LIC: Acceleration, Animation, and Zoom

Hans-Christian Hege, Detlev Stalling

Konrad-Zuse-Zentrum für Informationstechnik Berlin (ZIB)

1 Introduction

Line integral convolution is a nice technique, being of interest in visualization as well as in computer graphics and art. An essential prerequisite for its practical use is interactive speed. Furthermore, enlarged flexibility and functions for zooming or texture animation are required. In the following we will discuss algorithmic ideas that allow us to meet these requirements. The lecture is intended for both practioners, who want a deeper understanding of LIC algorithms, and programmers, aiming on implementing or modifying LIC code.

According the landmark paper of Brian Cabral and Casey Leedom [4], LIC basically means convolution of an input texture along lines that are defined by a vector field. For simplicity we call these lines "convolution lines". The main idea we are building upon in this lecture is to *separate the computation of convolution lines from that of convolution,* [14]. Thereby we are able to exploit economies and to provide wider functionalism in each of the computational steps. An implementation along these ideas is provided[1] in the Creative Applications Laboratory of Siggraph '97. This allows you to get hands-on experience and to try out various ideas.

Input to a LIC algorithm are a texture, usually some noisy image, and a vector field. In scientific visualization the vector field is just the object one aims to visualize. In computer graphics and art, usually artificial vector fields are used. Typically these are defined by applying some algorithmic rule to a given image. This may be the image to be convolved later, or another one. In Brian Cabral's and Casey Leedom's lecture an example of an useful algorithmic rule has already been mentioned: apply a bandlimiting filter to the input image, take the gradient and rotate the vectors by 90°. See also their original paper and the cover of the Siggraph '93 proceedings for such LIC images. Other kinds of artificial vector fields are used in Fig. 1.

The algorithmic decoupling mentioned above suggests an alternative procedure: Instead of calulating the convolution lines on basis of a vector field, they could be specified interactively, i.e. drawn freely, according to the artist's intention. For LIC a dense set of lines would be necessary, such that each pixel of the output image is crossed by one line at least. This could be achieved by interpolating somehow between the few lines drawn. However, for reasons that will become clear later, it is more appropriate also in this application to calculate a vector field from the user's input and compute the convolution lines on this basis. Therefore, in the following we assume that the

[1]The code is also available at `http://www.zib.de/Visual/software/lic`.

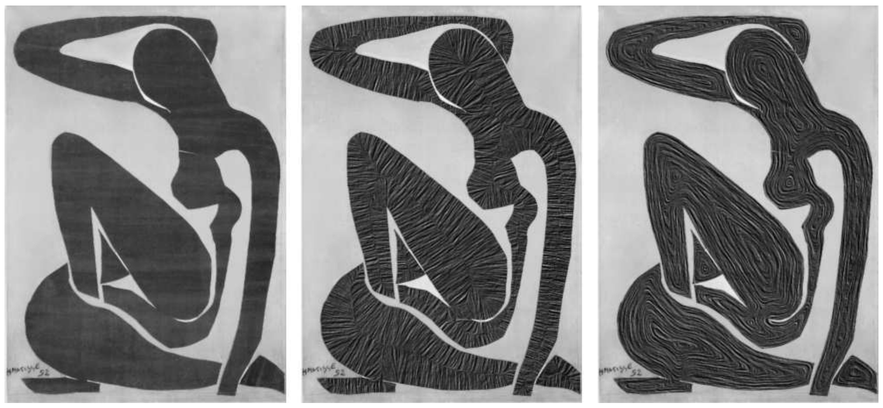


Figure 1: Scissors cut by Henri Matisse. By defining artificial vector fields on basis of the original image (left), variations (mid, right) may be produced with LIC.

directional information is specified in form of a vector field. A schematic view of the steps to be performed when generating a LIC image is shown in Fig. 2.

The material of this lecture has been organized as follows. First we give a continous formulation of LIC, neglecting the discreteness of all objects. For this we present in Sect. 2 some basics about vector fields, field lines, and line convolution. In Sect. 3 algorithms for fast and flexible generation of LIC images are presented. This comprises computation of field lines (including field interpolation, field integration, field line representation and stepping along lines), and computation of LIC integrals. Here various topics are addressed, like exploitation of redundancies, design of a fast LIC algorithm, strategies for seed point selection, and adaptive control of field line lengths. Sect. 4 explains how the resolution independence can be exploited in practice. In Sect. 5 we propose methods for animating LIC textures with constant and space dependent velocity. Several parallel algorithms that speed up generation of LIC images are presented in Sect. 6. The lecture concludes with some remarks and an outlook.

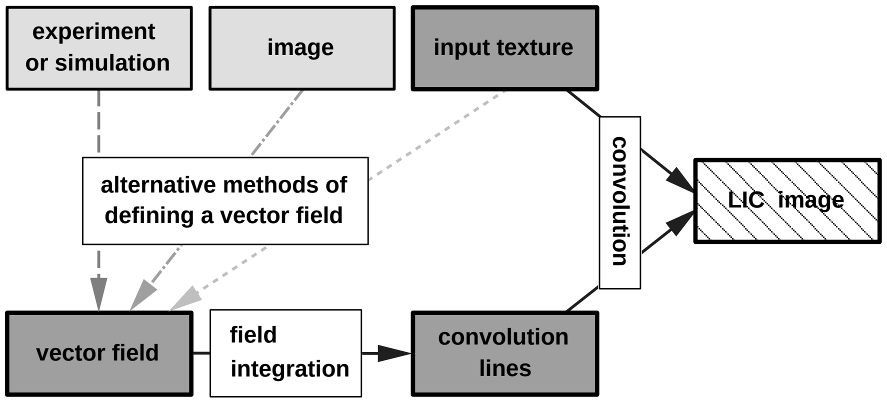


Figure 2: The main ingredients of the LIC algorithm are the input texture (some noisy image) and a dense set of lines along which the input texture is convolved. The lines are calculated on basis of a vector field. This field is provided either directly by the user or is calculated from an image – which may be the input texture or some other image.

2 Continous Formulation of LIC

Although images are discrete objects and all computations are performed in discrete domains, it is operationally convenient to consider LIC as a continous problem first. Afterwards, various discretized algorithms shall be designed to approximate the continous solution.

2.1 Vector Fields and Field Lines

We are concerned with line integral convolution in flat 2D space. Therefore we start with a *vector field* $\boldsymbol{v}: G \to \mathbb{R}^2$ that associates a vector $\boldsymbol{v}(\boldsymbol{x})$ with each point $\boldsymbol{x}$ in a 2D domain G. The directional structure of this field can be graphically depicted by its *field lines*, also called *integral curves* or *stream lines* (see Fig. 3). The relation between vector field and field lines is most illustrative in

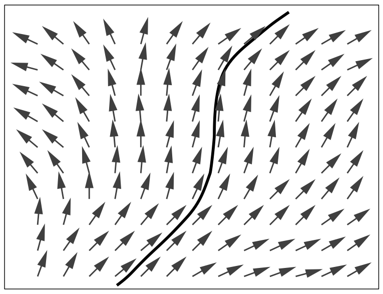


Figure 3: With each vector field we can associate field lines. These are curves whose tangent vector coincide with the vector field at each point.

the fluid flow metaphor. Imagine the vector field to be a velocity field of a stationary fluid flow. Interspersing a small dust particle in it, this will move with time. The motion can be described by a variable position vector $\boldsymbol{\tau}(t)$. For a finite interval of t, say $t \in [t_0, t_1]$, the particle traces out a curve $\boldsymbol{\tau}(t)$. This is a stream line.

If t and t' are two points of time, the chord joining the corresponding points is the vector $\boldsymbol{\tau}(t') - \boldsymbol{\tau}(t)$. Now we look at the limit $t' \to t$. Dividing $\boldsymbol{\tau}(t') - \boldsymbol{\tau}(t)$ by $t' - t$ the length of the vector is kept finite as $t' \to t$. In the limit this is $\frac{d}{dt}\boldsymbol{\tau}(t)$, the *tangent vector* of the curve at point $\boldsymbol{\tau}(t)$. In the fluid flow picture this is just the velocity vector of the particle.

The shape of a streamline $\boldsymbol{\tau}(t)$ is determined by the local field directions: The particle moves along a path $\boldsymbol{\tau}(t)$ whose *tangent vectors* coincide with the vector field at each point of time:

$$\boxed{\frac{d}{dt}\boldsymbol{\tau}(t) = \boldsymbol{v}(\boldsymbol{\tau}(t))\ .} \tag{1}$$

Because $\boldsymbol{\tau}$ is a 2D vector, this is a system of two ordinary differential equations[2] (ODEs). To calculate a specific field line, we have to single out one of the many possible lines in G, by specifying a point $\boldsymbol{x}$ where the particle has been, say at time $t = t_0$. In mathematical terminology: we have to provide an initial condition $\boldsymbol{\tau}(t_0) = \boldsymbol{x}_0$. It can be proved that there is a *unique* solution if the right hand side $\boldsymbol{\tau}$ locally obeys a so-called Lipschitz-condition. This condition is fulfilled in particular for any right hand side with continuous first derivative. Otherwise, there may exist *multiple* solutions at a single point $\boldsymbol{x}$, i.e. multiple stream lines may start or end at that point. Points where the

[2]Since the right-hand side does not depend on t explicitely, this equation represents a so-called autonomous system.

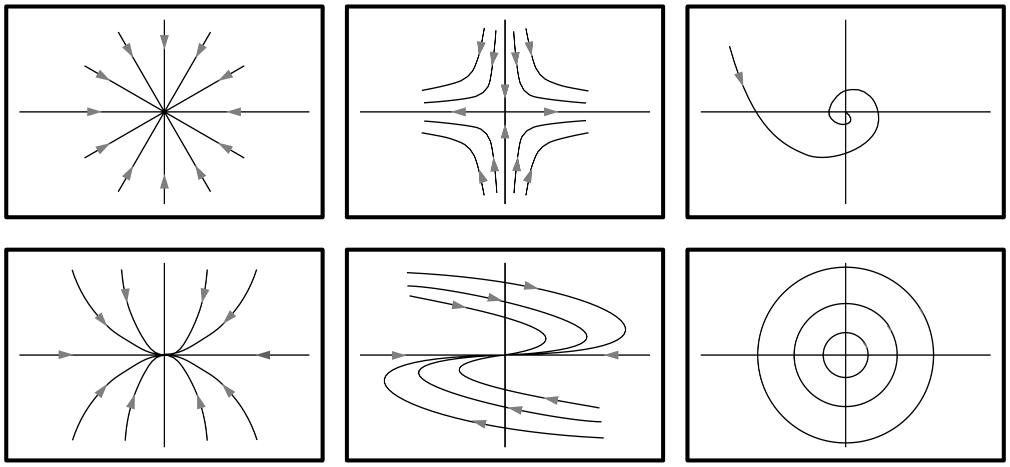


Figure 4: Different kinds of kinds of ciritical points in a 2D vector field (sinks, saddle point, improper node, spiral point, center).

vector field $\boldsymbol{v}$ vanishes are called critical points. If $\boldsymbol{a}$ is a critical point then the constant vector $\boldsymbol{\tau}(t) \equiv \boldsymbol{a}$ is a solution of Eq. (1). From a physical viewpoint, critical points correspond to rest: dust particles interspersed in a fluid flow come to rest at these points. In critical points often many field lines meet. They can be classified according to their topology, s. Fig. 4, and play an important role for the overall structure of a vector field. In the context of visualization, critical points have been analyzed by J. Helman and L. Hesselink, see [8, 9].

Beside critical points also discontinuities occur quite often in vector fields. Usually these are encountered across the boundaries of distinctly characterized field regions, e.g. in electromagnetic fields regions with different materials. An example is shown in Fig. 5. Numerical integrators used in LIC have to be robust enough to handle critical points and such discontinuities.

