A New Line Integral Convolution Algorithm for Visualizing Time-Varying Flow Fields

Han-Wei Shen and David L. Kao

Abstract—New challenges on vector field visualization emerge as time-dependent numerical simulations become ubiquitous in the field of computational fluid dynamics (CFD). To visualize data generated from these simulations, traditional techniques, such as displaying particle traces, can only reveal flow phenomena in preselected local regions and, thus, are unable to track the evolution of global flow features over time. This paper presents a new algorithm, called UFLIC (Unsteady Flow LIC), to visualize vector data in unsteady flow fields. Our algorithm extends a texture synthesis technique, called Line Integral Convolution (LIC), by devising a new convolution algorithm that uses a time-accurate value scattering scheme to model the texture advection. In addition, our algorithm maintains the coherence of the flow animation by successively updating the convolution results over time. Furthermore, we propose a parallel UFLIC algorithm that can achieve high load-balancing for multiprocessor computers with shared memory architecture. We demonstrate the effectiveness of our new algorithm by presenting image snapshots from several CFD case studies.

Index Terms—Flow visualization, vector field visualization, image convolution, line integral convolution, flow animation, unsteady flows, texture synthesis, parallel algorithm.

1 Introduction

Vector field data arise from computer simulations in a variety of disciplines, such as computational fluid dynamics (CFD), global climate modeling, and electromagnetism. Visualizing these vector data effectively is a challenging problem due to the difficulties in finding suitable graphical icons to represent and display vectors on two-dimensional computer displays. At present, new challenges emerge as time-dependent simulations become ubiquitous. These simulations produce large-scale solutions of multiple time steps, which carry complex dynamic information about the underlying simulation model. To visualize these time-varying data, two types of methods are generally used. One can be referred to as the *instantaneous method*, where visualizations are created based on an instance of the data field in time. The visualization results, such as streamlines and vector plots, from a sequence of discrete time steps are then animated together. The instantaneous method often suffers from the problem of lacking coherence between animation frames. This incoherent animation will interfere with the understanding of a flow field's unsteady phenomena. In addition, the streamline shown at any given instance in time does not correspond to the path that a particle will travel because the flow is changing its direction constantly. To amend these problems, researchers have developed a different method, called the *time-dependent method*, which can better characterize the evolution of the flow field by continuously tracking visualization objects, such as particles over time. Examples are numerical streaklines and pathlines [1], [2].

This paper presents a time-dependent method for visualizing vector data in unsteady flow fields. Using the *Line Integral Convolution* (LIC) [3] as the underlying approach, we propose a new convolution algorithm, called *UFLIC* (Unsteady Flow LIC), to accurately model the unsteady flow advection. The LIC method, originally proposed by Cabral and Leedom [3], is a visualization technique that can produce continuous flow textures which resemble the surface oil patterns produced in wind-tunnel experiments. The synthesized textures can effectively illustrate global flow directions of a very dense flow field. However, because LIC convolution is computed following the traces of streamlines in a steady flow field, it cannot be readily used for visualizing unsteady flow data. A simple extension is to compute the LIC at every time step of the flow field and then animate the results together. Unfortunately, this approach suffers the same problems as the instantaneous method that we mentioned above. Forssell and Cohen [4] proposed an extension by changing the convolution path from streamlines to pathlines for visualizing time-varying vector fields. While their method produces a better visualization of unsteady flow fields, the resulting animation can lack coherence between consecutive frames when the underlying flows are fairly unsteady.

Our algorithm extends the LIC method by devising a new convolution algorithm that simulates the advection of flow traces globally in unsteady flow fields. As the regular LIC, our algorithm takes a white noise image as the input texture. This input texture is then advected over time to create directional patterns of the flow at every time step. The advection is performed by using a new convolution method, called *time-accurate value scattering* scheme. In the time-accurate value scattering scheme, the image value at every pixel is scattered following the flow's pathline trace,

- *H.-W. Shen is with MRJ Technology Solutions at NASA Ames Research Center, Mail Stop T27A-2, Moffett Field, CA 94035. E-mail: hwshen@nas.nasa.gov.*
- *D.L. Kao is with NASA Ames Research Center, Mail Stop T27A-2, Moffett Field, CA 94035.*

For information on obtaining reprints of this article, please send e-mail to: tvcg@computer.org, and reference IEEECS Log Number 106590.

which can be computed using numerical integration methods. At every integration step of the pathline, the image value from the source pixel is coupled with a timestamp corresponding to a physical time and then deposited to the pixel on the path. Once every pixel completes its scattering, the convolution value for every pixel is computed by collecting the deposits that have timestamps matching the time corresponding to the current animation frame. To track the flow patterns over time and to maintain the coherence between animation frames, we devise a process, called *successive feed-forward*, that drives the convolutions over time. In the process, we repeat the time-accurate value scattering at every time step. Instead of using the white noise image as the texture input every time, we take the resulting texture from the previous convolution step, perform high-pass filtering, and then use it as the texture input to compute the new convolution.

Based on our preliminary work in [5], this paper provides a precise description of the UFLIC algorithm and its important implementation details. In addition, to improve the algorithm's interactivity, we present a parallel implementation of our algorithm for multiprocessor machines with shared-memory architectures. In our parallel algorithm, we distribute the convolution workload among available processors by subdividing the texture space into subregions. By carefully choosing the shapes of subregions, we can achieve high load balancing.

In the following, we first give an overview of the LIC method. Next, we describe and analyze an existing algorithm for unsteady flows. We then present our UFLIC algorithm in detail, followed by our parallel algorithm. We conclude this paper by presenting performance results and case studies by applying our method to several unsteady flow data sets from CFD simulations.

2 Background and Related Work

In this section, we briefly review the LIC method proposed by Cabral and Leedom [3]. We then describe and analyze the method proposed by Forssell and Cohen [4] for unsteady flow field data.

2.1 Line Integral Convolution

The LIC method is a texture synthesis technique that can be used to visualize vector field data. Taking a vector field and a white noise image as the input, the algorithm uses a low-pass filter to perform one-dimensional convolution on the noise image. The convolution kernel follows the paths of streamlines originating from each pixel in both positive and negative directions. The resulting intensity values of the LIC pixels along each streamline are strongly correlated so the directional patterns of the flow field can be easily visualized. To perform the convolution, different periodic filter kernels can be used. Examples are the Hanning filter [3] and the box filter [6], [7]. Fig. 1 illustrates the process of LIC convolution.

Recently, several extensions to the original LIC algorithm have been proposed. Forssell and Cohen [4] adapt the LIC method for curvilinear grid data. Stalling and Hege [6] propose an efficient convolution method to speed up the LIC computation. Shen et al. [7] combine dye advection with three-dimensional LIC to visualize global and local flow features at the same time. Okada and Kao [8] use post-filtering techniques to sharpen the LIC output and highlight flow features, such as flow separations and reattachments. Kiu and Banks [9] propose using multifrequency noise input for LIC to enhance the contrasts among regions with different velocity magnitudes. Recently, Jobard and Lefer [10] devised a Motion Map data structure that can encode the motion information of a flow field and produce a visual effect that is very similar to the LIC image.

