5

Vector Field Visualization

Vector fields

A static vector field $\mathbf{v}(\mathbf{x})$ is a vector-valued function of space.

A time-dependent vector field $\mathbf{v}(\mathbf{x},t)$ depends also on time.

In the case of velocity fields, the terms steady and unsteady flow are used.

The dimensions of **x** and **v** are equal, often 2 or 3, and we denote components by *x,y,z* and *u,v,w*:

$$\mathbf{x} = (x,y,z),\ \mathbf{v} = (u,v,w)$$

Sometimes a vector field is defined on a surface $\mathbf{x}(i,j)$. The vector field is then a function of parameters and time:

$$\mathbf{v}(i,j,t)$$

An elementary visualization is to draw arrows

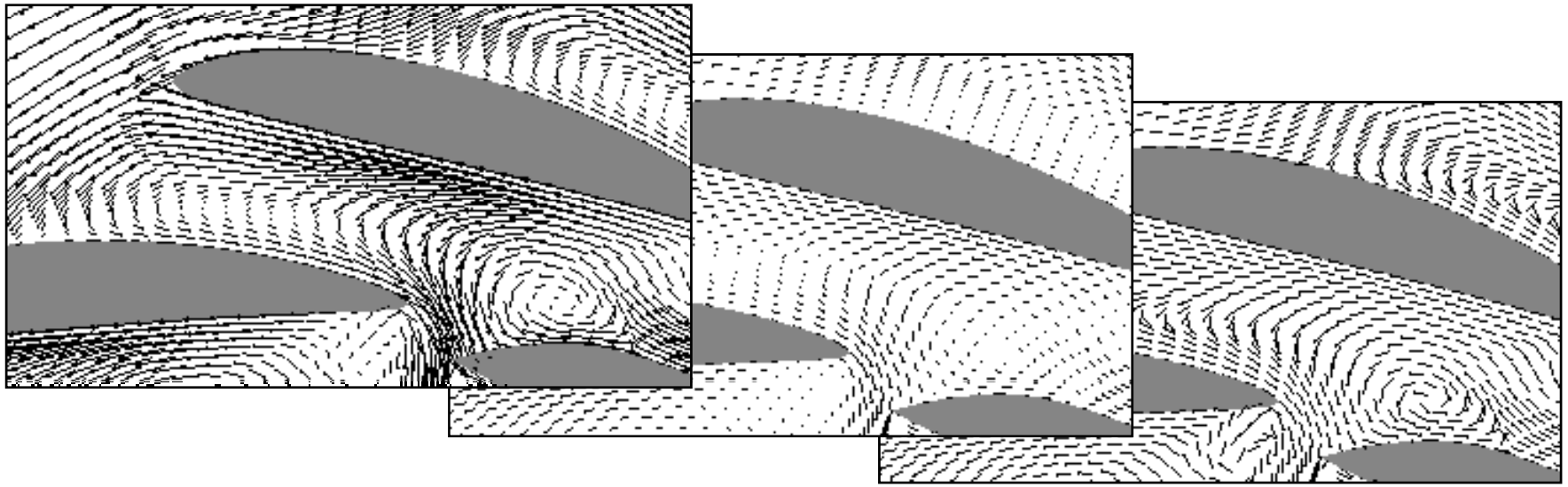
- at the data points (grid nodes or cell centers), or
- at a new (uniform) grid, for 3D fields often a 2D slice

Arrows can visualize:

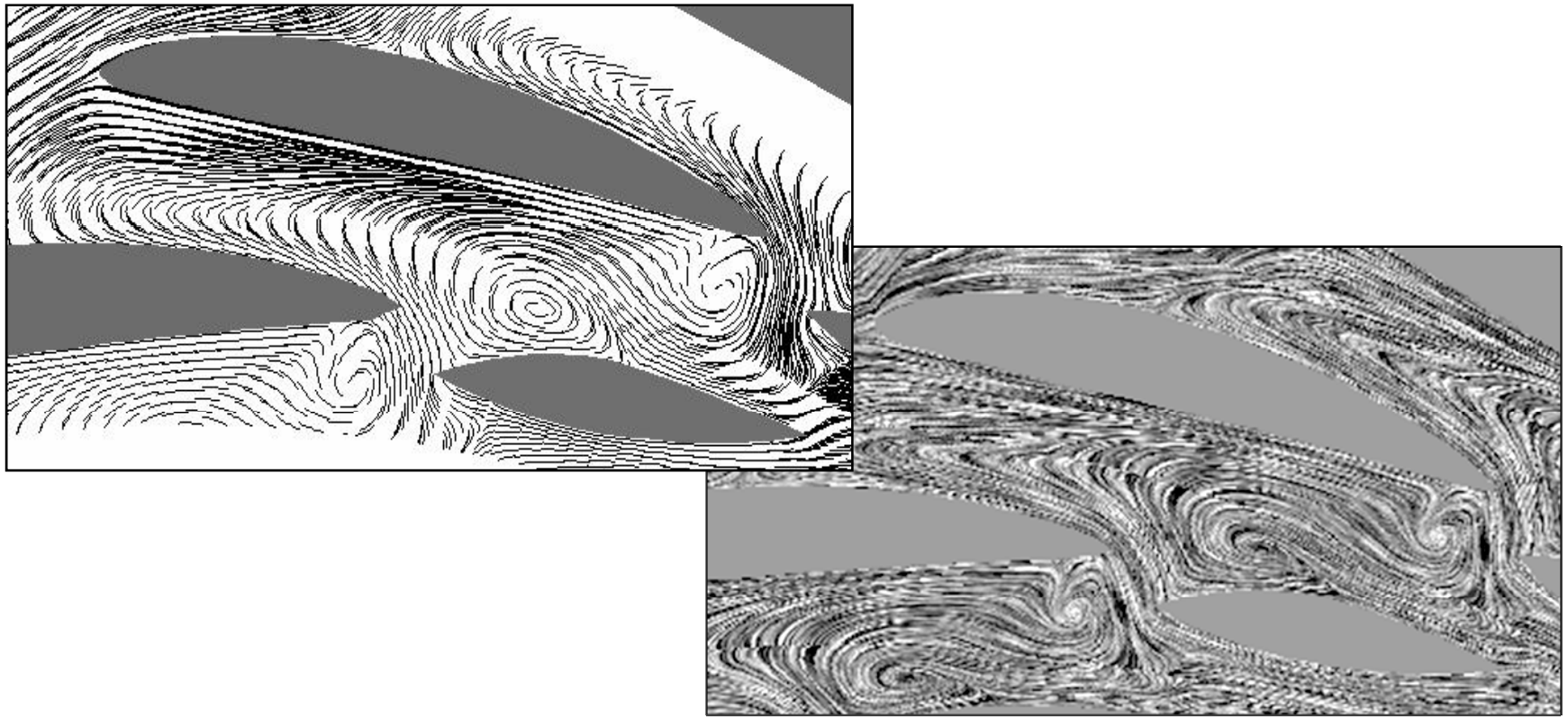
- direction
- relative magnitude (when approproiately scaled)
- time dependency (when animated)

Problems of visualization with arrows:

- It is not clear whether arrows represent vector values at the start point or at the midpoint of the arrow
- Often no satisfactory scaling factor exists:
  - large scaling: Arrows occlude each other
  - small scaling: Direction is not recognizable in some regions
  - fixed length: Magnitude information is lost



Streamline-based techniques for comparison: streamlets (short streamlines) and LIC (line integral convolution)



Magnitude information can be added by coloring or with animated texture.

Vector fields as ODEs

For simplicity, the vector field is now interpreted as a velocity field.

Then the field $\mathbf{v}(\mathbf{x},t)$ describes the connection between location and velocity of a (massless) particle.