The variable t is just a parameter, which may be thought of as the time. Like any path, $\boldsymbol{\tau}(t)$ can be reparametrized without changing its shape and orientation, by a continous, strictly increasing

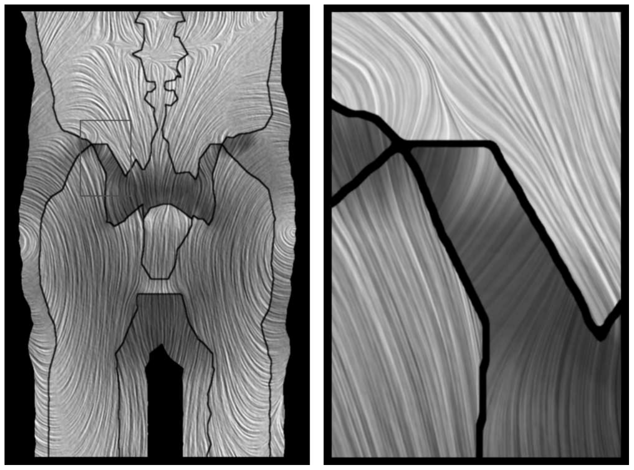


Figure 5: LIC image of a vector field (electrical field) containing *discontinuities*. Field strength v, indicated by grey level, and field direction change abruptly.

function of t. A natural parameter with these properties is arc-length[3]. From this point on we will use s as the parameter, since this makes it easy to step along a field line with evenly spaced steps. Applying the chain rule, and using the fact that the derivative of an inverse function is the reciprocal of the derivative of the original function, we get

$$\frac{d\boldsymbol{\tau}(t(s))}{ds} = \frac{d\boldsymbol{\tau}}{dt}\,\frac{dt}{ds} = \frac{d\boldsymbol{\tau}}{dt}\left(\frac{ds}{dt}\right)^{-1} = \frac{\boldsymbol{v}(\boldsymbol{\tau}(t(s)))}{v(\boldsymbol{\tau}(t(s)))}\,.$$

In the last step we have used Eq. (1) and the relation $ds/dt = v$. Introducing a new function $\boldsymbol{\sigma}(s) = \boldsymbol{\tau}(t(s))$, we get the alternative definition of field lines

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

with initial condition $\boldsymbol{\sigma}(t_0) = \boldsymbol{x}_0$. The right hand side in Eq. (2) is just the normalized vector field. Obviously, this reparametrization is valid only in regions of non-vanishing v, i.e. aside from critical points.

2.2 Line Convolution

Given a field line $\boldsymbol{\sigma}$ and an input texture T, line integral convolution at point $\boldsymbol{x}_0 = \boldsymbol{\sigma}(s_0)$ consists in computing the intensity

$$\boxed{I(\boldsymbol{x}_0) = \frac{1}{Z}\int\limits_{s_0-L}^{s_0+L} k(s-s_0)\,T(\boldsymbol{\sigma}(s))\,ds} \tag{3}$$

where Z is just a normalization factor,

$$Z = \int\limits_{-L}^{L} k(s)\,ds\,, \tag{3a}$$

and filter length L is a free parameter. That is, a 1D convolution is performed along the field lines $\boldsymbol{\sigma}(s)$ of the vector field $\boldsymbol{v}$. To generate a LIC image we have to compute at least one such intensity for each pixel. However, in order to reduce aliasing effects, it might also be useful to calculate several intensity values for different positions $\boldsymbol{x}_0$ within a pixel and average these values.

The convolution operation increases the intensity correlations along field lines. Perpendicular to such lines intensity correlations are also enhanced, due to the averaging effect of convolution and the sampling of finite texture pixels — but much less. This *asymmetry of intensity correlations* lets the vector field's directional structure become visible.

The shape of the filter kernel k obviously affects the resulting image. To our knowledge the influence of filter kernel types on LIC textures has not been analyzed systematically up to now. Are there any guidelines on how to choose a filter kernel k ? Of course, this depends on the effects one wants to achieve. If one doesn't like non-local effects due to contributions from far away pixels,

[3]If t is the time, the tangent vector $\frac{d\tau}{dt}$ is just the velocity vector of the particle and $(\frac{d\tau}{dt}\cdot\frac{d\tau}{dt})^{1/2} = v$ its amount. Noticing $v = ds/dt$, the arc-length from any given point $t = t_0$ is $s(t) = \int_{t_0}^{t} v(t')\,dt'$.

the support of k, i.e. the domain where the function k does not vanish, should be concentrated around zero. Furthermore, if there are no specific reasons for introducing an anisotropy, k should be symmetric around zero. Anisotropies also let errors due to inaccurate field line computation become more visible [15]. The next question is, whether k should vary or even take negative values, e.g. like $k(r) \propto f(|r|)\,(1 + \cos(\alpha r))$ or $k(r) \propto f(|r|)\cos(\alpha r)$ respectively, where f is a positive, non-increasing function. There are reasons for using filter kernels of the first kind in animated LIC sequences, see Ref. [4] and Sect. 5.1. However, we will see that the use of complicated filter kernels can be circumvented by using a different animation technique. Therefore, in our example implementation we chose the computationally least expensive filter that fits to all our needs (see Sect. 3.2.1). This is a constant filter, or due to the finite L, a “box filter”.

A further question is what filter length one should choose. Of course noisiness in the output image is reduced with growing L. As a very rough advise, appealing results are usually obtained by choosing 10 to 40 pixel lengths for L.

3 Fast and Flexible Generation of LIC Images

During the algorithm design we will pursue three major goals. First, vector field, input texture, and output image should be handled as independent entities, possibly with different spatial resolutions. Second, wherever possible, redundancies should be avoided during calculation. Third, very efficient algorithms should be employed in each computational subtask.

In order to evaluate Eq. (3) we have first to compute the field lines $\boldsymbol{\sigma}(s)$ according to Eq. (2) and then to perform convolution. It seems natural to decouple these two steps. The connecting links are the field lines. These will be represented internally in a resolution independent way. Each field line covers lots of image pixels. This can be exploited by using each line to calculate intensity values of many pixels.

3.1 Computation of Convolution Lines

Field line computation itself can be divided into four algorithmic steps: interpolating the vector field, integrating it, establishing a continuous representation for field lines, and stepping along field lines. In the following all these issues will be discussed in detail.

3.1.1 Field Interpolation

Typically the vector field has been sampled at discrete locations in G. We assume that an underlying function $\boldsymbol{v}$ exists on G, which is at least locally continous. If there are known discontinuities in the vector field, e.g. if boundaries of geometrical objects immersed in flow or electrical fields, adequate provisions should be taken in the following steps. Concerning the continous regions of the field we assume that it is bandwidth limited and has been sampled at or above the Nyquist frequency.

During the field line calculation we will need vector field values at arbitrary locations $\boldsymbol{x} \in G$. To reconstruct vector values at intermediate locations we have to interpolate somehow. In the example implementation we assume $\boldsymbol{v}$ to be given at discrete locations of an uniform rectangular grid. Of course, in a versatile application program also other grids like non-uniform triangular grids should be supported, too. In order to reconstruct vector values at intermediate locations we use bilinear interpolation in our demo implementation. Better reconstruction schemes can be employed if appropriate information is available about the underlying function. In the following we just assume a function to be available that returns vector values at arbitrary positions in G.

Sometimes global field properties are known, like the existence of closed field lines. In general such properties are not preserved when a local interpolation scheme like bilinear interpolation is used – even with perfect field integration. In particular closed field lines in the true vector field may no longer be closed in the interpolated field [11]. For two reasons this is not very critical in our application. First, deviations of the computed field lines from the exact ones are less visible in LIC images than in images showing isolated field lines. Second, if the Nyquist sampling condition is fulfilled, errors due to interpolation are usually much smaller than errors caused by a poor numerical integrator.

3.1.2 Field Integration

Solving an ordinary differential equation like Eq. (2) is often called "integration". We will follow this usage. Many different methods have been developed to solve such initial value problems [12, 7, 5].

For choosing an appropriate integrator, we have to take into account that the field values are not very smooth. Bilinear interpolation, for instance, results in a representation of the field that is continous but not differentiable across the boundaries of grid cells. Therefore, we cannot rely on sophisticated ODE integrators which require a very smooth right hand side. Instead we propose to employ a Runge-Kutta method with error monitoring and adaptive step size control. A general problem especially with good integrators is, that they proceed with rather large steps and therefore are susceptible to miss fluctuations that may exist in otherwise homogeneous regions. This can be tackled by delimiting the maximum allowed step size – if necessary down to the sampling distance of the vector field. Having this in mind, the idea might arise to use analytical expressions for the field lines inside a grid cell. In fact, for a linearly interpolated vector field such formulae can be found. Their evaluation requires to compute eigenvalues and eigenvectors of a coefficient matrix and to evaluate transcendental functions. A faster and much versatile method, that also can be combined with better interpolation schemes (bilinear, quadratic, ...), is to use a robust numerical integrator.

Runge-Kutta Methods. For the moment assume we have found an exact solution of the ordinary differential equation Eq. (2). It may formally be expressed by the so-called continuous propagator ϕ^h which allows us to step along a field line $\boldsymbol{\sigma}(s)$:

$$\boldsymbol{\sigma}(s+h) = \phi^h \boldsymbol{\sigma}(s).$$

A numerical integrator may be thought of as an approximation of the true propagator. Given such an approximation $\hat{\phi}^h$, successive points on the field line can be computed recursively from the initial value $\boldsymbol{x}_0$:

$$\boldsymbol{x}_{n+1} = \hat{\phi}^{h_{n+1}} \boldsymbol{x}_n, \quad n = 0, 1, 2, \ldots$$

The increment h may vary from step to step as indicated by its subscript. Different integration methods can be distinguished by how the deviation from the true propagator scales with h. A method is conventionally called of n-th order if its error term is $O(h^{n+1})$. The most simple integration scheme is Euler's method,

$$\hat{\phi}^h \boldsymbol{x} = \boldsymbol{x} + h\boldsymbol{f}(\boldsymbol{x}), \tag{4}$$

which is of first order only. A better approximation can be obtained by taking an intermediate step halfway across the interval, then using the midpoint derivative to determine the new solution:

$$\hat{\phi}^h \boldsymbol{x} = \boldsymbol{x} + h\boldsymbol{f}(\boldsymbol{x} + \frac{1}{2}h\boldsymbol{f}(\boldsymbol{x}))\,. \tag{5}$$

This so-called midpoint method is of second order, as may be verified by expansion in power series. By taking more intermediate steps and combining all these to obtain the final solution, methods of higher order can be constructed. This is the basic idea of so-called explicit Runge-Kutta methods [10]. Given a set of coefficients a_{ij}, s terms are recusively calculated,

$$\boldsymbol{k}_i = h\boldsymbol{f}(\boldsymbol{x} + \sum_{j=1}^{i-1} a_{ij}\boldsymbol{k}_j), \quad i = 1, \dots, s$$

and combined to give the final solution

$$\hat{\phi}^h \boldsymbol{x} = \boldsymbol{x} + \sum_{i=1}^{s} b_i\, \boldsymbol{k}_i\,,$$

where the coefficients b_i must satisfy the condition $\sum_i b_i = 1$. A well-known example established in this way is the classical fourth-order Runge-Kutta formula, which requires four $\boldsymbol{f}$-evaluations:

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

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

For higher order formulas more $\boldsymbol{f}$-evaluations are needed. If the additional computational expense can be compensated by a correspondingly bigger step size h, higher order methods are worth the effort.

Adaptive Step Size Control. A black box integrator should return a solution with some user-defined accuracy *tol* and with minimum computational effort. This can be achieved by controlling adaptively the step size h on basis of the *actual* value of the integration error.

Assume that we are able to estimate the local integration error Δ. Using a p-th order integration method, the error term scales as h^{p+1}. Therefore if a step size h results in an error Δ, an optimized step size h^* can be obtained by

$$h^* = h \sqrt[p+1]{\rho\,\frac{tol}{\Delta}}\,, \tag{7}$$

with some safety factor $\rho < 1$. This formula can be utilized to set up a control mechanism as follows: If for the current step the error estimate Δ is bigger than *tol*, repeat it with $h = h^*$. Otherwise, proceed and take $h = \min(h^*, h_{\max})$ for the next iteration, where $h_{\max}$ is the maximum allowed step size. If h becomes much smaller than the spacing between the sampled $\boldsymbol{f}$ values, assume that a singularity has been encountered and terminate field line integration.

It should be noted that the total discretization error not only consists of a local error caused by the propagation $\hat{\phi}^h \boldsymbol{x}_n$, but also of the accumulated error from all previous steps. However, controlling this error may require recomputation of the entire field line. For that reason this term is usually neglected – an excuse which might be acceptable for an uncritical application like LIC.

The remaining question is, how to compute the local error estimate Δ. The usual way to obtain such an estimate is to compare two discrete solutions of different order, a more accurate $\hat{\phi}^h \boldsymbol{x}$ and a less accurate one $\bar{\phi}^h \boldsymbol{x}$. The difference of these two solutions is an approximation of the less accurate solution's error, $\Delta \approx \hat{\phi}^h \boldsymbol{x} - \bar{\phi}^h \boldsymbol{x}$. Fortunately, it can be shown [5] that in many cases this approximative error can safely be used to control the step size of the more accurate integrator.

Instead of computing two completely different discrete solutions some of the intermediate steps $\boldsymbol{k}_i$ in Eq. (6) may be reused. Corresponding methods are known as embedded Runge-Kutta formulas. Another trick due to Fehlberg is to use the derivative $\boldsymbol{f}$ at the resulting location $\hat{\phi}^h \boldsymbol{x}$ in both the current and the following step. In this way from the classical fourth-order Runge-Kutta formula (6) a third-order approximation can be constructed, namely

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

This leads to a very simple expression for the error estimate:

$$\Delta = \hat{\phi}^h \boldsymbol{x} - \bar{\phi}^h \boldsymbol{x} = \frac{1}{6}(\boldsymbol{k}_4 - \boldsymbol{f}(\hat{\phi}^h \boldsymbol{x}))\,. \tag{8}$$

The resulting adaptive integrator, denoted as RK4(3) hereafter, turns out to be very robust and well suited for our application. For practical experiences with higher order methods in LIC applications see [14].