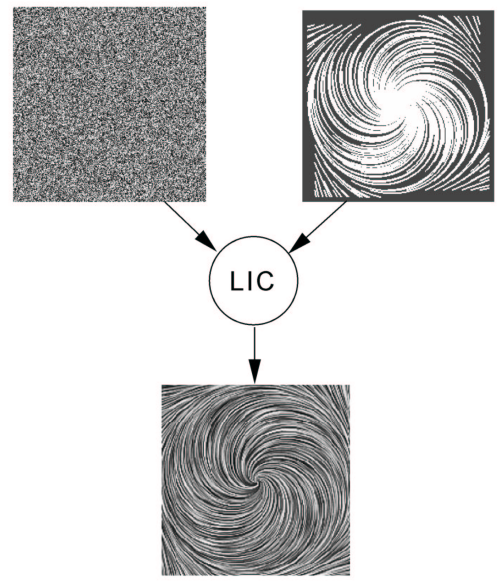


Fig. 1. The process of LIC convolution.

The texture outputs from the LIC method provide an excellent visual representation of the flow field. This effectiveness generally comes from two types of coherence. The first is *spatial coherence*, which is used to highlight the flow lines of the field in the output image. The LIC method establishes this coherence by correlating the pixel values along a streamline as the result of the line integral convolution. The second type of coherence is *temporal coherence*. This coherence is required for animating the flow motion. The LIC method achieves this temporal coherence by shifting the filter phase used in the convolution so that the convolved noise texture can periodically move along the streamlines in time.

2.2 Line Integral Convolution for Unsteady Flows

The LIC technique proposed originally is primarily intended for visualizing data in steady flow fields. To visualize unsteady flow data, an extension was proposed by Forssell and Cohen [4]. In contrast with convolving streamlines in the steady flow field, the extension convolves forward and backward pathlines originating from each pixel at every time step. A pathline is the path that a particle travels through the unsteady flow field in time. A mathematical definition of a pathline is given in the next section. To animate the flow motion, the algorithm shifts the filter's phase at every time step to generate the animation sequence.

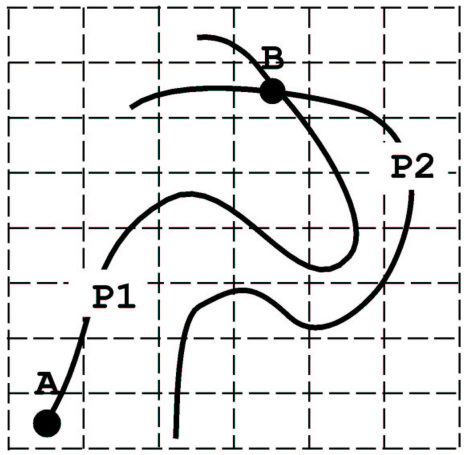


Fig. 2. The convolution values of pixels A and B are uncorrelated because pixels A and B have different convolution paths, represented by P1 and P2, respectively.

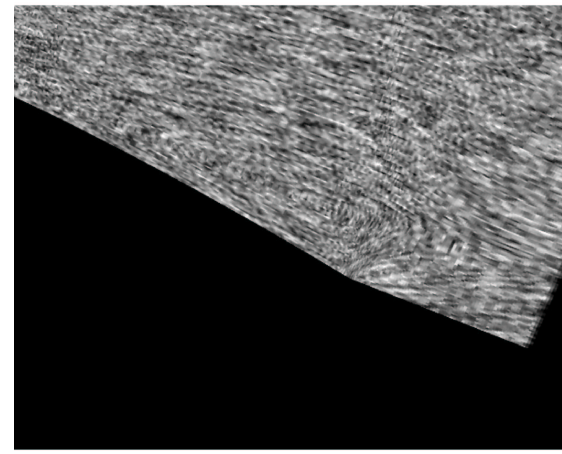


Fig. 3. A convolution image generated using the pathline convolution algorithm.

The purpose of convolving along pathlines is to show the traces of particles moving in unsteady flows, thus revealing the dynamic features of the underlying field. However, there are several problems associated with the pathline convolution when the flow is rather unsteady. First, the coherence of the convolution values along a pathline is difficult to establish. We illustrate this problem in Fig. 2. A pathline P_1 that starts from pixel A at time T_1 and passes through pixel B at time T_2 is the convolution path for pixel A. Similarly, pathline P_2 starting from B at time T_1 is the convolution path for pixel B. Since pathlines P_1 and P_2 pass through B at different times (T_2 and T_1, respectively), they have different traces. As a result, the convolution values of A and B are uncorrelated because two different sets of pixel values are used. Hence, image value coherence along neither pathline P_1 or P_2 is established. Forssell and Cohen in [4] reported that flow lines in the output images become ambiguous when the convolution window is set too wide. The problem mentioned here can explain this phenomenon.

The other drawback of using pathline convolution comes from the difficulties of establishing temporal coherence by using the phase-shift method as proposed in the regular LIC method. The reason lies in the variation, over time, of the pathlines originating from the same seed point when the flow is unsteady. As a result, in the algorithm, the same filter with shifted phases is applied to different convolution paths. However, the effectiveness of using the phase-shift method to create artificial motion effects mainly relies on the fact that the convolution path is fixed over time. As a result, the temporal coherence between consecutive frames to represent the flow motion using this phase-shift method becomes rather obscure for unsteady flow data.

In addition to the above problems, we have discussed some other issues of the pathline convolution in [5]. These problems in practice limit the effectiveness of using pathline convolution to visualize flow fields that are rather unsteady. Fig. 3 shows a sample image produced by the pathline convolution method. The obscure coherence in the flow texture is clearly noticeable.

In the following section, we propose a new algorithm, UFLIC, for visualizing unsteady flow fields. Instead of relying on shifting convolution kernel phase and gathering pixel values from a flow path for every pixel to create animated textures, we devise a new convolution algorithm to simulate the advection of textures based on the underlying flow fields. The new algorithm consists of a time-accurate value-scattering scheme and a successive feed-forward method. Our algorithm provides a time-accurate, highly coherent solution to highlight dynamic global features in unsteady flow fields.

3 New Algorithm

Image convolution can be thought of as gathering values from many input samples to an output sample (*value gathering*) or as scattering one input sample to many output samples (*value scattering*). The LIC method proposed originally by Cabral and Leedom [3] uses a value-gathering scheme. In this algorithm, each pixel in the field travels in both positive and negative streamline directions to gather pixel values to compute the convolution. This convolution can be implemented in a different way by first letting every pixel scatter its image intensity value along its streamline path; the convolution result of each pixel in the resulting image is then computed by averaging the contributions that were previously made by other pixels. We refer to this method as a *value-scattering scheme*. When the convolution path is a streamline in a steady-state vector field, value gathering and value scattering are equivalent. However, for time-varying vector fields, value gathering and value scattering can produce different results if a pathline is used as the convolution path. To illustrate this, again in Fig. 2, the pathline from pixel A to pixel B enables B to receive a contribution from A if the scattering scheme is used. However, when using the gathering scheme, B does not gather A's value, because the pathline $P2$ starting from B does not pass through A. To accurately reflect the physical phenomena in unsteady flow fields, the method of value scattering convolution is more appropriate than the value-gathering method. The reason lies in the nature of value-scattering which corresponds to flow advections, where every pixel in

the image plane can be thought of as a particle. The particles move along the flow traces and leave their footprints to create a convolution image.