It can equivalently be expressed as an ordinary differential equation

$$\dot{\mathbf{x}}(t) = \mathbf{v}(\mathbf{x}(t),t)$$

This ODE, together with an initial condition

$$\mathbf{x}(t_0) = \mathbf{x}_0 \ ,$$

is a so-called initial value problem (IVP).

Its solution is the integral curve

$$\mathbf{x}(t) = \mathbf{x}_0 + \int_{t_0}^{t} \mathbf{v}(\mathbf{x}(\tau),\tau)\, d\tau$$

The integral curve is a pathline, describing the path of a massless particle which was released at time t_0 at position x_0.

Remark: $t < t_0$ is allowed.

For static fields, the ODE is autonomous:

$$\dot{\mathbf{x}}(t) = \mathbf{v}(\mathbf{x}(t))$$

and its integral curves

$$\mathbf{x}(t) = \mathbf{x}_0 + \int_{t_0}^{t} \mathbf{v}(\mathbf{x}(\tau))\, d\tau$$

are called field lines, or (in the case of velocity fields) streamlines.

In static vector fields, pathlines and streamlines are identical.

In time-dependent vector fields, instantaneous streamlines can be computed from a "snapshot" at a fixed time T (being in a static vector field)

$$\mathbf{v}_T(\mathbf{x}) = \mathbf{v}(\mathbf{x}, T)$$

In practice, time-dependent fields are often given as a dataset per time step. Each dataset is then a snapshot.

Streamlines, pathlines, streaklines, timelines

Besides streamlines and pathlines, two more types of lines can be obtained by integration: streaklines and timelines.

A streakline is obtained by continually releasing particles at a fixed location and taking a snapshot at a fixed time.

A timeline is obtained by simultaneously releasing particles densely on a seed curve and taking a snapshot at a fixed (later) time. This concept can be extended to time surfaces, obtained by releasing particles on a surface (e.g. rectangle or sphere).

Computing streaklines or timelines is more expensive than solving a single IVP.

Algorithm for streakline:

- for time samples $t_0, t_1, \cdots, t_n$ solve the IVP

$$\dot{\mathbf{x}}_i(t) = \mathbf{v}(\mathbf{x}_i(t), t)$$

$$\mathbf{x}_i(t_i) = \mathbf{y}$$

- extract from each integral curve $\mathbf{x}_i(t)$ the point $\mathbf{x}_i(t_n)$
- connect these points

The result is a streakline for time t_n.

In the numerical computation the temporal interval must be adaptively refined if two successive particles diverge too much.

Algorithm for timeline:

- for point samples $\mathbf{y}_0, \mathbf{y}_1, \cdots, \mathbf{y}_n$ on the seed curve solve the IVP

$$\dot{\mathbf{x}}_i(t) = \mathbf{v}(\mathbf{x}_i(t), t)$$

$$\mathbf{x}_i(t_0) = \mathbf{y}_i$$

- extract from the integral curve $\mathbf{x}_i(t)$ the point $\mathbf{x}_i(T)$
- connect these points

The result is a timeline for time *T*.

In the numerical computation the spatial interval must be adaptively refined if two neighbor particles diverge too much.

Comparison of techniques:

(1) Pathlines:

- are physically meaningful
- allow comparison with experiment (observe marked particles)
- are well suited for dynamic visualization (of particles)

(2) Streamlines:

- are only geometrically, not physically meaningful
- are easiest to compute (no temporal interpolation, single IVP)
- are better suited for static visualization (prints)
- don't intersect (under reasonable assumptions)