3.1.3 Representation of Field Lines

Using an ODE solver as discussed above we are able to compute field lines with high accuracy. However, the step sizes used by the integrator typically are so large that the distance between two successive field line locations $\boldsymbol{x}_n$ and $\boldsymbol{x}_{n+1}$ is much bigger than the distance h_t needed for texture sampling. Average distances from 10 to 30 times the spacing of the $\boldsymbol{v}$ grid are quite common in practice. Therefore, we represent the field line by a polynomial which interpolates between successive locations that have been returned by the integrator.

As illustrated in Fig. 6, the distance between successive locations and the curvature of the field line may easily take values such that linear interpolation is not sufficient. Linear interpolation would also not take into account the most accurate information we have, namely the tangents of the field line. In fact it is a general result, c.f. [7], that the outcomes of a fourth-order integrator as RK4(3) are adequately interpolated by a cubic Hermite polynomial:

$$\boldsymbol{p}(u) \;=\; \boldsymbol{a}u^3 + \boldsymbol{b}u^2 + \boldsymbol{c}u + \boldsymbol{d} \tag{9}$$

where we used a rescaled parameter $u \in [0, 1]$

$$u \;=\; \frac{s - s_n}{s_{n+1} - s_n} \tag{9a}$$

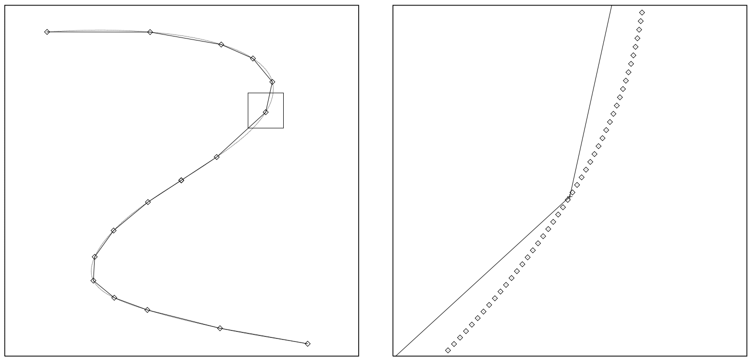


Figure 6: Successive points of a field line, as returned by adaptive numerical integrators of higher order, are usually so large that also a higher order interpolation is necessary to track the path for texture sampling.

and coefficients

$$\begin{aligned} \boldsymbol{a} &= 2\boldsymbol{p}(0) - 2\boldsymbol{p}(1) + \boldsymbol{p}'(0) + \boldsymbol{p}'(1) \\ \boldsymbol{b} &= -3\boldsymbol{p}(0) + 3\boldsymbol{p}(1) - 2\boldsymbol{p}'(0) - \boldsymbol{p}'(1) \\ \boldsymbol{c} &= \boldsymbol{p}'(0) \\ \boldsymbol{d} &= \boldsymbol{p}(0)\,. \end{aligned} \tag{9b}$$

The vectors $\boldsymbol{p}$ and $\boldsymbol{p}'$, expressed in terms of field line position and orientation at the boundaries of the interpolation interval, are:

$$\begin{aligned} \boldsymbol{p}(0) &= \boldsymbol{x}_n & \boldsymbol{p}'(0) &= (s_{n+1}-s_n)\,\boldsymbol{f}(x_n) \\ \boldsymbol{p}(1) &= \boldsymbol{x}_{n+1} & \boldsymbol{p}'(1) &= (s_{n+1}-s_n)\,\boldsymbol{f}(x_{n+1})\,. \end{aligned} \tag{9c}$$

This explicitely shows that the boundary points of the interpolating polynomial are defined by two sucessive field line points. The tangents of the polynomial coincide at these points with the vector field directions.

3.1.4 Stepping along Field Lines

To compute LIC integrals, the cubic interpolation polynomial has to be evaluated at many points. Since these sample points are evenly spaced, we can quickly track a field line by employing a forward difference scheme:

$$\begin{aligned} \Delta^1\boldsymbol{p}(u) &= \boldsymbol{p}(u+\delta) - \boldsymbol{p}(u) \\ &= 3\boldsymbol{a}\delta u^2 + (3\boldsymbol{a}\delta^2 + 2\boldsymbol{b}\delta)u + \boldsymbol{a}\delta^3 + \boldsymbol{b}\delta^2 + \boldsymbol{c}\delta \\[1em] \Delta^2\boldsymbol{p}(u) &= \Delta^1\boldsymbol{p}(u+\delta) - \Delta^1\boldsymbol{p}(u) \\ &= 6\boldsymbol{a}\delta^2 u + 6\boldsymbol{a}\delta^3 + 2\boldsymbol{b}\delta^2 \\[1em] \Delta^3\boldsymbol{p}(u) &= \Delta^2\boldsymbol{p}(u+h) - \Delta^2\boldsymbol{p}(u) \\ &= 6\boldsymbol{a}\delta^3 \;=\; \text{const.} \end{aligned} \tag{10}$$

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

$$\begin{aligned} \boldsymbol{p}(u_{k+1}) &= \boldsymbol{p}(u_k) + \Delta^1 \boldsymbol{p}(u_k) \\ \Delta^1 \boldsymbol{p}(u_{k+1}) &= \Delta^1 \boldsymbol{p}(u_k) + \Delta^2 \boldsymbol{p}(u_k) \\ \Delta^2 \boldsymbol{p}(u_{k+1}) &= \Delta^2 \boldsymbol{p}(u_k) + \Delta^3 \boldsymbol{p}(u_k) \,. \end{aligned} \tag{11}$$

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

$$au^3 + bu^2 + cu + d = ((a * u + b) * u + c) * u + d.$$

Note, that we cannot assume $u_0 = 0$, because the distance between two neighbouring positions returned by the integrator in general is not a multiple of h_t. Instead, the remainder of h_t which doesn't fit into the previous interval anymore serves as the initial offset for the next interval.

3.2 Computation of LIC Integrals

The most important point is that many integrals have to be computed – at least one for each pixel of the output image. Hence the major goals are to accelerate the computation of individual integrals, as well as to utilize coherences along particular convolution lines. Furthermore, the numbers of field lines and convolution integrals to be computed should minimized. This can be accomplished by two steps: first, smartly choosing the seed points where field integration is started and, second, adapting the length of field line segments to the actual coverage.

3.2.1 Exploiting Redundancies

Stepping along a stream line and computing intensity values for neighbored pixels, rather similar intensity values result. This is most obvious and also very simple to exploit if a *constant* filter kernel k is chosen. Consider two points located on the same field line, $\boldsymbol{x}_1 = \boldsymbol{\sigma}(s_1)$ and $\boldsymbol{x}_2 = \boldsymbol{\sigma}(s_2)$ with distance $\Delta s = s_2 - s_1$. Then for a constant and normalized filter kernel $k' = \frac{k}{Z} = \frac{1}{2L}$ Eq. (3) can obviously be written as

$$I(\boldsymbol{x}_2) = I(\boldsymbol{x}_1) \; - \; k' \int_{s_1-L}^{s_1-L+\Delta s} T(\boldsymbol{\sigma}(s))\, ds \; + \; k' \int_{s_1+L}^{s_1+L+\Delta s} T(\boldsymbol{\sigma}(s))\, ds \,. \tag{12}$$

The intensities differ only by small correction terms from the ends of the integration interval. Using a constant filter kernel therefore has *two* advantages: evaluation of the LIC integral (3) is very fast, and, after computing it for some initial location $\boldsymbol{x}_0$, further intensity values on the particular field line can be obtained by calculating just the correction terms. For this reason we will use *constant* filter kernels k, or equivalently, box filters of length $2L$.

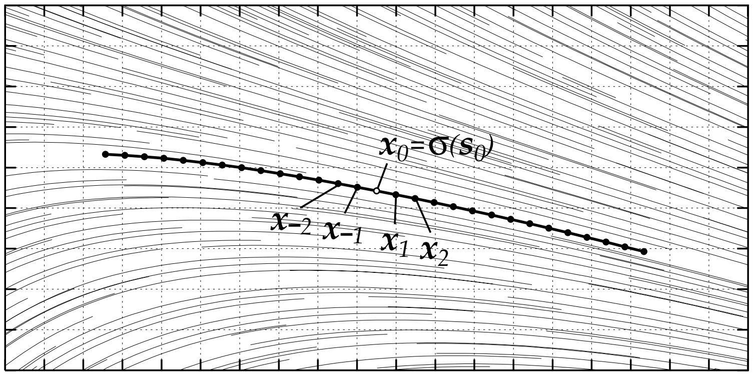


Figure 7: The input texture is sampled at evenly spaced locations $\boldsymbol{x}_i$ along a field line $\boldsymbol{\sigma}$. For each location the convolution integral $I(\boldsymbol{x}_i)$ is added to the pixel containing $\boldsymbol{x}_i$. A new field line is computed only for those pixels where the number of samples does not already exceed a user-defined limit.

3.2.2 Putting it Together: A Fast LIC Algorithm

We allocate a hit count and accumulation buffer to store the number of hits and the accumulated intensity value for each pixel of the output image. To compute a LIC image we proceed in the following way.

First we initiate field line computation for some location $\boldsymbol{x}_0 = \boldsymbol{\sigma}(s_0)$. The convolution integral for this location is approximated by sampling the input texture T at $2N_f+1$ evenly spaced locations along the precomputed field line $\boldsymbol{\sigma}(s)$:

$$I(\boldsymbol{x}_0) = \frac{1}{2N_f+1} \sum_{i=-n}^{n} T(\boldsymbol{x}_i), \qquad \text{with } \boldsymbol{x}_i = \boldsymbol{\sigma}(s_0 + i\frac{L_f}{N_f}). \tag{13}$$

One could calculate nearly as fast more accurate trapezoidal sums by multiplying the first and last term with 0.5, but this does not pay visually. Typical values of L_f comprise 10 to 40 pixel lengths. Samples should be taken with an increment of half a pixel in the input texture. Larger sampling distances generally cause aliasing. The resulting intensity is added to the accumulated intensity of the output image pixel containing point $\boldsymbol{x}_0$ and the corresponding hit counter is incremented.

Then we step along the current field line to calculate recusively further intensity values via

$$I(\boldsymbol{x}_{i\pm1}) = I(\boldsymbol{x}_i) \ - \ \frac{T(\boldsymbol{x}_{i\mp N_f})}{2N_f+1} + \frac{T(\boldsymbol{x}_{i\pm(N_f+1)})}{2N_f+1} \tag{14}$$

for N_s samples in both directions. A strategy for choosing N_s is discussed in Section 3.2.4. For each point $\boldsymbol{x}_i$ the corresponding output image pixel is determined, its accumulated intensity is updated, and its hit counter is incremented. This way we efficiently obtain intensities for $2N_s+1$ pixels covered by the same field line (see Fig. 7). We run through all output image pixels according to some strategy (being described in Sect. 3.2.3). For each pixel, we require the total number of hits to be larger than some minimum. If the number of hits is smaller than the minimum, a new field line computation is initiated. Otherwise that pixel is skipped. At the end all accumulated intensities are normalized against the number of individual hits. This algorithm can be described by the following pseudocode:

Algorithm fastLIC

```
allocate buffers numHits and accumInt
choose pixel $p$ according to some strategy
    if (numHits($p$) <minNumHits) then
        initiate field line computation with $\boldsymbol{x}_0$ = center of $p$
        compute convolution $I(\boldsymbol{x}_0)$
        add result to pixel $p$
        set $i = 1$
        while $i <$ some limit $N_s$
            update convolution to obtain $I(\boldsymbol{x}_i)$ and $I(\boldsymbol{x}_{-i})$
            determine pixels $p_i$ and $p_{-i}$ that contain $\boldsymbol{x}_i$ and $\boldsymbol{x}_{-i}$
            add results to accumInt($p_i$) and accumInt($p_{-i}$)
            increment numHits($p_i$) and numHits($p_{-i}$)
            set $i = i + 1$
until each pixel has been hit at least minNumHit times
for each pixel $p$
    normalize accumulated intensity accumInt($p$) according to numHits($p$)
```

Some additional remarks:

- The resolution of input texture and output image can be chosen independently. This is necessary for zooming, c.f. Sect. 4.
- In most applications it is sufficient to choose minNumHits $= 1$.
- The computation of field line segments is performed without referencing input texture or output image. This allows us to utilize powerful integration methods that accelerate field line computation significantly in homogeneous regions.
- Using a single field line segment for many pixels instead of computing these for each pixel separately, effectively increases the pixel intensity correlation along the field line, since exactly the same sampling points are used for convolution.
- The order in which the output image pixels are processed influences the efficiency of the algorithm. The goal is to hit as many uncovered pixels as possible with each new field line. Optimization strategies are discussed in Sect. 3.2.3.
- In order to have a full sized convolution range for all pixels we need to extend the field lines and the input texture beyond the boundary of G. Therefore, when a field line touches the domain of $\boldsymbol{v}$, we continue the path in the current direction. We continue field lines in a similar way if $|\boldsymbol{v}|$ vanishes or if a singularity was encountered. For texture sampling all points are remapped, e.g. by continuing the texture periodically or mirroring it at the boundary.