In this section, we present a new algorithm, UFLIC, which uses a value-scattering scheme to perform line-integral convolution for visualizing unsteady flow fields. Starting from a white noise image as the input texture, our UFLIC algorithm successively advects the texture to create a sequence of flow images. To achieve this, we propose a new convolution method called the *time-accurate value scattering* scheme which incorporates time into the convolution. This time-accurate value scattering convolution scheme, driven by a *successive feed-forward* process, iteratively takes the convolution output from the previous step, after high-pass filtering, as the texture input for the next convolution to produce new flow textures. Our UFLIC algorithm can effectively produce animations with spatial and temporal coherence for tracking dynamic flow features over time. In the following, we first present the time-accurate value-scattering scheme. We then describe the successive feed-forward process.

3.1 Time-Accurate Value Scattering

In a time-varying simulation, two different notions of time are frequently used. One is the *physical time* and the other is the *computational time*. Physical time is used to describe a continuous measurable quantity in a physical world, such as seconds or days. Let the physical delta time between any two consecutive time steps in an unsteady flow be denoted by dt. Then, $t_{i+1} = t_i + dt$, where t_i is the physical time at the ith time step. Computational time is a nonphysical quantity in computational space. At the ith time step, let τ_i be the corresponding computational time, then $\tau_i = i$. The computational step size is one. In the following, both types of time are involved and carefully distinguished.

We propose a time-accurate value-scattering scheme to compute the line integral convolution. This value-scattering scheme computes a convolution image for each step of the time-varying flow data by advecting the input texture over time. The input texture can be either a white noise image or a convolution result output from the preceding step. We delay the discussion of the choice of an input until the next section. In the following discussion, we assume that the current computational time step is τ and its corresponding physical time is t, and we explain our value-scattering convolution method.

Given an input texture, every pixel in the field serves as a *seed particle*. From its pixel position at the starting physical time t, the seed particle advects forward in the field both in space and time following a pathline that can be defined as:

$$\mathbf{p}(t + \Delta t) = \mathbf{p}(t) + \int_t^{t+\Delta t} v(\mathbf{p}(t), t)dt,$$

where $\mathbf{p}(t)$ is the position of the particle at physical time t, $\mathbf{p}(t + \Delta t)$ is the new position after time Δt, and $v(\mathbf{p}(t), t)$ is the velocity of the particle at $\mathbf{p}(t)$ at physical time t. To evaluate the above expression and generate particle traces, numerical methods, such as the *Runge-Kutta* second- or fourth-order integration scheme, can be used.

At every integration step, the input image value, I_p, of the pixel from which the particle originates is normalized and scattered to the pixels along the pathline. The normalization is determined by two factors:

- the length of the current integration step.
- the "age" of the particle.

Assuming that a box kernel function $\kappa = 1$ is used, the line integral convolution can then be computed by multiplying the image value by the distance between two consecutive integration steps. This explains the first normalization factor. Assume that the particle is at its nth integration step; then, the distance between the particle positions of the current and preceding integration is defined as ω and can be expressed as:

$$\omega = \|\mathbf{p}(t + \Delta t) - \mathbf{p}(t)\|.$$

The second normalization factor simulates the effect of fading, over time, of the seed particle's intensity. To do this, we define a particle's "age" at its nth integration step as:

$$A_n = \sum_{i=1}^{n} \Delta t_i,$$

where Δt_i is the ith time increment in the pathline integration. Based on a particle's age, we define a normalization variable ψ, which has a value that decreases as the "age" of the particle increases. Assuming that the expected life span of a particle is T, then:

$$\psi = 1 - \frac{A_n}{T}.$$

With ω and ψ, the overall normalization weight W is:

$$W = \omega \times \psi.$$

Then, the normalized scattering value at the nth integration step becomes:

$$I_{normalized} = I_p \times W.$$

In our data scattering scheme, the normalized pixel value at every integration step is associated with a time-stamp. Given that the pathline starts from its seed pixel at physical time t, the corresponding physical time at the nth integration step is, then:

$$t' = t + \sum_{i=1}^{n} \Delta t_i.$$

We compute the convolution frame only at every integer computational time step corresponding to the original data. Therefore, we use a rounded-up computational time corresponding to its physical time as the timestamp associated with the normalized image value. This can be computed by:

$$\tau' = \left\lceil \tau + \sum_{i=1}^{n} \Delta\tau_i \right\rceil.$$

To receive the scattering image values, each pixel keeps a buffer, called the *Convolution Buffer (C-Buffer)*. Within the C-Buffer, there are several buckets corresponding to different computational times. Each bucket has a field of accumulated image values, I_{accum}, and a field of accumulated weights, W_{accum}. The scattering at the nth integration step is done by adding the normalized image value $I_{normalized}$ and its weight W to the bucket of the pixel, at the nth integration step, that corresponds to the computational time τ':

$$I_{accum} = I_{accum} + I_{normalized}$$

$$W_{accum} = W_{accum} + W.$$

For each seed particle, the distance that it can travel is defined as the convolution length. We determine this convolution length indirectly by specifying the particle's life span. We defined this life span in computational time, which can be converted to a physical time and used for every particle in the convolution. The advantage of using this global life span to control the convolution length is that the lengths of different pathlines are automatically scaled to be proportional to the particle's velocity magnitude, which is a desirable effect, as described in [3], [4]. In addition, this life span gives the number of time steps for which data must be loaded into main memory so that a particle may complete its advection.

Based on the life span specified for the seed particle advection, the number of buckets in a C-Buffer structure that is required can actually be predetermined. Assuming that the life span of a pathline is N in computational time, i.e., the particle starts at τ_i and ends at $\tau_i + N$, then only N buckets in the buffer are needed, because no particle will travel longer than N computational time steps after it is born. In our implementation, the C-Buffer is a one-dimensional ring buffer structure. The integer Φ is an index which points to the bucket that corresponds to the current computational time, τ, when every pixel starts the advection. The value of Φ can be computed as:

$$\Phi = \tau \bmod N.$$

Hence, assuming that the particle is at a computational time τ' at its current integration step, then the corresponding bucket in the ring buffer of the destination pixel that it should deposit to has index:

$$(\Phi + \tau' - \tau) \bmod N.$$

Fig. 4 depicts the structure of the ring buffer.

In the time-accurate value-scattering scheme, every pixel advects in the field and scatters its image value based on the scattering method just described. After every pixel completes the scattering process, we can start computing the convolution to get the resulting texture. We proceed by including in the convolution only those scattered pixel values that have a timestamp equal to the current computational time τ. Therefore, we go to each pixel's C-Buffer and obtain the accumulated pixel value I_{accum} and the accumulated weight W_{accum} from the corresponding bucket, which is the bucket that has the index Φ. The final convolution value C is then computed:

$$C = \frac{I_{accum}}{W_{accum}}.$$

The accumulated image values in other buckets with future timestamps will be used when the time comes. We increment the value Φ by one so the current bucket in the C-Buffer can be reused.

$$\Phi = (\Phi + 1) \bmod N.$$

In addition, the current computational time τ is also incremented by one, and the value-scattering convolution will proceed to the next time step of data.

