, then in each frame slightly different stream lines are computed. This causes annoying variations in texture to occur from frame to

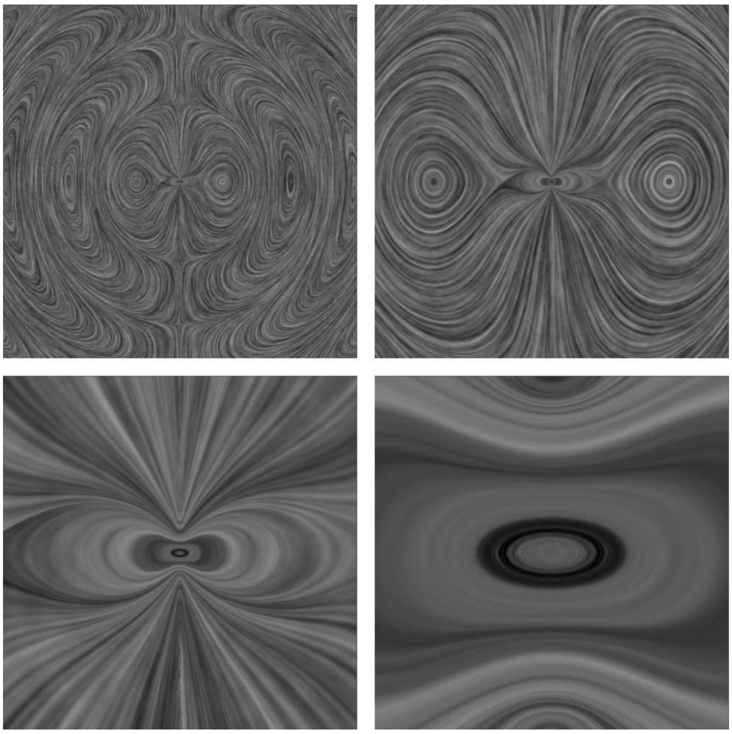


Figure 6: Details of a vector field displayed at different magnification factors (1, 3, 15, 100). For each frame a completely new LIC image has been computed. The data set had a resolution of 500^2. At the finest level only a few grid points are covered.

frame. One solution would be to increase the minimal number of hits required for a pixel. Another more robust method is to try exactly the same stream lines used in the previous frame first. For these lines the starting point will not correspond to the center of an output image pixel anymore. Remaining pixels are treated as usual afterwards. This method yields smooth animation sequences, allowing one to compute striking trips into details.

9 Results

We have implemented the fast-LIC algorithm in the C++ programming language within the framework of the modular visualization environment *IRIS Explorer*^{TM}. Within this system it is possible to pre-process the vector field as well as to post-process the resulting LIC images in various ways. We have found it especially useful to apply a directional gradient filter to the raw LIC images to further emphasize directional information. Another useful method is to multiply color into the images to simultaneously visualize a scalar quantity in addition to vector field