3.2.3 Seed Point Selection

If two field line calculations are started with seed points $\boldsymbol{x}_0$ that are close to each other, both lines are likely to hit the same pixels. This lowers the overall efficiency of the algorithm. Therefore we are interested in an inexpensive strategy for selecting seed points such that the image is covered with a minimum number of lines.

In the following we will compare three different strategies for selecting seed points. The first algorithm chooses seed points in scanline order. The second algorithm divides the image into 4x4 blocks, taking a first pixel out of each block, then a second one, and so on. The third strategy uses Sobol quasi random sequences [2, 13].

A Sobol sequence is a permutation $\boldsymbol{x}: i \leftrightarrow x_i$ with $i, x_i \in \{1, \dots, n_x\}$ that results in a point set whose discrepancy is smaller than one would expect for a truly random set. Such sequences can be generated by number theoretic methods. In its original form the length of Sobol sequences is a power of two. Sequences of arbitrary length are obtained simply by discarding numbers. Given two such sequences $\boldsymbol{x}$ and $\boldsymbol{y}$ of equal length $n_x = n_y$, all pixels of a square image can be accessed in quasi-random order by the sequence

$$\begin{array}{l} (x_1, y_1), (x_2, y_2), \dots, (x_{n_x}, y_{n_y}), \\ (x_1, y_2), (x_2, y_3), \dots, (x_{n_x}, y_1), \\ \dots \\ (x_1, y_n), (x_2, y_1), \dots, (x_{n_x}, y_{n_y-1}) \end{array} \tag{15}$$

For non-square images with $n_x < n_y$ a similar numbering can be obtained by utilizing an extended sequence $\bar{\boldsymbol{x}}$ with $\bar{x}_i = x_{i \bmod n_x}$, $i \in \{1, \dots, n_y\}$. If $n_x > n_y$ then $\boldsymbol{y}$ is extended instead of $\boldsymbol{x}$. For better performance both sequences may be tabulated.

In Fig. 8 pixel coverages are shown that are obtained after 120, 240 and 360 field lines have been computed according to one of the three strategies. Grey levels encode the number of hits per pixel. The Sobol technique obviously achieves higher pixel coverage for a given number of field lines. The total number of samples is usually about 10-20% less, compared to the block iteration method. For different computer architectures we found the following performance results:

	SGI Indigo^2 R4400 150 MHz		SGI Indy R4600PC 100 MHz		Cray T3D (1 DEC Alpha processor)	
	work	computing time	work	computing time	work	computing time
block	100%	4.83 sec	100%	7.30 sec	100%	11.42 sec
sobol	88%	5.59 sec (115%)	88%	6.95 sec (95%)	88%	10.87 sec (95%)

This shows that the Sobol scheme only pays on machines without second level cache (here: SGI Indy R4600PC and Cray T3D). Accessing large blocks of memory in quasi-random order on second level cache machines apparently causes too many cache misses, which may completely outweigh the benefits of the selection strategy.

3.2.4 Adaptive Field Line Length

To obtain the convolution integral for a seed point $I(\boldsymbol{x}_0)$, a field line segment of length $2L_f$ is needed. In order to take advantage of the fast update scheme Eq. (14), we want to continue the field line some distance L_s in both directions. At a first glance performance should be best when making L_s as large as possible. However, this is not true for most vector fields: In fields containing sinks and sources, many field lines will run toward these points, producing lots of hits for a small set of pixels, until the integrator detects the singularity and stops field line integration. In fields containing vortices, a field line could turn around these many times without ever hitting new pixels.

In consequence an optimal field line length exists. It depends on the vector field characteristics *and* the actual coverage. In the following we will outline a simple adaptive strategy to determine an estimate of the optimal value of L_s.

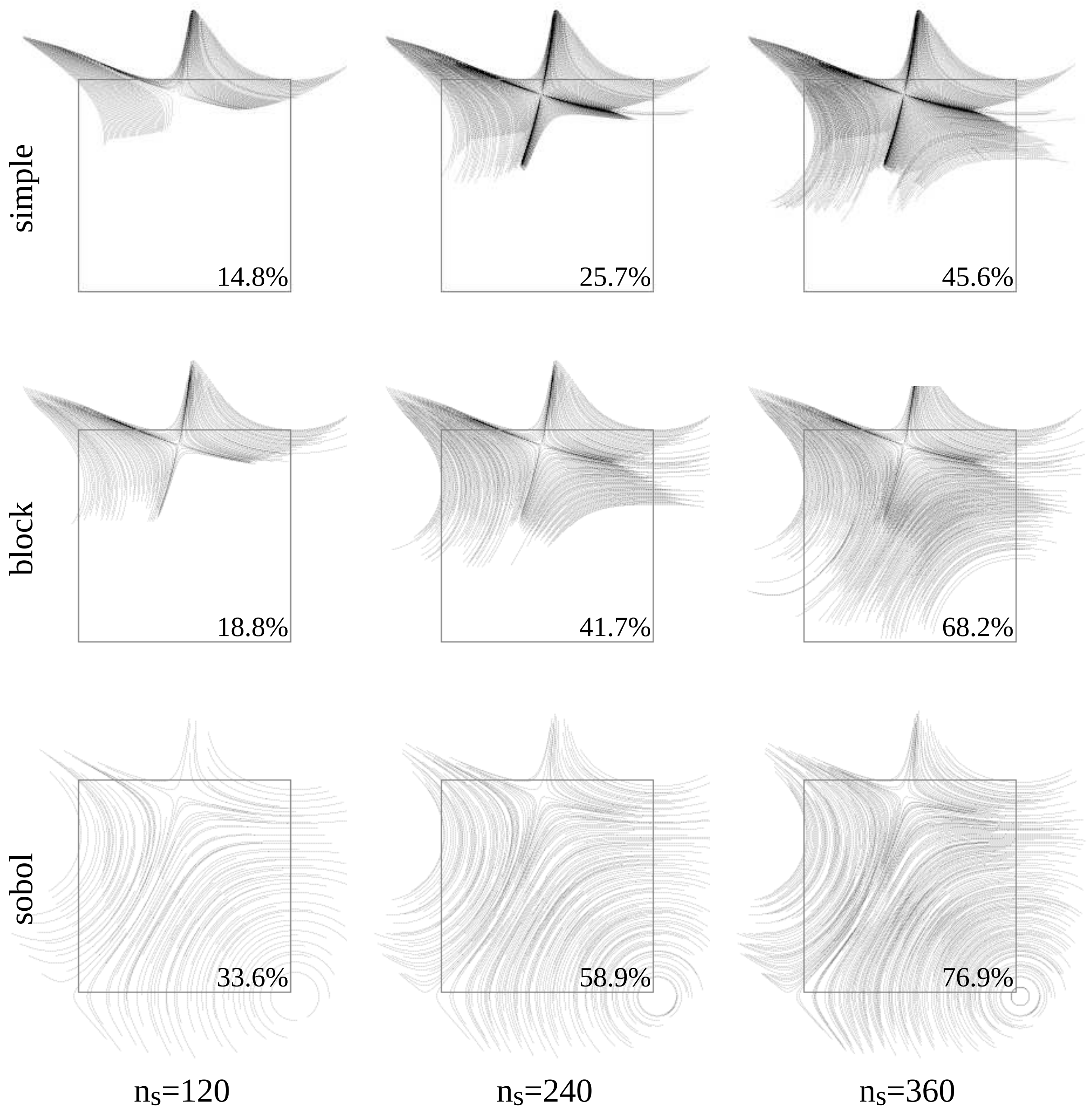


Figure 8: Pixel coverage for three different seed point selection strategies. Grey levels encode the number of hits per pixel. For a given number of field lines, the quasi-random Sobol sequence yields highest pixel coverage.

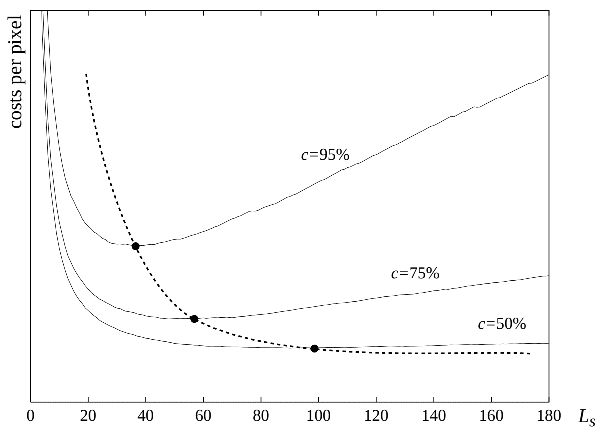


Figure 9: Costs per pixel versus length L_s of the computed field line segments. Three curves corresponding to different degrees of coverage are shown.

The costs for obtaining pixel intensities for all samples of a field line segment of length $2L_s$ can be assumed to be proportional to the actual length of the field line. This equals $2L_s$ plus an inital offset of $2L_f$ to hold the inital filter box. Therefore we have

$$C_{\text{field line}}(L_s) = \alpha(L_f + L_s) \ . \tag{16}$$

Small initialization costs for refilling the cache and starting the integrator are ignored. Sampling a field line segment of length l causes a certain number $P(l)$ of previously untouched pixels to be hit. As stated above $P(l)$ is monotonously increasing, but *not* proportional to l. Since field lines are started always on untouched pixels, at least one pixel is hit, i.e. $P(0) = 1$. The average *costs per pixel* for field lines of length L_s are then

$$C_{\text{pixel}}(L_s) = \frac{\text{costs of field line}}{\text{number of new pixels}} = \frac{\alpha(L_f + L_s)}{P(L_s)} \ . \tag{17}$$

This function will in general have a minimum, indicating the optimal value of L_s. To locate this minimum, we measure $P(l)$ for a certain number of field lines – of course, often enough to get reliable statistics. Then Eq. (17) is evaluated and the new field line size L_s is set to the optimal value. $P(l)$ is measured by keeping track of the number of new hits obtained per evaluation of Eq. (14) at a particular distance.

Fig. 9 shows the costs per pixel for three different degrees of coverage. The same data set as in Fig. 8 has been used. As expected $C_{\text{pixel}}(L_s)$ increases as more and more pixels are being covered. At the same time the optimal field line length gets shorter. Comparing an algorithm with and without adaptive length control gave the following results for a 512×512 image:

L_s	N field lines	N hits	Total costs
150 (fixed)	5200	1,300,000 (100%)	100%
200...10 (adaptive)	6670	901,000 (69.3%)	80.7%

Notice that the total number of field lines has increased due to a reduced average length. Nevertheless for the adaptive case the total costs estimated by $C_{\text{total}} \sim N_{\text{hits}} + N_f N_{\text{field lines}}$ are only about 80% of the costs of the fixed length strategy due to the smaller number of hits.

4 Resolution Independence, Zooming and All That

A key feature of the presented LIC algorithm is the mutual resolution independence of vector field, input texture, and output image. In the following we will explain how this can be exploited in practice.

To have a fixed point, we assume that the length scale is set by the domain G of the vector field. Let's say G has extension 1 in x direction, e.g. $G = [0,1] \times [0,y]$. For simplicity we assume that all pixels are square. Pixel sizes for the input texture and output image are denoted a_t and a_o. If the input texture and output image fit exactly into the whole domain G, their resolutions are $\frac{1}{a_t} \times \frac{y}{a_t}$ and $\frac{1}{a_o} \times \frac{y}{a_o}$, respectively.

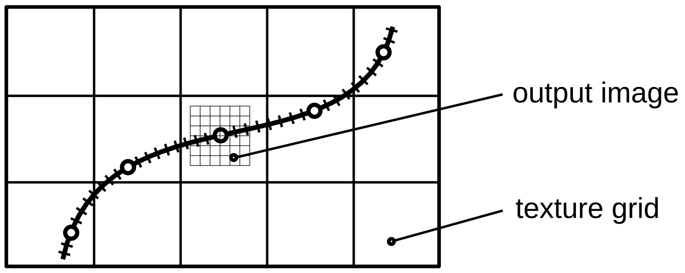
The standard situation is $a_t = a_o$. Now consider the following cases:

- **Scaling $\mathbf{a_t}$ and $\mathbf{a_o}$ simultaneously**

  The standard method to generate a LIC image with prescribed resolution a_o^{-1} is to choose $a_t = a_o$. Varying both pixel sizes simultaneously we get LIC images that display the vector field in different resolutions. Of course, the resulting LIC images are rather different if the input textures are unrelated. Even when using multi-resolution textures, where coarse resolutions have been generated by resampling finer ones, details of resulting LIC images may differ more than one would expect at a first glance. This is due to the fact that two-dimensional pixels are sampled by a strictly one-dimensional process. In general it is hard to generate "the same image at different resolutions" with this method.