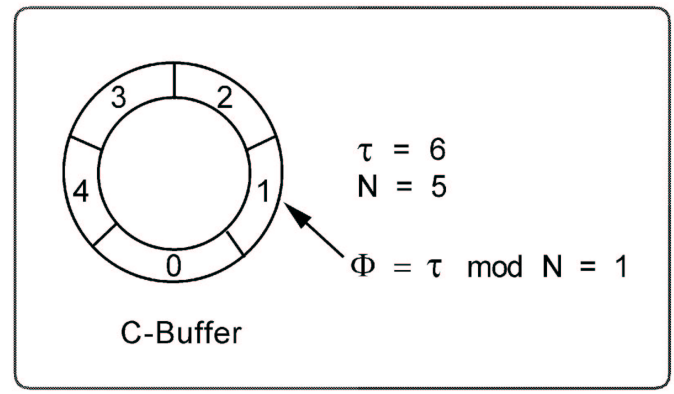


Fig. 4. In this example, the life span of a seed particle (N) is five computational time steps, and the current computational time (τ) at which a pixel starts the advection is six; then the value of Φ pointing to the bucket in the Convolution Buffer (C-Buffer) is one.

It is worth mentioning that, in our method, each pixel scatters its value along only the forward pathline direction but not the backward direction. The reason lies in nature: Backward scattering does not correspond to an observable physical phenomenon; flows do not advect backwards. In addition, the symmetry issue mentioned in [3] does not appear as a problem in our unsteady flow animations.

3.2 Successive Feed-Forward

Our value-scattering scheme provides a time-accurate model for simulating the flow advection to create spatial coherence in the output texture. In this section, we describe the process that successively transports the convolution results over time to maintain the temporal coherence for visualizing unsteady flows.

As mentioned previously, we define a time-dependent method as one that progressively tracks the visualization results over time. In this section, we present a time-dependent process, called *successive feed-forward*, which drives our time-accurate value-scattering scheme to create temporal coherence. Our algorithm works as follows: Initially, the input to our value-scattering convolution algorithm is a regular white noise texture. Our value scattering scheme advects and convolves the noise texture to obtain the convolution result at the first time step. For the subsequent convolutions, instead of using the noise texture again, we use the output from the previous convolution as the input texture. This input texture, showing patterns that have been formed by previous steps of the flow field, is then further advected. As a result, the output frames in consecutive time steps are highly coherent, because the flow texture is continuously convolved and advected throughout space and time.

There is an important issue with the successive feed-forward method that must be addressed. The line integral convolution method in general, or our value scattering scheme in particular, is really a low-pass filtering process. Consequently, contrasts among flow lines will gradually be reduced over time as the low-pass filtering process is repeatedly applied to the input texture. This would cause problems if one tried to visualize a long sequence of unsteady flow data. To correct this problem, we first apply a

high-pass filter (HPF) to the input texture, which is the result from the previous convolution, before it is used by the value-scattering scheme at the next step. This high-pass filter helps to enhance the flow lines and maintain the contrast in the input texture. The high-pass filter used in our method is a *Laplacian* operator. A two-dimensional Laplacian can be written as a 3×3 mask:

$$\begin{vmatrix} 1 & 1 & 1 \\ 1 & -8 & 1 \\ 1 & 1 & 1 \end{vmatrix}.$$

The result computed from the mask does not have exclusively positive values. In order to display the result, a common technique used in digital image processing applications is to subtract the Laplacian from the original image. The filter mask of overall operations of Laplacian and subtraction can be derived as:

$$HPF = \begin{vmatrix} -1 & -1 & -1 \\ -1 & 9 & -1 \\ -1 & -1 & -1 \end{vmatrix} = \begin{vmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{vmatrix} - \begin{vmatrix} 1 & 1 & 1 \\ 1 & -8 & 1 \\ 1 & 1 & 1 \end{vmatrix}.$$

To prevent the high-pass filter from introducing unnecessary high frequencies which might cause aliasing in our final image, we jitter the resulting output from the high-pass filter with the original input noise texture. This is done by masking the least significant seven bits of the output with the original input noise.

With the high-pass filtering and the noise jittering, we can produce convolution images with restored contrast and clearer flow traces. Fig. 5 shows a snapshot from an animation sequence without using the noise-jittered high-pass filter. Fig. 6 is the result with the noise-jittered high-pass filter. The difference is quite dramatic. Note that Fig. 6 can be used to compare with Fig. 3. Both images are produced from the same time step of data. Fig. 7 gives an overview of our entire algorithm. Note that the noise-jittered high-pass filtering process applies only to the input texture for the next iteration of convolution and not to the animation images.

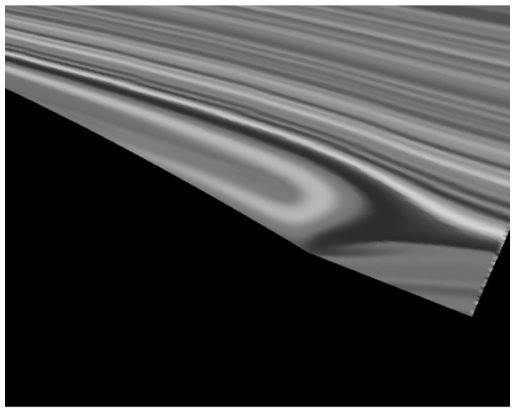


Fig. 5. A convolution image generated from an input texture without noise-jittered high-pass filtering.

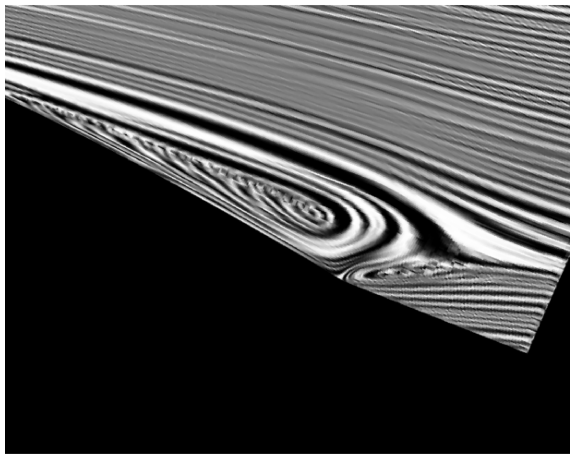


Fig. 6. A convolution image generated from an input texture with noise-jittered high-pass filtering.

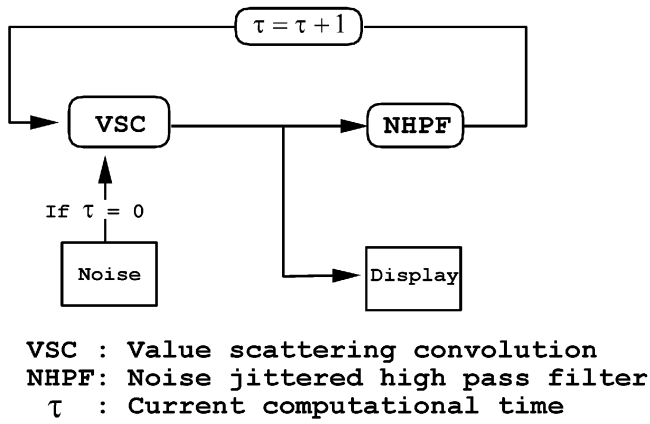


Fig. 7. Algorithm flowchart.

4 Parallel Implementation

In this section, we present a simple parallel implementation of the UFLIC algorithm for multiprocessor machines with shared-memory architectures. In the following, we first discuss how we subdivide the convolution workload among processors. We then describe the synchronization steps among the processors. It is noteworthy that a parallel algorithm for a regular LIC method on massively parallel distributed memory computers was proposed by Zöckler et al. [11]. Both their algorithm for steady LIC and our method for unsteady LIC subdivide the texture image space to distribute the workload.