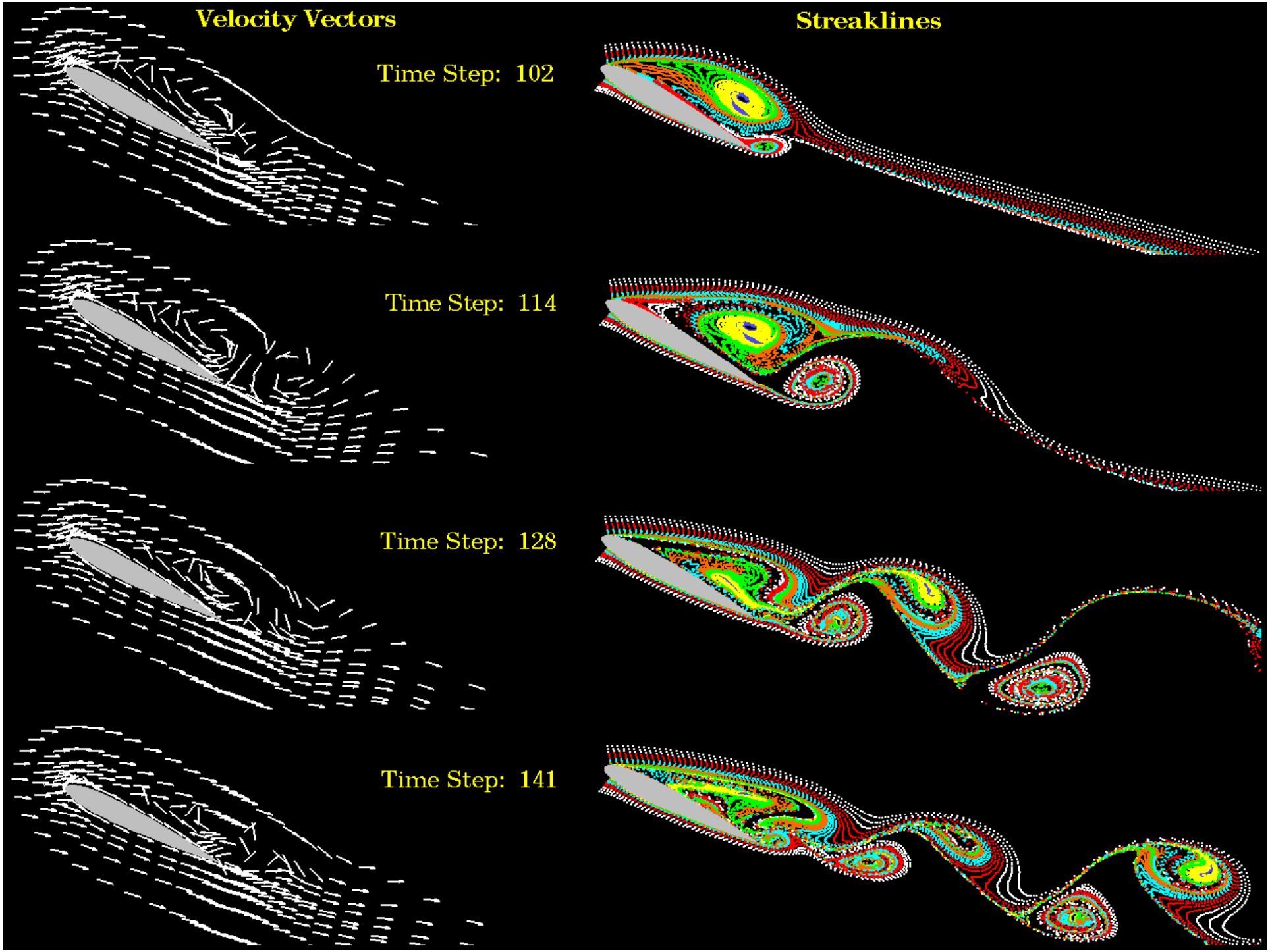
(3) Streaklines:

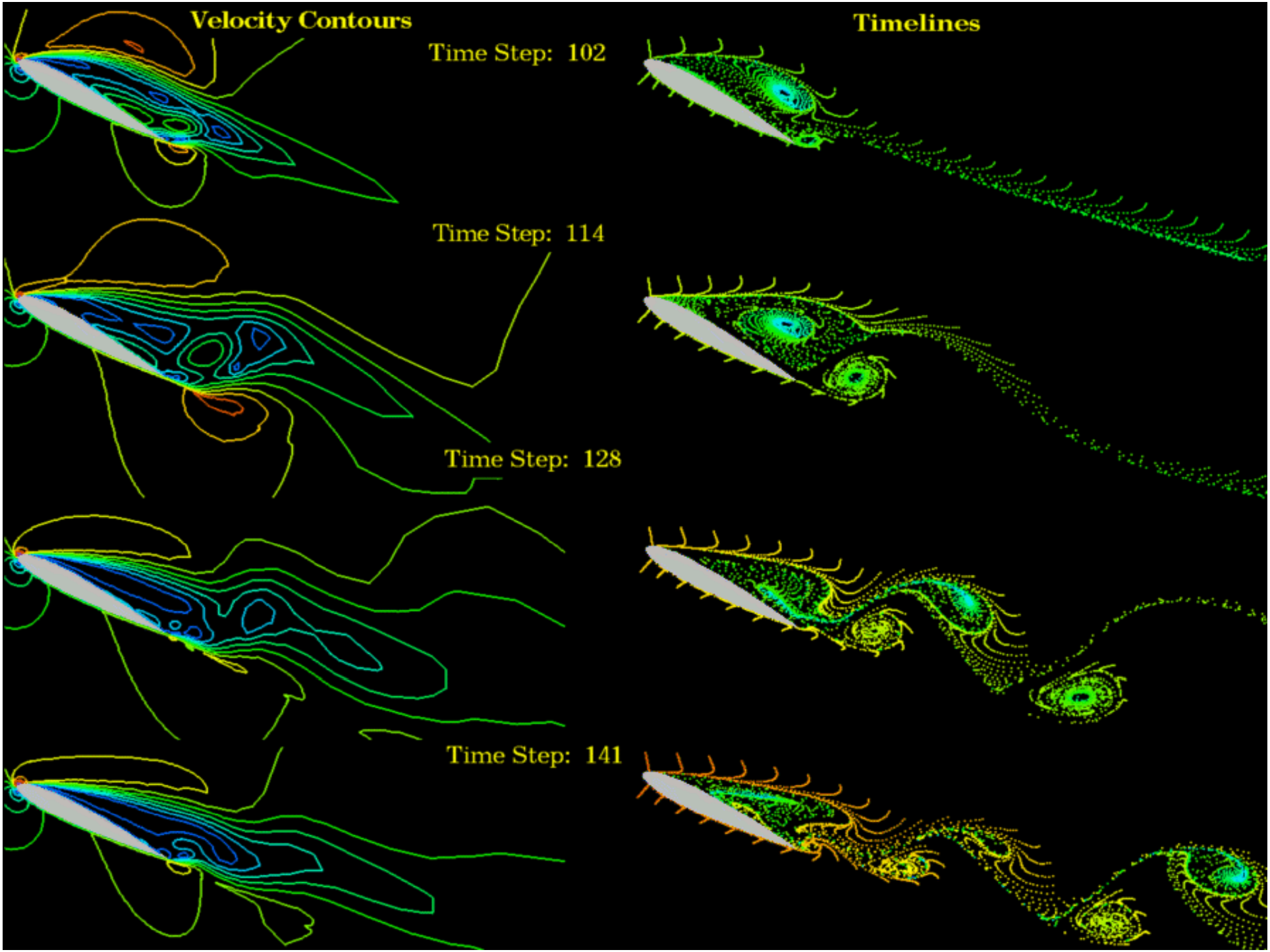
- are physically meaningful
- allow comparison with experiment (dye injection)
- are well suited for static and dynamic visualization
- good choice for fast moving vortices
- can be approximated by set of disconnected particles