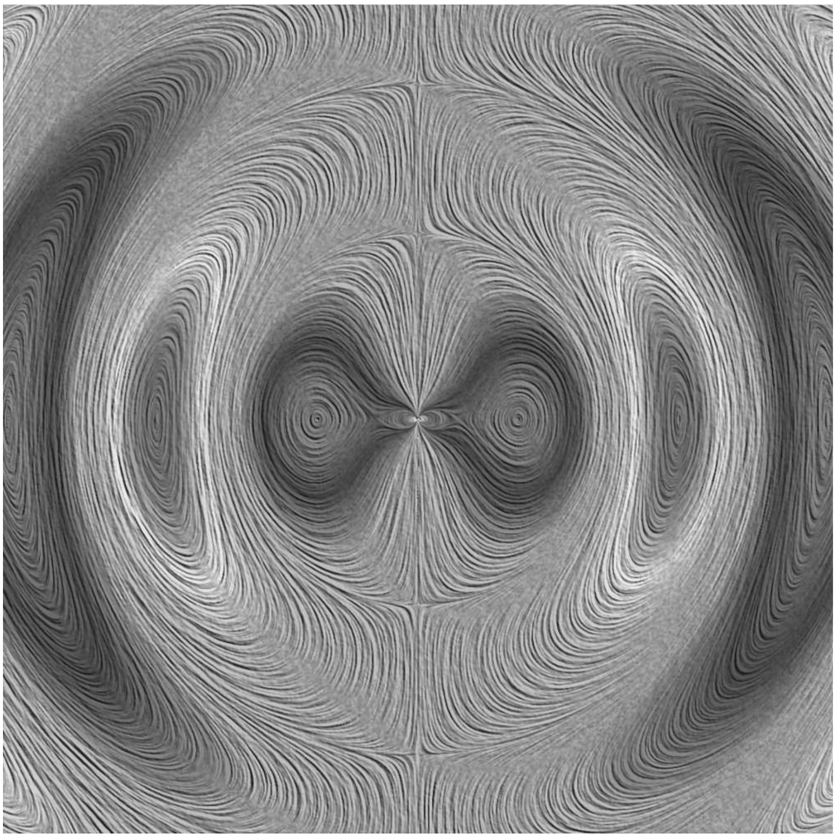


Figure 7: Field of an irradiating dipole antenna. The same data as in Fig. 6 is shown. Field strength is indicated by color. Note, how gradient filtering and coloring emphasize the vector field structure.

orientation. The images in Fig. 1, 7, and 8 were post-processed in this way.

The data shown in Fig. 1 comes from so-called hyperthermia simulation, a form of cancer therapy based upon radiating radio waves into the human hip region. In Fig. 7 electrical field lines irraditated by a dipole antenna are depicted. This image has to be compared with Fig. 6a. In both cases the same vector field is shown. However, after gradient filtering and coloring, the image looks much more attractive. In Fig. 8 a snapshot from the simulation of an instationary fluid flow around a cylinder is shown. Finally, Fig. 9 presents an application of LIC in modern art.

Table 1 summarizes some execution times of fast-LIC compared to the original LIC algorithm of Cabral and Leedom. The numbers, obtained on a SGI Indigo^2 with 150 MHz MIPS R4400, are in seconds. They refer to the vector fields shown in Fig. 1, 7 and 8, but do not take into account computing time for gradient filtering and coloring. For better comparision with the original algorithm the dimensions of input texture, vector field, and resulting image were chosen to be equal. The actual