4.1 Workload Subdivision

The nature of the successive feed-forward process requires that the UFLIC algorithm must be executed sequentially over time because the convolution in one time step has to be completed before the next convolution can start. In our implementation, we parallelize the time-accurate value-scattering scheme which is executed at every computational time step. To distribute the workload, we subdivide the image space into subregions and distribute the convolution task in those subregions among available processors. In choosing the shapes of the subregions, there are two considerations:

- Subregions that are assigned to each processor should be well distributed in the image space.
- For each processor, the locality of the data access should be maintained when computing the convolution.

During the convolution, the workload incurred from the value scattering of each pixel is generally determined by the length of the pathline. The farther the pathline extends, the more work is needed, because more pixels are encountered along the way and more operations of value accumulation are involved. This occurs when the seed particle travels through regions that have higher velocity. In a flow field, the variation of velocities in different regions of a flow field is quite dramatic, and the distribution of velocities is usually quite uneven. In order to give each processor a balanced workload, a processor should be assigned subregions that are evenly distributed over the entire field.

The other issue that has to be considered when subdividing the work space is the maintenance of the locality of data access to avoid the penalty caused by local cache miss or memory page fault when accessing the flow data. This can happen when two consecutive seed particles that a processor schedules to advect are located a long distance from each other. In this case, the vector data that was brought into the cache or main memory for one particle advection has to be flushed out for a new page of data when the advection of the second particle starts.

Based on the above two considerations, our parallel algorithm divides the image space into rectangular tiles, as in Fig. 8. Given P processors, we first specify the number of tiles, M, that each processor will receive. We then divide the entire texture space into $M \times P$ tiles. We randomly assign the tiles to each processor by associating each tile with a random number; then we sort the tiles by these random numbers. After the sorting, each processor takes its turn to grab M tiles from the sorted tile list.

4.2 Process Synchronization

Once the work distribution is completed, each processor starts computing the convolution values for the pixels in its tiles. The parallel convolution process can be divided into three phases which are the synchronization points among the processors. These three phases are:

- Scattering pixel values along pathlines.
- Computing the convolution results from the C-Buffers.
- Performing noise-jittered high-pass filtering.

In the first phase, the value scattering involves writing to the C-Buffers that belong to the pixels along a pathline. Since all the processors are performing the scattering simultaneously, it is possible that more than one processor needs to write to the same bucket of a C-Buffer at the same time. In our implementation, instead of locking the C-buffer when a processor is writing to it, which will inevitably incur performance overhead, we allocate a separate set of buckets for each processor. Recall that, in the sequential algorithm, the number of buckets in a C-buffer is equal to the pathline's life span N. Given P available processors, we allocate N buckets for each processor, so the total number of buckets in a C-Buffer is $P \times N$. In this way, although more memory is needed, there is no locking mechanism required for the C-Buffer. For most of the currently available multiprocessor, shared-memory machines, we have found that this overhead is not overwhelming.

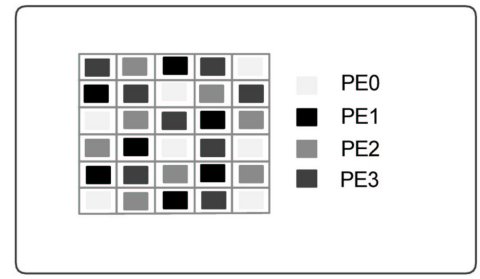


Fig. 8. UFLIC is parallelized by subdividing the texture space into tiles and randomly assigning these tiles to available processing elements (PEs).

In the second phase, the parallelization is fairly straightforward, where each processor independently traverses through the C-Buffers of the pixels in its own tiles to compute the convolution results. Note that now, within each C-Buffer, there are P buckets corresponding to the same computational time step. The convolution is computed by accumulating and normalizing the scattered image values from those buckets.

In the third phase, each processor simply applies the noise-jittered, high-pass filtering process independently to the convolution results of pixels belonging to its own tiles.

5 Results and Discussion

In this section, empirical results for evaluating the UFLIC algorithm are provided. We first present a performance analysis of our sequential and parallel implementations. We then show case studies of several CFD applications.

5.1 Performance Analysis

Data from two unsteady CFD simulations were used in our experiments. As shown in Table 1, we applied the UFLIC algorithm to a two-dimensional curvilinear surface of a delta wing model with a 287×529 texture resolution and to a two-dimensional curvilinear surface of an airfoil model with a 196×389 texture resolution. The machine used to generate the results is an SGI Onyx2 with 195 MHZ R10000 processors. Table 2 shows the breakdown of the average computation time using one processor for generating a texture image in an animation series. We used three computational time steps as the seed particle's life span. In the table, we show the time spent at each of the UFLIC's three phases: scattering pixel values, computing convolutions, and high-pass filtering. We found that the process of value scattering required more than 99 percent of the total execution time; the majority of the calculations provided the pathlines of seed particles. It is noteworthy that the computation time for data scattering is not necessarily proportional to the texture resolution. The reason is that seed particles in different flow fields, given the same life span, may travel different distances due to different flow velocities. This will result in a variation of the workload in the data scattering process, as we mentioned previously in the section on the parallel algorithm. In Table 2, we can see that the computational time of value scattering for the airfoil data set is longer than that for the delta wing data set, even though the texture resolution of the airfoil is smaller. This

TABLE 1
TEXTURE RESOLUTION ON SURFACES OF TWO UNSTEADY FLOW DATA SETS

Data Set	Texture Resolution
Delta Wing	287 × 529
Airfoil	196 × 389

TABLE 2
UFLIC COMPUTATION TIME (IN SECONDS) WITH ONE PROCESSOR

Data Set	Scattering	Convolution	Filtering	Total
Delta Wing	71.13	0.23	0.19	71.55
Airfoil	123.85	0.11	0.09	124.05

TABLE 3
AVERAGE PATHLINE LENGTH (IN PIXELS)

Data Set	Pathline Length
Delta Wing	40
Airfoil	130

TABLE 4
PERFORMANCE OF PARALLEL UFLIC: DELTA WING

CPUs	Time	Speedup	Efficiency
1	71.55	1	1.00
2	44.53	1.6	0.80
3	25.99	2.75	0.91
4	19.62	3.64	0.91
5	15.90	4.50	0.90
6	13.17	5.43	0.90
7	11.39	6.28	0.89

TABLE 5
PERFORMANCE OF PARALLEL UFLIC: AIRFOIL

CPUs	Time	Speedup	Efficiency
1	124.05	1	1.00
2	66.86	1.85	0.92
3	45.16	2.74	0.91
4	36.08	3.44	0.86
5	26.79	4.63	0.92
6	22.79	5.44	0.90
7	20.22	6.13	0.88

can be explained by Table 3, which compares the average pathline lengths of seed particles for the airfoil and the delta wing data sets.

In our parallel implementation, given P processors, we assign each processor $P \times 16$ tiles by dividing the two-dimensional texture space into $4P \times 4P$ tiles. Table 4 and Table 5 show the total execution times (in seconds), speed up factors, and parallel efficiencies of our parallel algorithm with up to seven processors. The parallel efficiency is defined by dividing the speed up factor by the number of processors. From the results, we observe that we achieve about 90 percent parallel efficiency, which is a very good load balance. Fig. 9 and Fig. 10 show the parallel speed up graphs.