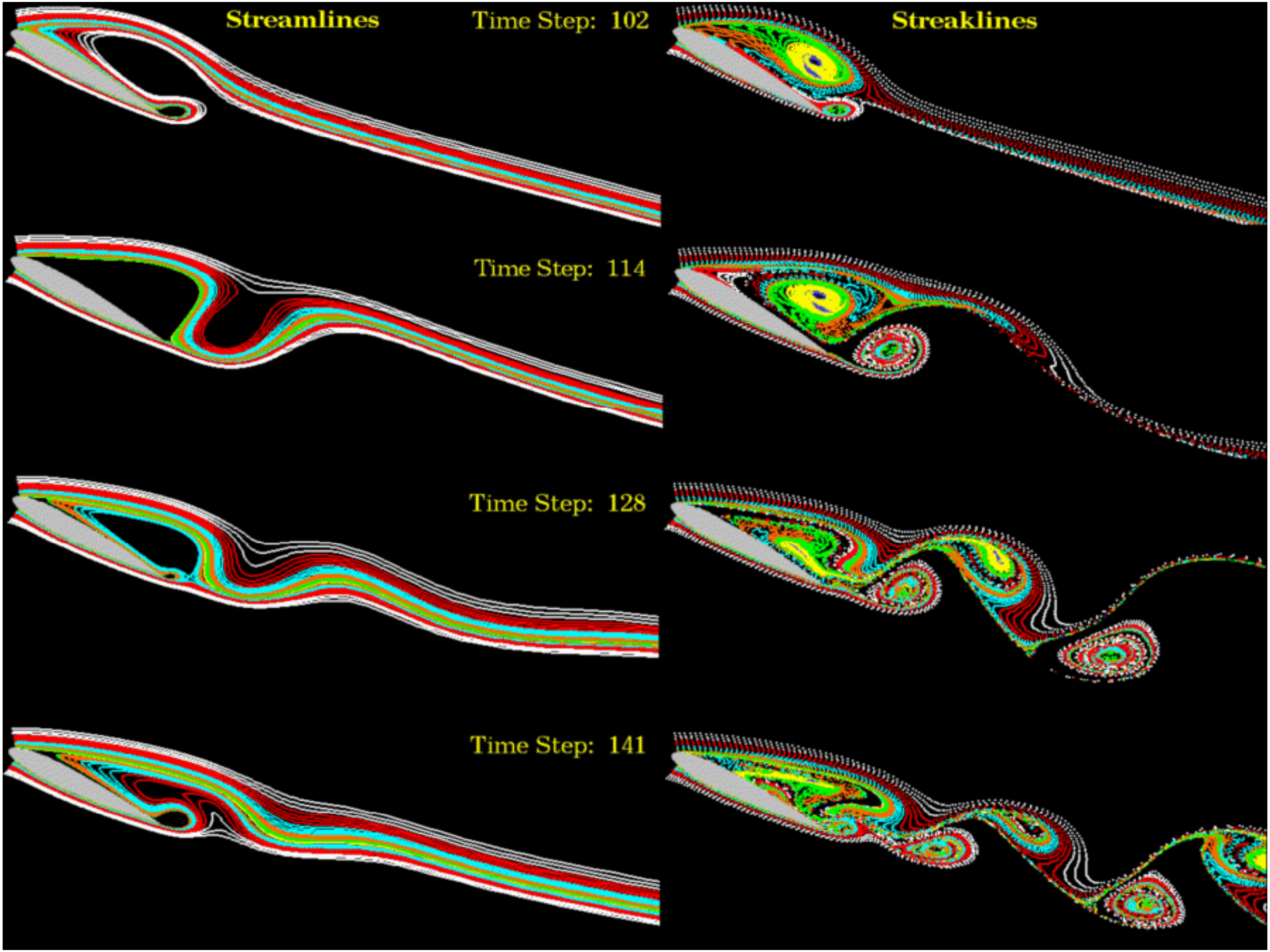
(4) Timelines:

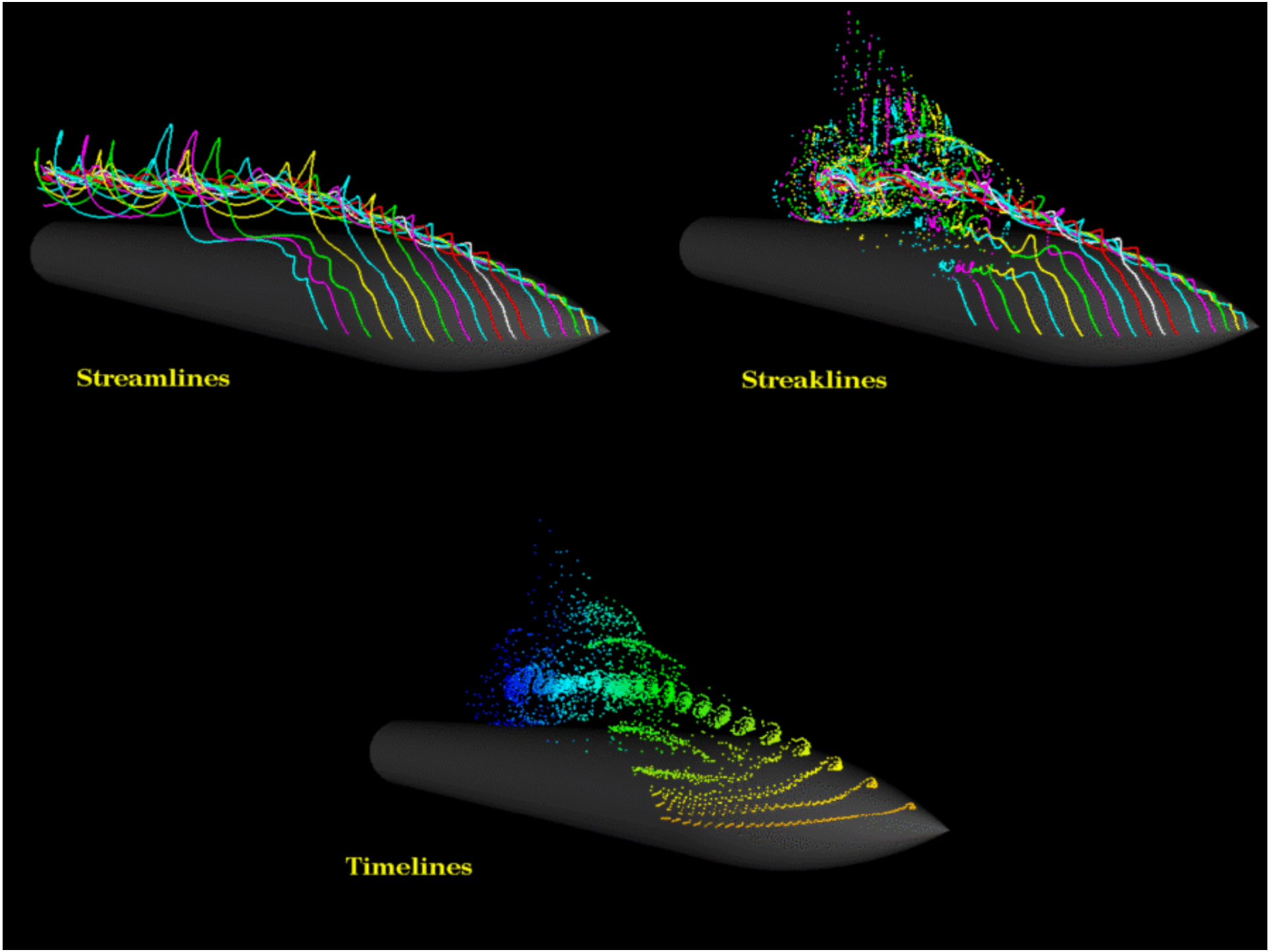
- are physically meaningful
- are well suited for static and dynamic visualization
- can be approximated by set of disconnected particles

Visual comparison of the techniques (from a NASA web page):