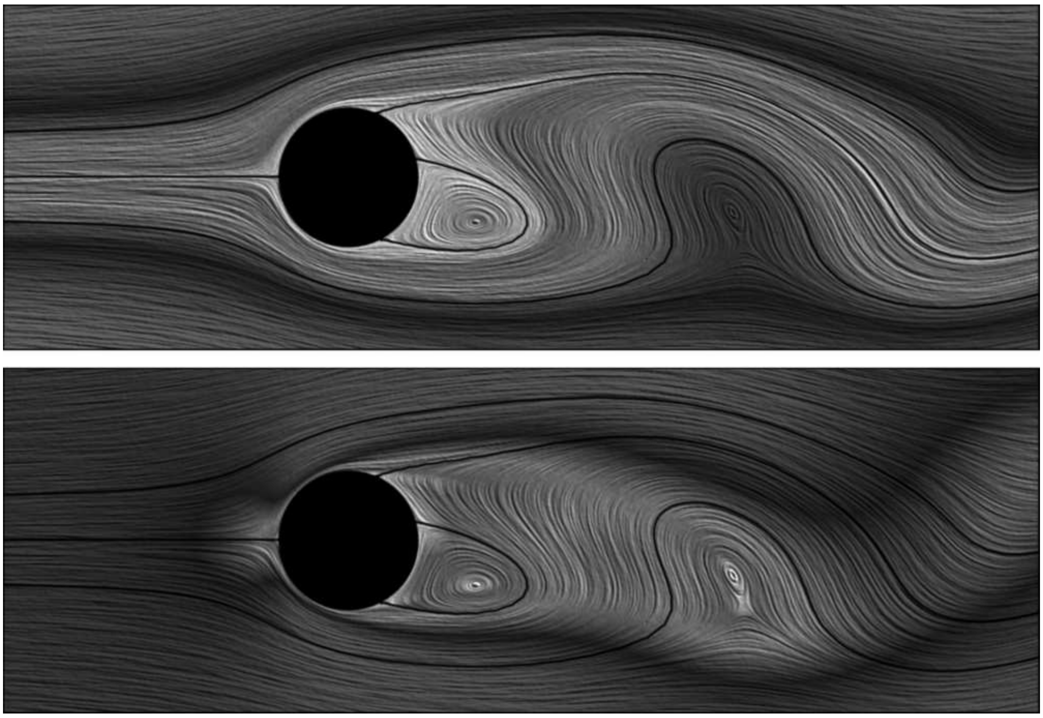


Figure 8: Flow around a cylinder. Color depicts the value of stream function (upper) and magnitude of velocity (lower). In the lower image directional information still is clearly visible, although color does not correspond to stream line shape.

Figure 9: LIC-based variations on a scissors cut of Henri Matisse.

	L	LIC	RK	CK	DP
Hyperthermia 400×600	10	12.26	**3.36**	3.65	3.55
	20	21.93	**3.75**	4.15	3.99
	40	41.36	**4.60**	5.20	4.88
Dipole 500×500	10	18.35	4.35	4.41	**4.31**
	20	34.29	4.78	4.81	**4.60**
	40	71.14	5.61	5.61	**5.39**
Cylinder 600×200	10	7.76	**1.49**	1.54	1.57
	20	14.44	**1.62**	1.65	1.70
	40	27.01	**1.92**	1.99	2.00

Table 1: Performance of the original LIC algorithm compared to the new algorithm using different numerical integrators: RK = adaptive Runge-Kutta scheme RK4(3), CK = Cash and Karp, DP = Dormand and Prince (cf. Sect. 4). The boldface entry gives the shortest time in each row.

sizes are indicated in the table. L is the extent of the convolution integral in one direction. The table contains different columns for various numerical integrators we have implemented. These integrators do not differ much in performance. Usually only about 25% of the time is spent in stream line integration. Most time is spent in texture sampling. For the hyperthermia data set, fast-LIC performs somewhat worse than in the other examples. This is caused by the discontinuities in the vector field, forcing the adaptive integrators to choose very small step sizes across the boundaries. The higher order methods are more affected by this than RK4(3).

10 Conclusion

We have introduced a new line integral convolution algorithm that performs an order of magnitude faster than previous methods. A feature of our method is the ability to compute images at arbitrary resolution. We presented methods for producing high quality texture animation sequences, employing constant filter kernels only.

The new techniques have particular significance for computer graphics. They are useful for fast procedural generation of textures with directional features and of texture sequences with continously variable spatial resolutions. The production of such sequences is of growing interest in computer animation, where several versions of a texture with different spatial resolutions are often needed for different views or output media.

There are a number of directions for future research. We intend to investigate the visualization of time varying and three-dimensional vector fields. The inclusion of visual representations of global and local vector field characteristics other than flow lines is also an interesting topic that deserves further investigation.

Finally there is room for considerable further research work with respect to computer animation, e.g. concerning the production of hierarchies of directional textures with different spatial resolutions, or new methods for synthesizing vector fields from images to auto-convolve them. This may lead to a new class of directional filters for image processing.

Acknowledgements

We would like to thank Charlie Gunn, Roland Wunderling, and Gerhard Zumbusch for reviewing the manuscript and for various helpful discussions. We are also grateful to the anonymous reviewers of this paper for their valueable remarks, and to Brian Cabral and Casey Leedom for making their code available on the net.