5.2 Case Studies

In this section, we show the results of applying our new algorithm to several CFD applications. Animation is the ideal method to show our results; for the paper, we will show several snapshots from the animation.

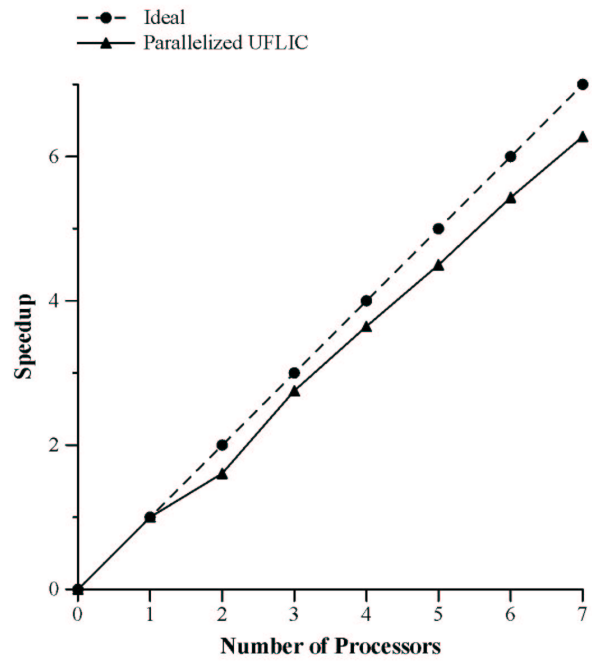


Fig. 9. Parallel speedup of UFLIC algorithm: delta wing.

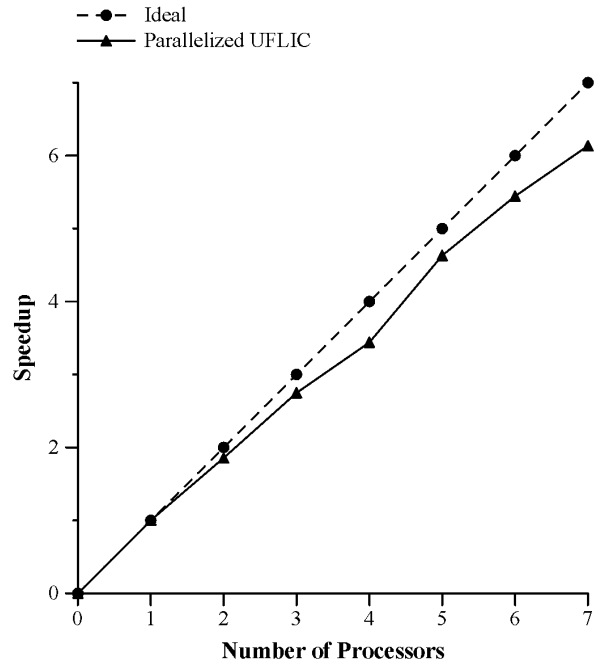


Fig. 10. Parallel speedup of UFLIC algorithm: airfoil.

There have been many CFD simulations performed to study flow phenomena about oscillating wings. Generally, a simulation is done using a two-dimensional airfoil. The first example is based on a simulation of unsteady two-dimensional turbulent flow over an oscillating airfoil, which pitches down and then up 11 degrees.

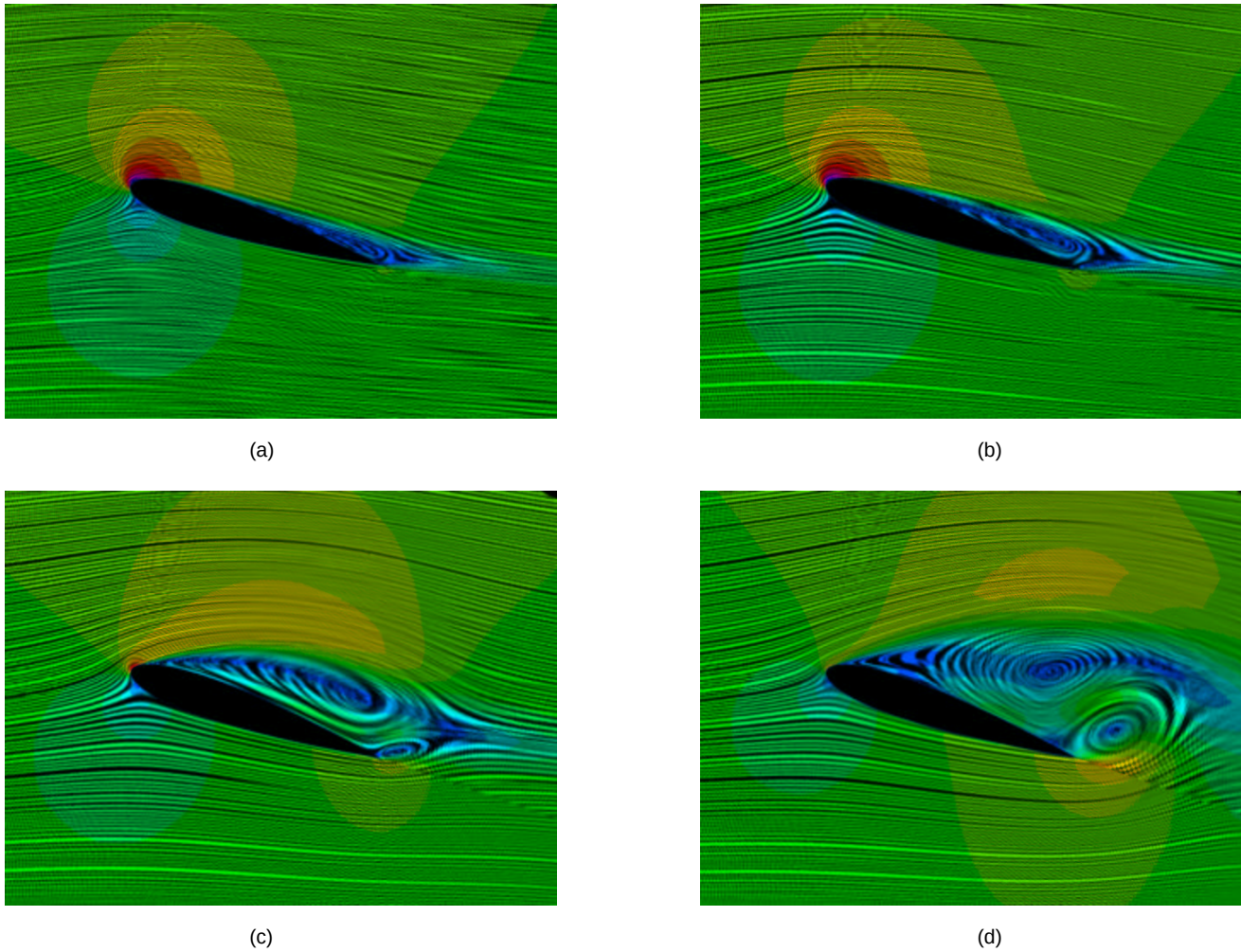


Fig. 11. Flow pattern surrounding an oscillating airfoil. The flow has high velocity near the leading edge of the airfoil (a) and (b). A clockwise primary vortex forms above the airfoil surface as it pitches downward. The vortex is then separated from the airfoil as it pitches upward, and a secondary vortex behind the trailing edge of the airfoil forms and gains strength over time (c) and (d).