- **Scaling $\mathbf{a_t}$ with fixed $\mathbf{a_o}$**

  Let us increase the texture cell size a_t, such that a single texture cell is covered by several output pixels. Though the input texture can be sampled at increments $h_t = 0.5\,a_t$, we use a smaller step size for stepping along the field line and updating the integral according to Eq. (14), in order to hit as many pixels as before:



  This procedure can be utilized to lower the highest frequencies present in the LIC image. Typically LIC images are created from high frequency textures such as white noise. High frequencies are retained in directions perpendicular to the field direction. This may cause problems if the LIC images are 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. Since our algorithm can easily compute images with large N_f, a more direct approach is to simply scale up the size of a texture cell as well as the convolution length L.

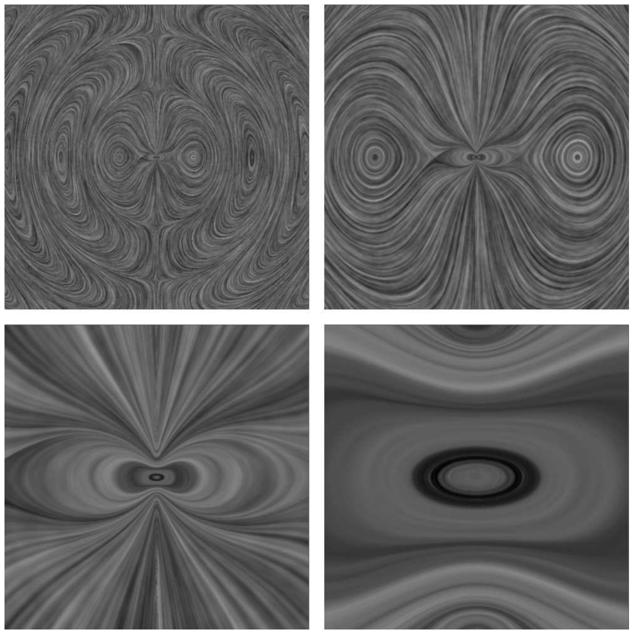


Figure 10: Details of a vector field displayed at different magnification factors (1, 3, 15, 100). For each frame a completely new LIC image has been computed. The data set had a resolution of 500^2. At the finest level only a few grid points are covered.

- **Scaling $\mathbf{a_o}$ with fixed $\mathbf{a_t}$**

  This allows to create several versions of a single image at different resolutions, e.g. adopted to various output devices. A slightly different utilization of this feature is the computation of smooth zooms into the vector field to enlarge details. As an example some close-ups of vector field details are shown in Fig. 10. The linear magnification extends up to a factor 100 in this example.

  If the zoom is to be played back as a movie, care has to be taken for low magnification factors. Starting field line integration at the center of output image pixels as usual, in each frame slightly different field lines are computed. This causes annoying variations to occur from frame to frame. One solution would be to increase the minimal number of hits required per pixel. A more robust method is to first utilize field lines that have been used also in the previous frame. 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 with striking trips into details.

5 Animating LIC Textures

LIC images can be animated by changing the shape and location of the filter kernel k over time [4]. There are two variants, LIC textures with constant, i.e. location independent velocity and those with locally varying velocity.

The first variant shows vector field *direction* in addition to the pure tangential information that is contained in static LIC images. The second variant additionally depicts field *strength*, if velocity is taken to be $|\boldsymbol{v}|$. The resulting image sequences bear great resemblance to that of a real stationary fluid flow with interspersed moving particles. LIC movies of this kind therefore provide a highly intuitive means for envisioning stationary vector fields. A process generating animated LIC textures should meet the following demands:

1. The main feature of LIC images, namely high intensity correlations along field lines and low ones perpendicular to them, should be preserved.
2. The intensity correlations between points fixed on a field line should be constant over time, to avoid disturbing artifacts.
3. The process should generate periodic sequences of LIC images in order to make long lasting animations possible.
4. The computational costs should be low.

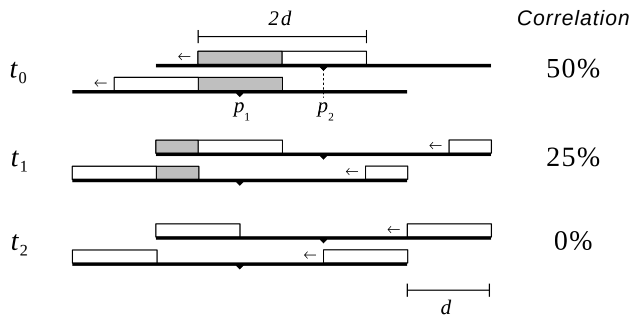
5.1 Intensity Correlations

Imagine two different points $\boldsymbol{p}_1$ and $\boldsymbol{p}_2$ in G. We assume that white noise is used as input texture. The intensities $I(\boldsymbol{p}_1)$ and $I(\boldsymbol{p}_2)$ are correlated if their filter kernels overlap. In LIC this can happen for points that have sufficiently small distance and belong to the same field line. In the following we therefore assume that $\boldsymbol{p}_1$ and $\boldsymbol{p}_2$ are points on one field line. The intensity correlation ξ between two points is proportional to the relative overlap of their filter kernels. ξ is a function of the distance $d = \|\boldsymbol{p}_1 - \boldsymbol{p}_2\|$, and, for filter kernels with varying location or shape, also a function of time:

$$\xi(d,t) = \frac{\int \min(k(s,t), k(s+d,t))\, ds}{\int k(s,t)\, ds}. \tag{18}$$

Using box filters, an obvious method to animate LIC images is to cycle the boxes through some interval along the stream lines. For this we consider the interval to be periodically continued: when the moving box reaches the boundary of the interval, it starts to reenter the interval at the other side. That means, the box is temporarily broken into two boxes. If such a cycling is done synchronously 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. (14).

The intensity correlation for a cycled box filter is shown in Fig. 11a. The length of the filter box was chosen to be 0.5 times the length of the interval. Even for nearby points, i.e. small distances d, the intensity correlation temporarily drops to zero, while at the same time distant points become correlated. This is due to the breaking of the filter box, as illustrated in the following figure:



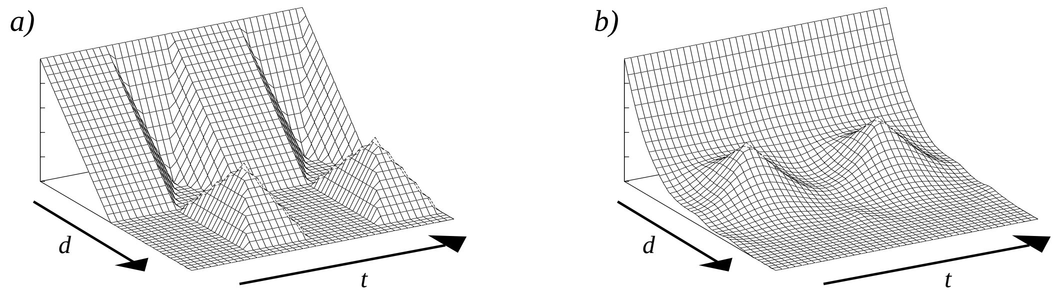


Figure 11: Intensity correlation between two points on a single field line for different motion filter kernels: box filter (a) and Hanning filter (b). Two periods are shown in t-direction.

To achieve a smoother motion Cabral and Leedom [4] suggested to employ a weighted filter kernel made up of two so-called Hanning filters,

$$k(s,t) \;=\; \frac{1+\cos(\kappa s)}{2} \times \frac{1+\cos(n\kappa s+\omega t)}{2}\,, \tag{19}$$

with $\kappa = 2\pi/2L$. For $n = 2$ the corresponding correlation is depicted in Fig. 11b. 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 which is restricted to box filters.

5.2 Frame Blending

The problems are caused by the splitting of the filter box. Therefore, we should try to work with unbroken boxes. Consider a convolution line containing point $\boldsymbol{x}_0$. Let us move the filter kernel N_a samples in both directions relative to $\boldsymbol{x}_0$, *without* reentering the interval. Then we obtain time dependent intensities

$$I(\boldsymbol{x}_0,t) = \frac{1}{2N_f+1} \sum_{i=-N_f-\lfloor tN_a \rfloor}^{N_f-\lfloor tN_a \rfloor} T(\boldsymbol{x}_i), \qquad t \in [-1,1]\,. \tag{20}$$

The negative sign in the index of $\boldsymbol{x}_{-\lfloor tN_a \rfloor}$ takes into account that a backward moving filter results in an apparent flow in forward direction. This image series will will exhibit a constant intensity correlation over time, but is non-periodic.

The idea to produce a periodic sequence is very simple: we divide the sequence in two halves and blend their frames. Since the motion of frames near the extreme positions cannot be continued, we weight frames less and less as their filter boxes approach the extreme positions:

$$I^*(\boldsymbol{x}_0,t) = w(t-1)\, I(\boldsymbol{x}_0,t-1) + w(t)\, I(\boldsymbol{x}_0,t), \qquad t \in [0,1]\,, \tag{21}$$

where $w(t)$ denotes the hat function $w(t)=1-|t|$ for $t \in [-1,1]$. This creates a periodic animation sequence with half the number of frames.

Due to the blending, contrast of the resulting images varies over time. This can be compensated by rescaling intensity I^* properly. In raw LIC images, intensity I of a single pixel is usually given

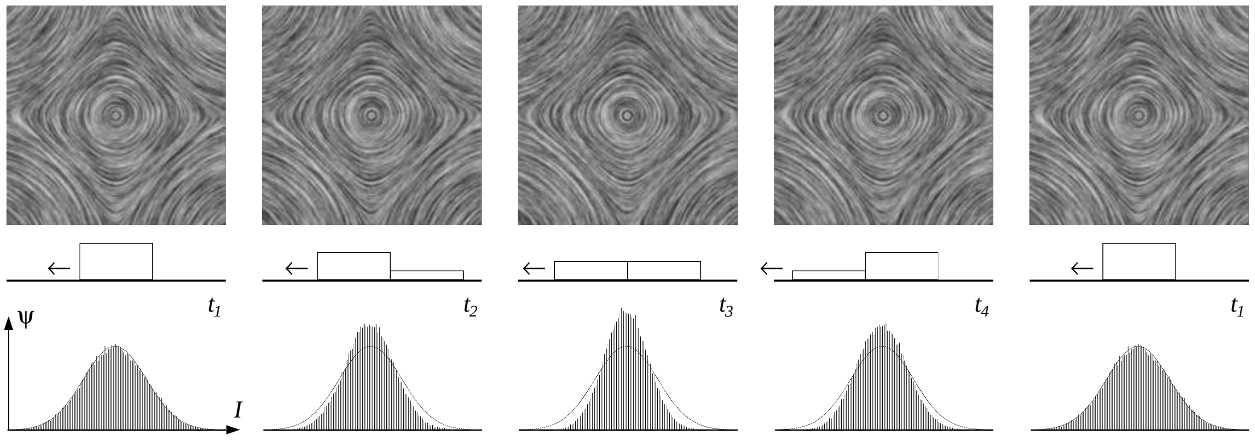


Figure 12: Snapshots from a periodic LIC animation obtained by frame blending. The first and the last image are identical. The figure contains a schematic view of the differently weighted filter boxes moving along the field line. In the lower part intensity histograms of the blended images are shown. To keep contrast constant, intensity has to be rescaled to fit the original gaussian distribution.

by convolving a large number of independent texture cells. Therefore the central limit theorem of statistics applies and intensities I can assumed to be gaussian distributed:

$$\psi(I) = \text{const.} \exp(-\frac{(I-\mu)^2}{2\sigma^2}). \tag{22}$$

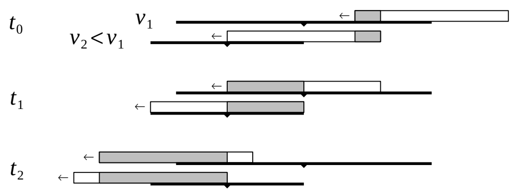
Here μ and σ^2 denote average and variance of the intensity distribution ψ, respectively. Any linear combination of independent gaussian distributed quantities will again be distributed gaussian. The resulting variance is given by $\sigma^2_{\text{res}} = \sum w_i^2 \sigma_i^2$. Consequently, after averaging 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

$$I \leftarrow \frac{I-\mu}{\sqrt{w_1^2 + w_2^2}} + \mu. \tag{23}$$

This equation is valid only if the intensities are statistically independent, i.e. if the filter boxes in the frames being averaged according to Eq. (21) do not overlap. This is guaranteed if the filter length does not exceed the length of the interval L_a, that is the distance needed for N_a samples. Alternatively, 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. The frame blending and intensity rescaling process is summarized in Fig. 12.