References

[1] Brian Cabral and Leith (Casey) Leedom. Imaging vector fields using line integral convolution. In James T. Kajiya, editor, *Computer Graphics (SIGGRAPH '93 Proceedings)*, volume 27, pages 263–272, August 1993.

[2] J. R. Cash and Alan H. Karp. A variable order Runge-Kutta method for initial value problems with rapidly varying right-hand sides. *ACM transactions on Mathematical Software, Vol. 16*, pages 201–222, 1990.

[3] Peter Deuflhard and Folkmar Bornemann. *Numerische Mathematik II: Integration gewöhnlicher Differentialgleichungen*. Verlag de Gruyter, Berlin, 1994.

[4] J. R. Dormand and P. J. Prince. Higher order embedded Runge-Kutta formulae. *J. Comp. Appl. Math.*, 7:67–75, 1981.

[5] Lisa K. Forssell. Visualizing flow over curvilinear grid surfaces using line integral convolution. In *Visualization '94*, pages 240–247. IEEE Computer Society, 1994.

[6] Paul E. Haeberli. Paint by numbers: Abstract image representations. In Forest Baskett, editor, *Computer Graphics (SIGGRAPH '90 Proceedings)*, volume 24, pages 207–214, August 1990.

[7] Ernst Hairer, Syvert Paul Nørsett, and Gerhard Wanner. *Solving Ordinary Differential Equations I, Nonstiff Problems*. Springer Verlag, Berlin, Heidelberg, New York, Tokyo, 1987.

[8] James L. Helman and Lambertus Hesselink. Visualizing vector field topology in fluid flows. *IEEE Computer Graphics and Applications*, 11(3):36–46, May 1991.

[9] Andrea J. S. Hin and Frits H. Post. Visualization of turbulent flow with particles. In *Visualization '93*, pages 46–52. IEEE Computer Society, October 1993.

[10] Jeff P. M. Hultquist. Interactive numerical flow visualization using stream S urfaces. *Computing Systems in Engineering*, 1(2-4):349–353, 1990.

[11] Kwan-Liu Ma and Philip J. Smith. Virtual smoke: An interactive 3d flow visualization technique. In *Visualization '92*, pages 46–52. IEEE Computer Society, October 1992.

[12] Gordon D. Mallinson. The calculation of the lines of a three-dimensional vector field. In Graham de Vahl Davis and Clive Fletcher, editors, *Computational Fluid Dynamics*, pages 525–534. North-Holland, August 1988.

[13] Jerrold E. Marsden and Anthony J. Tromba. *Vector Calculus*. W. H. Freeman, New York, 3rd edition, 1988.

[14] Nelson Max, Barry Becker, and Roger Crawfis. Flow volumes for interactive vector field visualization. In *Visualization '93*, pages 19–24, October 1993.

[15] Nancy John Nersessian. Faraday's field concept. In David Gooding and Frank A. J. L. James, editors, *Faraday Rediscovered: Essays on the Live and Work of Michael Faraday*, pages 175–187. Stockton Press, New York, 1985.

[16] Ken Perlin. An image synthesizer. In B. A. Barsky, editor, *Computer Graphics (SIGGRAPH '85 Proceedings)*, volume 19, pages 287–296, July 1985.

[17] William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. *Numerical Recipes in C: The Art of Scientific Computing*. Cambridge University Press, Cambridge, 2nd edition, 1992.

[18] Jarke J. van Wijk. Spot noise-texture synthesis for data visualization. In Thomas W. Sederberg, editor, *Computer Graphics (SIGGRAPH '91 Proceedings)*, volume 25, pages 309–318, July 1991.

[19] Jarke J. van Wijk. Rendering surface-particles. In *Visualization '92*, pages 54–61. IEEE Computer Society, October 1992.