The oscillational motion of the airfoil creates vortex shedding and vortex formation. Figs. 11a and 11b show the formation of the primary vortex rotating clockwise above the airfoil. The flow texture shown is colored by velocity magnitude. Blue indicates low velocity and magenta is high velocity. Initially, the velocity is high near the leading edge of the airfoil. As the airfoil pitches down and then up, the velocity decreases at the leading edge and increases near the trailing edge, a counterclockwise secondary vortex forms beyond the trailing edge of the airfoil, see Figs. 11c and 11d. Furthermore, the primary vortex gains strength, and the primary vortex separates from the airfoil.

We have compared the unsteady flows shown in Fig. 11 with the results of steady LIC. For the comparison, we used steady LIC to compute surface flows at each time step independently; then, we animated the steady LIC over time. The difference between this and the animation of UFLIC is very apparent. With our new algorithm, the animation reveals the dynamic behavior of the vortices during vortex formation and vortex shedding much more realistically than steady LIC.

Fig. 12 depicts a snapshot from a time sequence of flow patterns about several spiraling vortices from an unsteady flow simulation of four vortices. Some of the vortices orbit about other vortices in the flow. When compared to steady LIC, the spiraling motion of the vortices is not shown; instead, LIC reveals only the translation of the vortices.

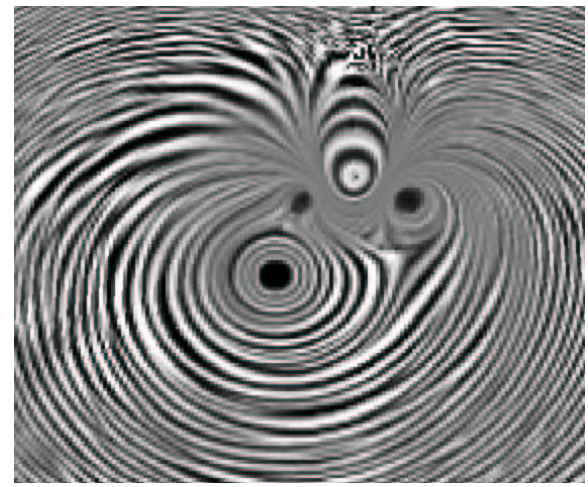


Fig. 12. Flow pattern generated from an unsteady flow simulation of four vortices. Some of the vortices have spiral motion and orbit about other vortices.

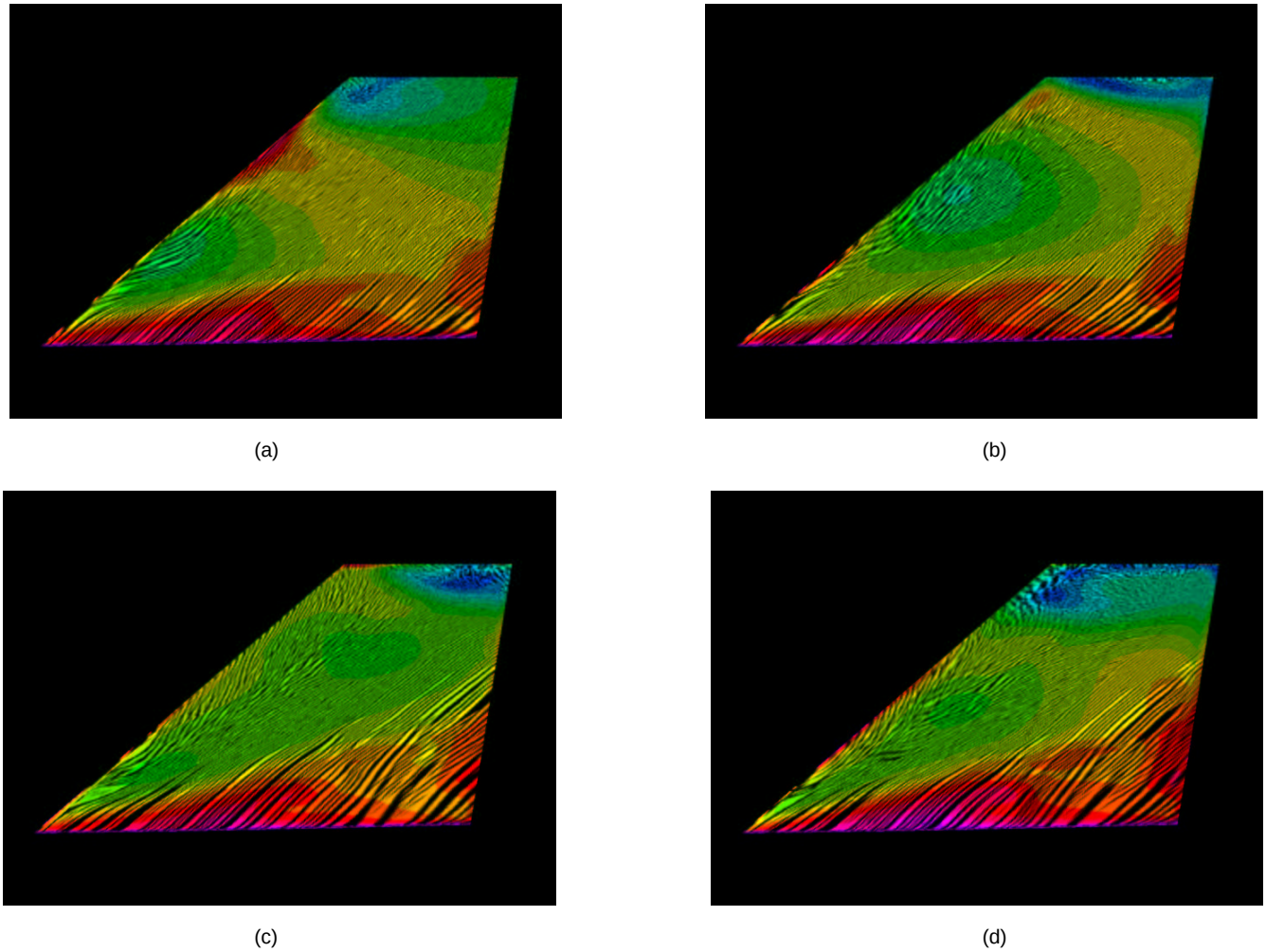


Fig. 13. At high angle of attack, the flow is highly unsteady and vortex bursting is frequent along the leading edge of the tail. The velocity magnitude changes rapidly along the leading edge, which indicates cycles of vortex bursting.

The next example is an unsteady flow simulation of the F/A-18 fighter aircraft. During air combat, the twin-tailed F/A-18 jet can have a high angle of attack. The simulation involves the study of tail buffet, which occurs when the vertical tails are immersed in unsteady flow and bursting of vortices along the leading edge of the tails is very frequent. This phenomenon can cause heavy loading on one of the vertical tails of the jet, which is of major safety concern. Figs. 13a and 13b show the first wave of vortex bursting along the leading edge of one of the vertical tails. The figure shows the outboard of the tail, e.g., the side of the tail that is facing away from the other vertical tail. Note the movement of the vortex, which occurs near the blue-shaded region, from the leading edge of the tail to the upper tip of the tail. Another wave of vortex bursting is shown in Figs. 13c and 13d. In the animation, the vortex bursting phenomenon is revealed dramatically.