5.3 Space Dependent Velocity

To animate LIC images with variable velocity we use a straightforward extension of the technique for constant velocity. Again we generate a sequence of images by moving the filter boxes along field lines. But now each filter box moves some *variable* distance $L_a(\boldsymbol{x}_0)$, proportional to the local velocity. Moving synchronously, correlations between neighbouring pixels are maintained, even with strongly varying velocities. This is depicted in the following picture:



A periodic sequence is generated by the blending technique described above. Only concerning the contrast adjustment an additional problem arises: In general the intensities being averaged are not independent because filter boxes overlap in regions of low velocity, i.e. $L_a(\boldsymbol{x}) < L_f$. Therefore Eq. (23) is not valid anymore.

The above mentioned remedy of using two image sequences computed from different input textures also does not work. 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.

Therefore for each pixel we take into account the actual overlap of filter boxes at times t and $t-1$ and rescale intensity *locally*. 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:

$$\sigma_{\text{res}}^2 = \left((w_1^2 + w_2^2)(1-u)^2 + u^2\right)\sigma^2. \tag{24}$$

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.

5.4 Temporal Coherence

In Sect. 3.2.1 we showed, how one can take advantage of the pixel coherence along a field line. Considering this, performance gains of an order of magnitude can be achieved.

When animation sequences are generated, also temporal coherence of pixel intensities can be exploited [16]. From Eq. (20) it is obvious that $I(\boldsymbol{x}_0, t) = I(\boldsymbol{x}_{-tN_a}, 0)$. This means that pixel intensity at a given location in one frame is equal to pixel intensity at some other location in some other frame. Therefore, by keeping all frames in memory, repeated computation of equal field lines and convolution sums can be avoided. Evaluating Eq. (14) for samples $\boldsymbol{x}_{-N_s-N_a}$ to $\boldsymbol{x}_{N_s+N_a}$ provides all information necessary to obtain *all* time dependent intensities for samples $\boldsymbol{x}_{-N_s}$ to $\boldsymbol{x}_{N_s}$. Of course, for space dependent velocities N_a depends on $\boldsymbol{x}$. This can easily be handled by considering $\max\limits_{\boldsymbol{x}} N_a(\boldsymbol{x})$.

The obvious drawback of this method is, that a large amount of memory is needed to hold an accumulation buffer for each frame. Recall that for periodic sequences we have to blend and therefore need twice as many frames as displayed. This amount can be halved by performing the blending operation on the fly. After $I(\boldsymbol{x}_0, t-1)$ and $I(\boldsymbol{x}_0, t)$ have been computed, both intensities are immediately composed according to Eq. (21). The result is assigned to an accumulation buffer which corresponds to the final frame. With this technique a blending operation has to be performed each time a pixel is being hit. On the other hand, there is no more need for an additional pass over all accumulation buffers. Actually, we found that this more than compensates the costs for the additional blendings. Usually each pixel is hit not more than 5-6 times.

6 Parallelization

6.1 Parallelization of the Original LIC Algorithm

Two parallel versions of the original LIC algorithm have been described in [3]. Recall that this algorithm works for arbitrary filter kernels k and uses a fixed resolution defined by the vector field grid. Concerning parallelization this algorithm has the nice property of being strictly local.

The first method is a coarse grained implementation which uses a partitioning of the vector field (or image space) to assign work to multiple CPUs. It has been implemented on a shared memory multiprocessor (SGI Challenge). The second one is a fine-grained implementation which imitates LIC by a series of image warps and sums. These operations can be performed very fast on modern rendering architectures.

In the following two subsections we report work of Brian Cabral and Casey Leedom [3].

6.1.1 Coarse-Grained Parallelization

The LIC integral (3) is rewritten in discrete form:

$$I(\boldsymbol{x}_0) = \frac{\sum_{i=0}^{n} T(\boldsymbol{x}_i) h_i}{\sum_{i=0}^{n} h_i} \tag{25}$$

where $\boldsymbol{x}_i$ is the i^{th} vector field position along the parametric curve, h_i the analytical area under the portion of the kernel which spans the local field line segment containing $\boldsymbol{x}_i$, and $n+1$ is the number of vector field positions on the parametric curve $\boldsymbol{\sigma}$. This can be rewritten algorithmically as follows:

Algorithm: Original LIC

```
for each pixel $p$
    set $I(p) = 0$
    for each $bx_i$, defined over the local field line at $bx_0$ = center of $p$
        set $I(p) = I(p) + T(\boldsymbol{x}_i)h_i$
    end for
    set $I(p) = I(p)/\sum_{i=0}^{n} h_i$
end for
```

This algorithm is parallelized by decomposing the outer loop into coarse grains. Because of the local nature and lack of communication contention the only issues then are per-set overhead and processor cache locality. Experiments with different tiling decompositions, some of them designed to provide optimal cache locality, showed less than 10% performance variations. Therefore a simple row-tile decomposition was chosen. Per-set overhead is primarily determined by the operating system's cost for spawning and reaping threads. A 20 processor SGI Challenge with 150 MHz R4400 CPUs needed about 0.17 seconds to spawn and reap 20 threads that performed all the setup for determining the appropriate row-stripe to compute and then exited. In Fig. 13 performance results are shown in log-log form. The per-set overhead are most apparent for the smallest grid size (256^2) where the efficiency for 20 processors is 0.54. However, for grid size 4096^2 near-linear speedup can be achieved.

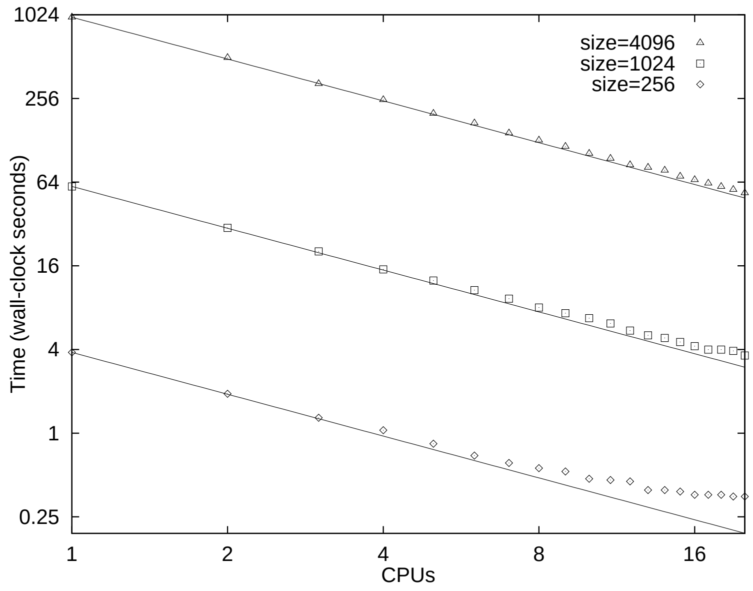


Figure 13: Performance results for vector fields with sizes 256^2, 1024^2, and 4096^2 using the coarse-grained parallel implementation on a SGI Challenge (150 MHz R4400 CPUs and 1 GB of 8-way interleaved memory). The lines show maximum theoretical linear speedup. Image courtesy of Brian Cabral and Casey Leedom.

6.1.2 Fine Grain Parallelization

The aim in this parallelization is to exploit the vast amount of specialized computing power in high-performance image rendering machines. The first idea is to reverse the order of the loops. Stepping along field lines, which is necessary to calculate the LIC sum, is performed simultaneously for all pixels. At each step one term of the LIC sum is added to the intensity values of all pixels. The second idea is to use texture mapping capabilities of the graphics hardware to imitate the stepping along field lines. This is done by image warping, which can be performed by texture mapping a trianglular mesh with the LIC input image. The mesh represents sample points in the vector field. The more the vector field varies just linearly over the mesh triangles, or the finer the warping mesh is, the more accurate is LIC approximated.

This LIC algorithm renders all the pixels each outer loop iteration, advecting a mesh as it proceeds. The time to advect a mesh however is significant. The main parallelization issue therefore becomes how fine a warping mesh is needed to accurately perform LIC.
The fine grained algorithm corresponding to equation is:

Algorithm: Original LIC, fine-grained parallelization

```
set $I \equiv 0$
for each $i$ from 0 to $2\sqrt{2}\,L$
    advect mesh for all field lines
    for each position $p$
        set $I(p) = I(p) + T(\boldsymbol{x}_i(p))h_i$
    end for
    set $I(p) = I(p)/(2\sqrt{2}\,L)$
end for
```

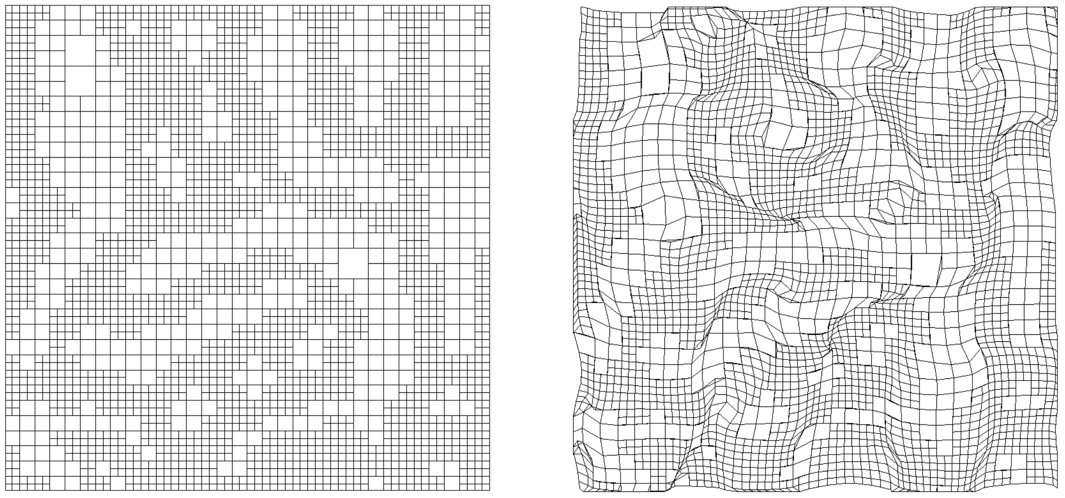


Figure 14: The geometric mesh (left) and the texture coordinate mesh (after 10 advection steps, right) used for the turbulent vector field imaged in Fig. 15. Images courtesy of Brian Cabral and Casey Leedom.

For each vertex a geometric coordinate, representing the screen position, and a texture coordinate is maintained. The texture coordinates, used by the graphics hardware to point into the input texture, are advected each step of the outer loop. This can be thought of as warping the input image onto the fixed geometric mesh. Both the geometric and texture coordinate meshes are illustrated ing Fig. 14.

The mesh advection is done as the streamline advection in [4]. While advected, each vertex moves a different distance. In order to advect each mesh vertex a distance L, the outer loop must iterate enough times. Setting the iteration count to $\sqrt{2}L$ produces results which are very similar to the original LIC algorithm, see Fig. 15. Differences occur in regions where the vector field is highly curved.

The mesh is created during the initialization phase by recursive quad-subdivision, according to the local vector field curvature. Mesh quads are subdivided if the integral of their curvature is greater than a given threshold and larger than a fixed minimal size (to prevent overmeshing near singularities).

The abilities of the graphics hardware both to render textured triangle meshes [1] and to sum pixels using the hardware buffer [6] at extremely high speed are exploited. The CPU time needed consists for three parts, the constant time to compute the curvature integration table, the CPU time to advect the mesh and the time to actually render the mesh. The latter one is always negligible. In Fig. 16 the wall-clock time versus mesh size is shown for a 1024^2 grid size on a SGI (150 MHz R4400 CPU and 4 raster manager RealityEngine).

The flat part of the 'rendering' curve indicates that the CPU time to advect the mesh prevails. The fine grained algorithm compares favorably to the coarse grained on 15+ processors.