The last example is taken from an unsteady flow simulation of a 65-degree sweep delta wing at a 30 degree angle of attack. For this simulation, there are several interesting flow separations, and reattachments occur along the leading edge of the delta wing at a zero degree static roll angle. Furthermore, vortex breakdown is present in the unsteady flow. Fig. 14 shows a snapshot of the surface flow at a given time step. The velocity magnitude color contours give some indication of the change in velocity over time. Along the leading edge of the wing, the flow velocity is relatively high compared to the velocity near the wing body.

Based on the examples presented in this section, some observations can be made regarding the surface flows generated by our new algorithm. First, because the flow is unsteady, blurriness is likely to occur in regions where the flow is changing rapidly. Unlike the surface flow patterns generated from regular LIC, the flow lines generated using our unsteady LIC may have different line widths. This could be attributed to changes in the velocity magnitude and the flow direction.

6 Conclusions and Future Work

We have presented UFLIC, an Unsteady Flow Line Integral Convolution algorithm, for visualizing vector data in unsteady flow fields. Using the time-accurate value-scattering scheme and the successive feed-forward process, our new convolution algorithm can accurately model the flow advection and create highly coherent flow animations. The results from several case studies using our algorithm have shown that the new technique is very effective in capturing dynamic features in unsteady flow fields.

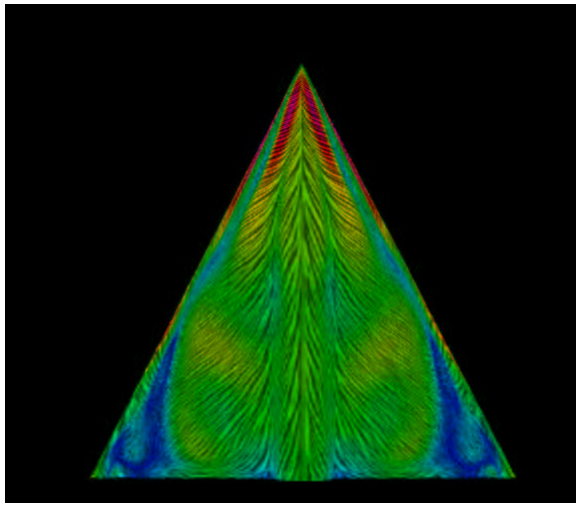


Fig. 14. At 30 degree angle of attack, flow separations and reattachments occur along leading edge of the delta wing. Near the center of the wing body, the flow is relatively steady, as indicated by the velocity magnitude color contours.

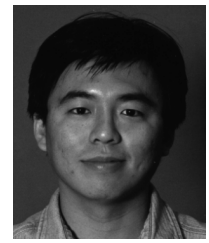
Future work includes applying our method to three-dimensional unsteady flow data sets. In addition, we would like to compare our unsteady LIC method with the *spot noise* technique introduced by van Wijk [12]. Spot Noise is an effective method for creating flow texture patterns. The final texture image quality is based on the distribution and the shape of spots. De Leeuw and van Wijk [13] enhanced the spot noise technique by bending spots based on local stream surfaces. The objective is to produce more accurate flow texture patterns near flow regions with high curvature. To extend the spot noise technique for unsteady flows, there are a few key issues to be resolved. For instance, as spots are advected over time, the distribution of the spots can change rapidly. A challenge is to maintain the coherence of the spots over time. Another consideration is that spot bending assumes the flow is steady over the local stream surfaces; however, for unsteady flows, this assumption may not be true. We plan to look into these issues.

ACKNOWLEDGMENTS

This work was supported in part by NASA contract NAS2-14303. We would like to thank Neal Chaderjian, Ken Gee, Shigeru Obayashi, and Ravi Samtaney for providing their data sets. Special thanks to Randy Kaemmerer for his meticulous proofreading of this manuscript, and to Michael Cox and David Ellsworth for interesting discussions and valuable suggestions in the parallel implementation. We also thank Tim Sandstrom, Gail Felchle, Chris Henze, and other members in the Data Analysis Group at NASA Ames Research Center for their helpful comments, suggestions, and technical support.

REFERENCES

[1] D.A. Lane, "Visualization of Time-Dependent Flow Fields," *Proc. Visualization '93*, pp. 32-38, 1993.
[2] D.A. Lane, "Visualizing Time-Varying Phenomena in Numerical Simulations of Unsteady Flows," *Proc. 34th Aerospace Science Meeting and Exhibit*, AIAA-96-0048, 1996.
[3] B. Cabral and C. Leedom, "Imaging Vector Fields Using Line Integral Convolution," *Proc. SIGGRAPH '93*, pp. 263-270, 1993.
[4] L.K. Forssell and S.D. Cohen, "Using Line Integral Convolution for Flow Visualization: Curvilinear Grids, Variable-Speed Animation, and Unsteady Flows," *IEEE Trans. Visualization and Computer Graphics*, vol. 1, no. 2, pp. 133-141, June 1995.
[5] H.-W. Shen and D.L. Kao, "Uflic: A Line Integral Convolution for Visualizing Unsteday Flows," *Proc. Visualization '97*, pp. 317-322, 1997.
[6] D. Stalling and H.-C. Hege, "Fast and Resolution Independent Line Integral Convolution," *Proc. SIGGRAPH 95*, pp. 249-256, 1995.
[7] H.-W. Shen, C.R. Johnson, and K.-L. Ma, "Visualizing Vector Fields Using Line Integral Convolution and Dye Advection," *Proc. 1996 Symp. Volume Visualization*, pp. 63-70, 1996.
[8] A. Okada and D.L. Kao, "Enhanced Line Integral Convolution With Flow Feature Detection," *Proc. IS&T/SPIE Electronic Imaging '97*, pp. 206-217, 1997.
[9] M.-H. Kiu and D. Banks, "Multi-Frequency Noise for LIC," *Proc. Visualization '96*, pp. 121-126, 1996.
[10] B. Jobard and W. Lefer, "The Motion Map: Efficient Computation of Steady Flow Animations," *Proc. Visualization '97*, pp. 323-328, 1997.
[11] M. Zöckler, D. Stalling, and H.-C. Hege, "Parallel Line Integral Convolution," *Proc. First Eurographics Workshop Parallel Graphics and Visualization*, pp. 111-128, Sept. 1996.
[12] J.J. van Wijk, "Spot Noise: Texture Synthesis for Data Visualization," *Computer Graphics*, vol. 25, no. 4, pp. 309-318, 1991.
[13] W. de Leeuw and J.J. van Wijk, "Enhanced Spot Noise for Vector Field Visualization," *Proc. Visualization '95*, pp. 233-239, 1995.



Han-Wei Shen received a BS in computer science from National Taiwan University in 1988, an MS in computer science from the State University of New York at Stony Brook in 1992, and a PhD in computer science from the University of Utah in 1998. Since 1996, he has been a research scientist with MRJ Technology Solutions at NASA Ames Research Center. His research interests include scientific visualization, computer graphics, digital image processing, and parallel rendering. In particular, his current research and publications focus on a variety of topics in flow visualization, isosurface extraction, and volume visualization.

David Kao received his Ph.D. from Arizona State University in 1991 and has been a computer scientist working at NASA Ames Research Center since then. He is currently developing numerical flow visualization techniques and software tools for computational fluid dynamics applications in aeronautics.

His research interests include flow visualization, scientific visualization, and computer graphics. He is a research advisor for the National Research Council and a mentor for the Computer Engineering Program at Santa Clara University.