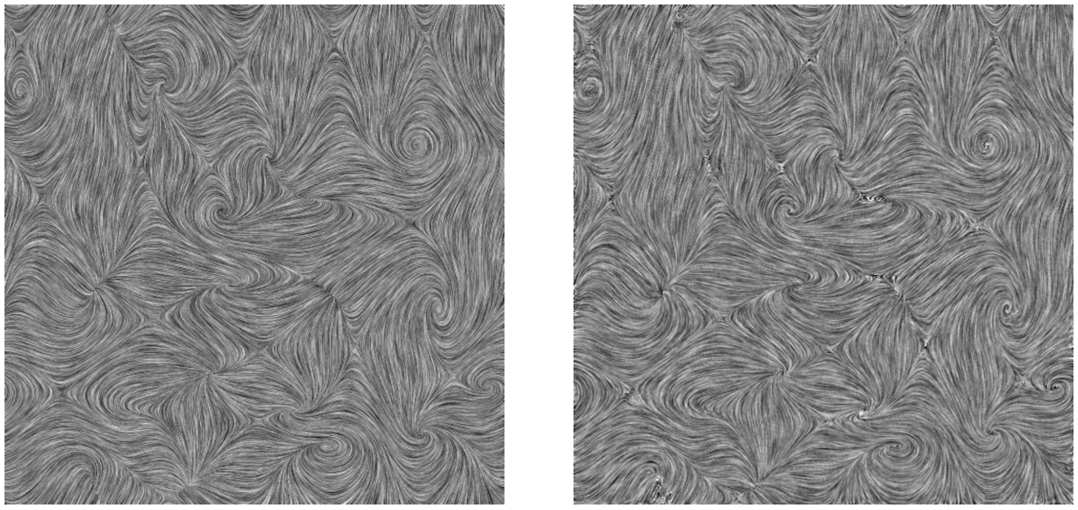


Figure 15: A synthetic, band-limited turbulence field, rendered using the coarse-grained algortihm (left) and the fine grain version (right). Images courtesy of Brian Cabral and Casey Leedom.

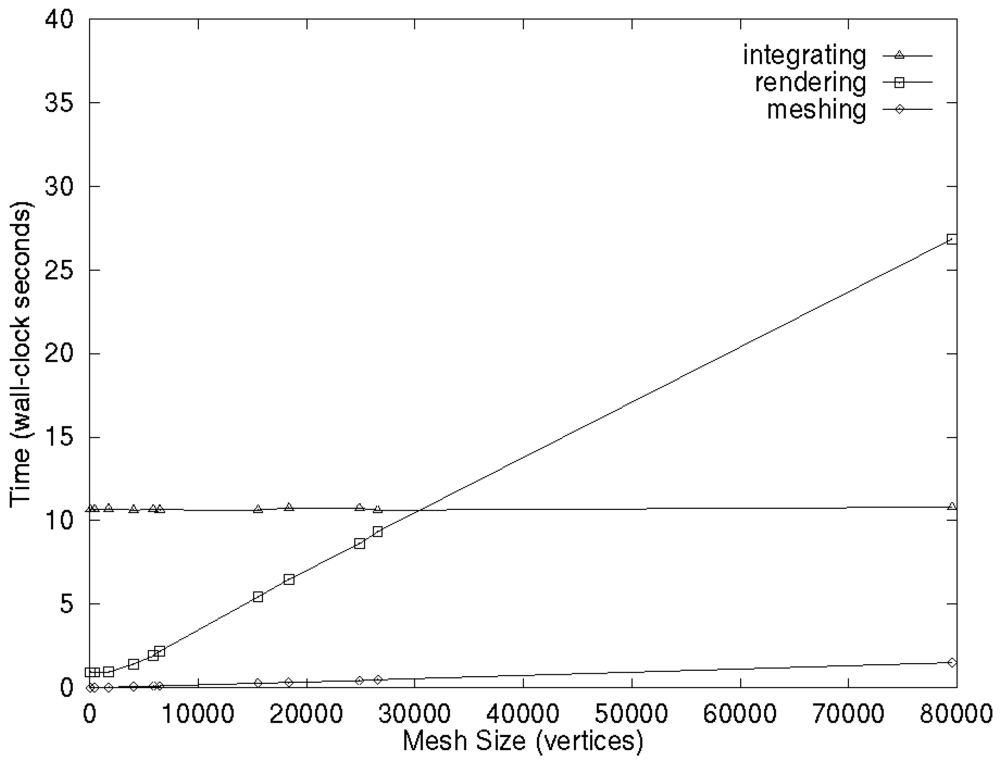


Figure 16: Performance of the fine-grained parallel LIC algorithm as a function of the number of mesh vertices.

6.2 Parallelization of the fastLIC Algorithm

6.2.1 Parallelization in Image Space

Truly massively parallel computer architectures contain physically distributed memory. A scalable implementation of LIC on such architectures is of interest for data visualization. It enables one to *interactively* explore *huge* data sets.

Designing such a parallel algorithm, the major decision to take is whether one bases it on a strictly local algorithm, like the original one by Brian Cabral and Casey Leedom [4], or on a optimized one like that in Sect. 3, that shows somewhat less locality properties. Since the latter ones need at least an order of magnitude less computational effort, we parallelize this. In the following we sketch a parallel version of the algorithm described in Sect. 3 (including zooming and animation capabilites). For a more detailed exposition see reference [16].

Since coherencies in time can be exploited using the technique described in Sect. 5.4 we distribute computation in image space. First we have to consider how to layout accumulation buffer and hit count buffer. One could use shared global buffers which are accessed by all processing elements (PEs), or alternatively, partitioned buffers which are accessed by individual PEs only.

Global buffers would require synchronization for every sample, and on distributed memory machines also communication. Even on shared memory systems such a fine granularity would be prohibitively slow, since locking mechanism would have to prevent that PEs update the same memory location at the same time. Obviously this is an even less suitable approach for distributed memory machines. Therefore we use distributed buffers.

In order to exploit temporal coherence we partition image space. The image is subdivided into several subdomains which are assigned to different processors. Within each partition also spatial coherence can be exploited. Each processor is responsible for computing an animated LIC sequence in a particular subdomain.

However, the algorithm is not strictly local. For calculating pixel values near the boundary of each subregion, the PEs need to access vector field data from neighboured subregions. Since the vector data are static they could simply be replicated, and each processor could then do its job without communication. However, if field line calculations were terminated at the boundaries of the subdomains, much of the potential benefits of the fast LIC algorithm would be lost. Non-neglectable startup costs make computation of few and long field lines significantly cheaper than computation of many but short ones.

Therefore we would like to choose the subdomains as large as possible. However, the more PEs are used, the smaller the subdomains are. Hence, with growing number of PEs the exploitation of spatial coherence decreases. In the limit of many PEs performance would converge against that of the original, strictly local LIC algorithm [4].

6.2.2 Update Scheme

The question arises how spatial and temporal coherence can be exploited *simultaneously.* The key observation is, that the update of accumulation and hit count buffers can be relaxed, since there is no need to represent the global state accurately at all times. If each PE has an additional buffer to store the values calculated in some *overlap region* outside its subdomain, these values can be combined with the results of other PEs within certain time intervalls. Using possibly outdated hit counts instead of globally correct ones, eventually superfluous field lines are calculated. The

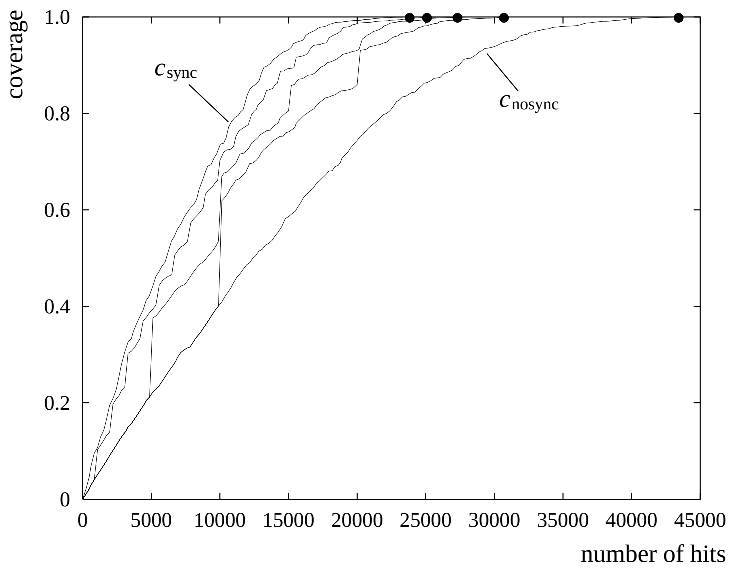


Figure 17: Experimentally determined coverage curves for different numbers of hit count synchronization. Data is taken every 200 hits. The steps correspond to buffer updates.

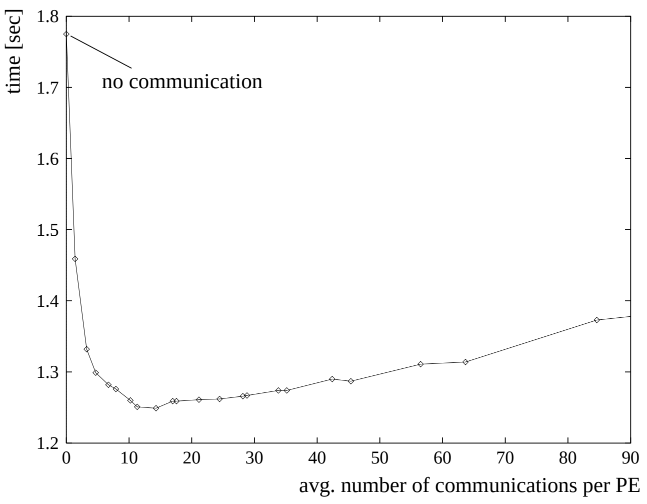
resulting image however remains correct.

The more frequently communication happens, the more exactly is the global state apprehended, and the less unnecessary computations are performed. The limiting case corresponds to the use of a single shared accumulation buffer, where spatial coherence is exploited as efficiently as in a sequential algorithm, but at the price of very high communication costs.

In order to analyze how much work can be saved by updating local hit count buffers only at certain intervals, we look at the pixel coverage c in the subdomain of a processor, i.e. the fraction of pixels that have been hit at least once. Not every hit leads to an increase in coverage because a pixel might already have been touched before. An analytical model for the dependence of coverage on the number of hits is described in [16]. In Fig. 17 some experimental data is shown. The curve c_{sync}, representing the synchronized case, approaches the limit of 100% coverage much faster than the line c_{nosync}, corresponding to never synchronized buffers. The intermediate curves depict pixel coverages where buffers are updated at discrete times. The jumps in these curves correspond to updates. It turns out that the limiting curve c_{sync} is approximated quite closely with comparetively few buffer updates.

Fig. 18 shows the time for computing an animation sequence versus the average number of communications. Obviously an optimal number of communications exists. Above a certain value the number of communications is not a very sensitive parameter.

For updating the buffers quickly, we introduce a bit matrix to mark each pixel that has been hit. While the hit count buffer contains only local information about a subregion, the bit matrix indicates if a pixel has been hit by *any* PE. Of course, this information is contained also in the hit count buffer. Each PE can receive bit matrices from its neighbours at any time and combine them with its own one using fast logical operations. The amount of data transferred per communication step is 1/32 compated to the transfer of full 4 byte hit count buffer. The exact number of hits per pixel is evaluated only once at the end of the LIC algorithm.
The basic parallel algorithm can be described by the following pseudo code (for minNUMHits= 1):



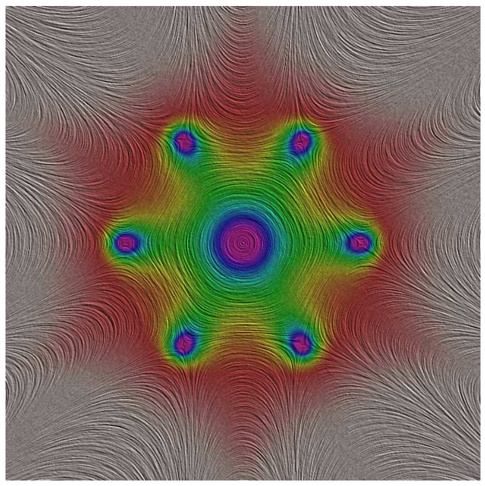


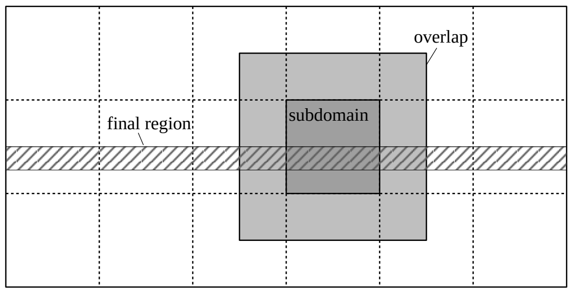
Figure 18: *Left:* Time for computing an animation sequence versus the average number of non-local buffer updates (20 frames, 800 × 800 pixels each, 32 PEs of a Cray T3D. *Right:* Relief filtered LIC image of the vector field used in this caclulation.

Algorithm: parallel fastLIC

```
set $nFieldlines = 0$
for each pixel $p$ (chosen according to some strategy) do
    if ($\mathsf{numHits}(p) == 0$) then
        initiate field line computation with $\boldsymbol{x}_0$ = center of $p$
        compute convolution $I(\boldsymbol{x}_0)$
        add result to pixel $p$
        set $i = 1$
        while $i < N_S$ do
            update convolution to obtain $I(\boldsymbol{x}_i)$ and $I(\boldsymbol{x}_{-i})$
            add results to pixels containing $\boldsymbol{x}_i$ and $\boldsymbol{x}_{-i}$
            set $i = i + 1$
        end while
        set $nFieldlines = nFieldlines + 1$
        if ($nFieldlines$ mod $comInterval == 0$) then
            for each neighbour PE $pe$ do
                combine bit matrix from $pe$ with own bit matrix
            end for
        end if
    end if
end for
wait for other PEs .
```

6.2.3 Data Collection

After the convolution integral has been computed for each pixel, some additional operations are necessary to generate the final images. First the accumulation buffers and hit counts in the overlap regions have to be combined. Then the accumulation buffers have to be divided by the hit counts and finally a contrast adjustment has to be performed. These steps are also executed in a distributed way. But we choose a different tiling for that. The image is divided into n equal sized horizontal stripes, where n is the number of PEs, cf. Fig. 19 (left). This allows fast output in scanline



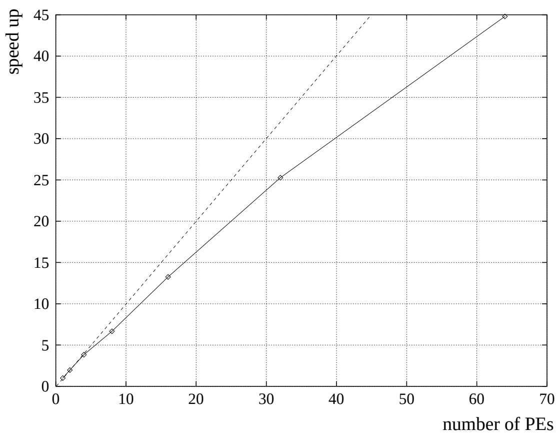


Figure 19: *Left:* Domain decomposition during LIC phase (rectangular blocks) and data collection phase (stripes). *Right:* Scalability of the parallel algorithm on a Cray T3D (pure computation time without I/O)

order without any further sorting steps. Each PE now retrieves those parts of other hit count and accumulation buffers that overlap with its stripe. Note that this can be done with a single shared memory get command per buffer, if these are organized in scanline order. This was the reason to redistribute the data after convolution.

6.2.4 Results

We implemented the algorithm on a Cray T3D. In Fig. 19 (right) the scalability based on measurements of computation time is depicted. Even with a large number of PEs satisfactory speed-ups are achieved. The deviations from the ideal linear behaviour are due to a number of reasons. On the one hand local differences in the vector field become more important as the subdomains get smaller. This increases load imbalances. On the other hand more and more hits are generated in the overlap region of each PE, which inevitably increases the amount of redundant computations. In addition start-up costs become more important.

While on a workstation (SGI Indigo^2 150 Mhz) it takes about 20 seconds to compute an animation sequence consisting of 20 frames 512×512 pixels each, we are able to create such a sequence in less than a second on a 64 PE partition of a Cray T3D.

We used this algorithm to implement a distributed environment for interactive exploration of vector fields. While the animated field is displayed on a local workstation, the user may enlarge interesting regions to examine details of the vector field. As the new image sequences are computed on the remote parallel machine, the user gets nearly immediate response. It is also possible to interact with the simulation (like changing boundary conditions) and to view the results without a significant latency. Snapshots of a CFD application in this environment are shown in Fig. 20.

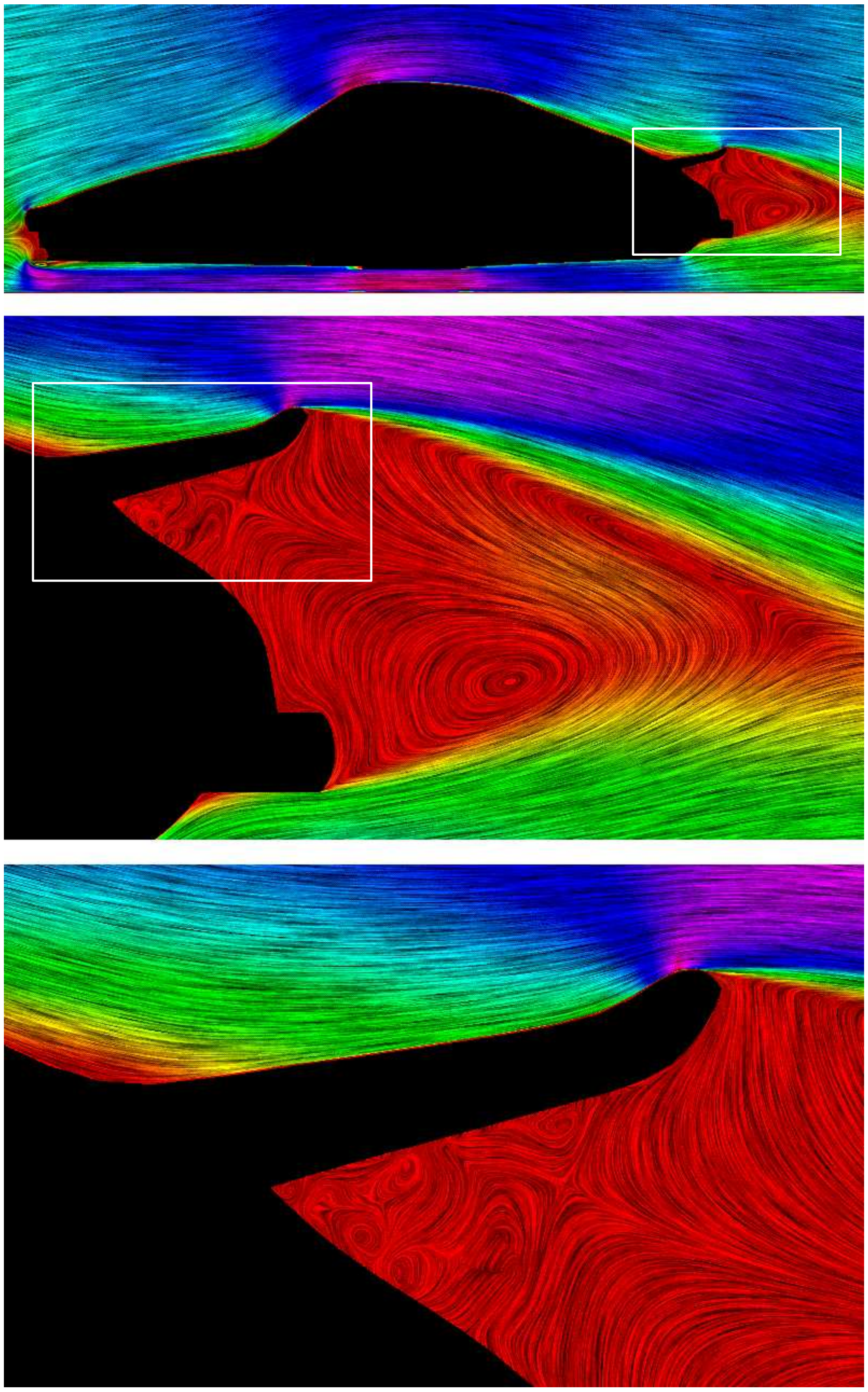


Figure 20: Snapshots from an interactive visualization application.

7 Conclusions and Outlook

Line integral convolution found many application areas, ranging from computer graphics and computer art to scientific visualization – both due to the aesthetically pleasing pictures and its capability to encode directional information in high resolution.

We have discussed algorithmic techniques that allow us to generate a LIC image within few seconds, even on a customary personal computer. A feature of this method is the mutual resolution independence of vector field, input texture and output image. This allows to produce smooth zooms in order to display details of the vector field. Furthermore, we presented methods for producing high quality texture animation sequences, both with constant and with location dependent velocity. Additionally algorithms and implementations for various parallel computer architectures have been described.

Lot of interesting work remains to do. A detailed analysis concerning the many degrees of freedom (properties of the input texture, type of filter kernel, length of filter kernel) is still missing. Another domain for future work is the generation of suitable artificial vector fields – comprising procedural methods and interactive techniques. The vector fields can depend on all types of image characteristics, e.g. visible boundaries at different length scales, or local directional features. An interesting question is, which kinds of of vector fields are suitable to imprint appealing directional textures into given images.

Another idea is to use LIC for the interactive generation of brush like effects, e.g. along the following lines: Moving a pencil on a given image defines locally a vector field. On basis of a specified noisy input texture then a LIC texture is calculated and possibly enhanced by a gradient filter. The resulting grey tone relief is impressed on the original image by multiplying its color values with the grey values of the LIC image.

All parameters including the type of filter kernel may vary in space and time. For instance it could be worthwhile to combine the generation of artificial vector fields with that of a scalar field that specifies a locally varying filter length. Interesting effects can probably also produced by varying the local velocities in time.

For scientific visualization the most interesting questions are how time dependent vector fields can be depicted using LIC images and how LIC ideas can be extended to visualize line like structures or flows in 3D.

In our eyes the LIC game has just started. Participate on this game!

Acknowledgements

Many thanks to Malte Zöckler who helped us during the development of optimized and parallelized LIC algorithms, and to Johannes Schmidt-Ehrenberg for programming assistance. We would also like to thank Charlie Gunn, Victoria Interrante, Christopher Johnson, Kwan-Liu Ma, Roland Wunderling, and Gerhard Zumbusch for various discussions about LIC techniques and algorithms. We are also grateful to Brian Cabral and Casey Leedom for making their code available on the net, which was our starting point in the LIC business.

References

[1] Akeley, K. RealityEngine graphics. In *Computer Graphics (SIGGRAPH '93 Proceedings)* (Aug. 1993), J. T. Kajiya, Ed., vol. 27, pp. 109–116.

[2] Bratley, P., and Fox, B. Algorithm 659: Implementing sobol's quasirandom sequence generator. *ACM Transactions on Mathematical Software 14* (1988), 88–100.

[3] Cabral, B., and Leedom, C. Highly parallel vector visualization using line integral convolution. In *Seventh SIAM Conference on Parallel Processing for Scientific Computing* (Feb. 1995), pp. 802–807.

[4] Cabral, B., and Leedom, L. C. Imaging vector fields using line integral convolution. In *Computer Graphics (SIGGRAPH '93 Proceedings)* (Aug. 1993), J. T. Kajiya, Ed., vol. 27, pp. 263–272.

[5] Deuflhard, P., and Bornemann, F. *Numerische Mathematik II: Integration gewöhnlicher Differentialgleichungen.* Verlag de Gruyter, Berlin, 1994.

[6] Haeberli, P. E., and Akeley, K. The accumulation buffer: Hardware support for high-quality rendering. In *Computer Graphics (SIGGRAPH '90 Proceedings)* (Aug. 1990), F. Baskett, Ed., vol. 24, pp. 309–318.

[7] Hairer, E., Nørsett, S. P., and Wanner, G. *Solving Ordinary Differential Equations I, Nonstiff Problems.* Springer Verlag, Berlin, Heidelberg, New York, Tokyo, 1987.

[8] Helman, J., and Hesselink, L. Representation and display of vector field topology in fluid flow data sets. *Visualization in scientific computing* (1990), 61–73.

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

[10] Kutta, W. Beitrag zur näherungsweisen Integration totaler Differentialgleichungen. *Zeitschr. für Math. u. Phys. 46* (1901), 435–453.

[11] Mallinson, G. D. The calculation of the lines of a three-dimensional vector field. In *Computational Fluid Dynamics* (Aug. 1988), G. de Vahl Davis and C. Fletcher, Eds., North-Holland, pp. 525–534.

[12] Press, W. H., Teukolsky, S. A., Vetterling, W. T., and Flannery, B. P. *Numerical Recipes in C: The Art of Scientific Computing*, 2nd ed. Cambridge University Press, Cambridge, 1992.

[13] Sobol, I. On the distribution of points in a cube and the approximate evaluation of integrals. *USSR Computational Mathematics and Mathematical Physics 7*, 4 (1967), 86–112.

[14] Stalling, D., and Hege, H. Fast and resolution independent line integral convolution. In *SIGGRAPH 95 Conference Proceedings* (Aug. 1995), R. Cook, Ed., Annual Conference Series, ACM SIGGRAPH, Addison Wesley, pp. 249–256. held in Los Angeles, California, 06-11 August 1995.

[15] van Wijk, J. J. Rendering surface-particles. In *Visualization '92* (Oct. 1992), IEEE Computer Society, pp. 54–61.

[16] Zoeckler, M., Stalling, D., and Hege, H. Parallel line integral convolution. In *Proceedings First Eurographics Workshop on Parallel Graphics and Visualisation* (Sept. 1996), Annual Conference Series, Eurographics, pp. 249–256. held in Bristol, UK, 26-27 September 1996.