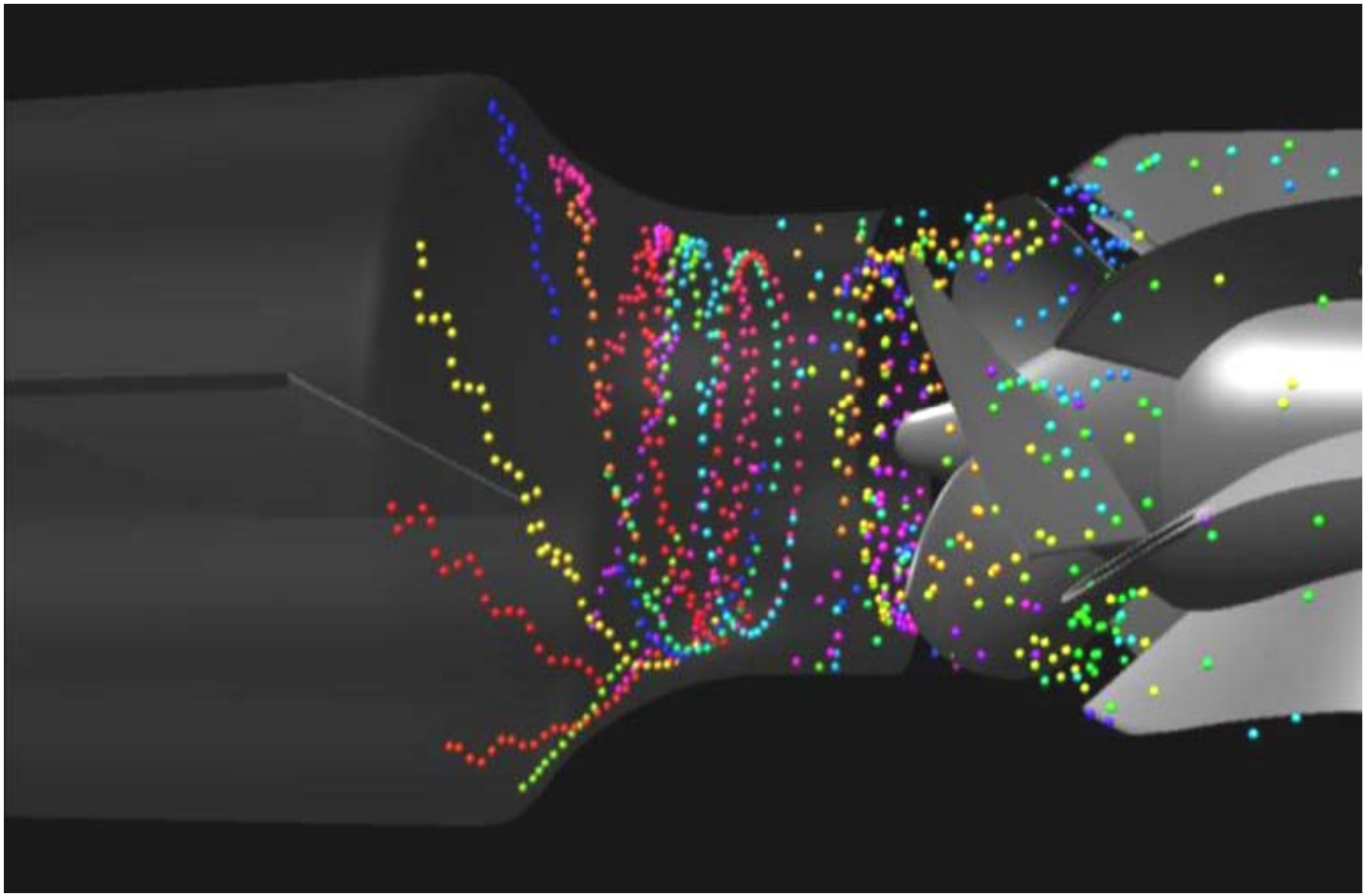

Velocity Vectors
Streaklines
Time Step: 102
Time Step: 114
Time Step: 128
Time Step: 141


Velocity Contours
Timelines
Time Step: 102
Time Step: 114
Time Step: 128
Time Step: 141

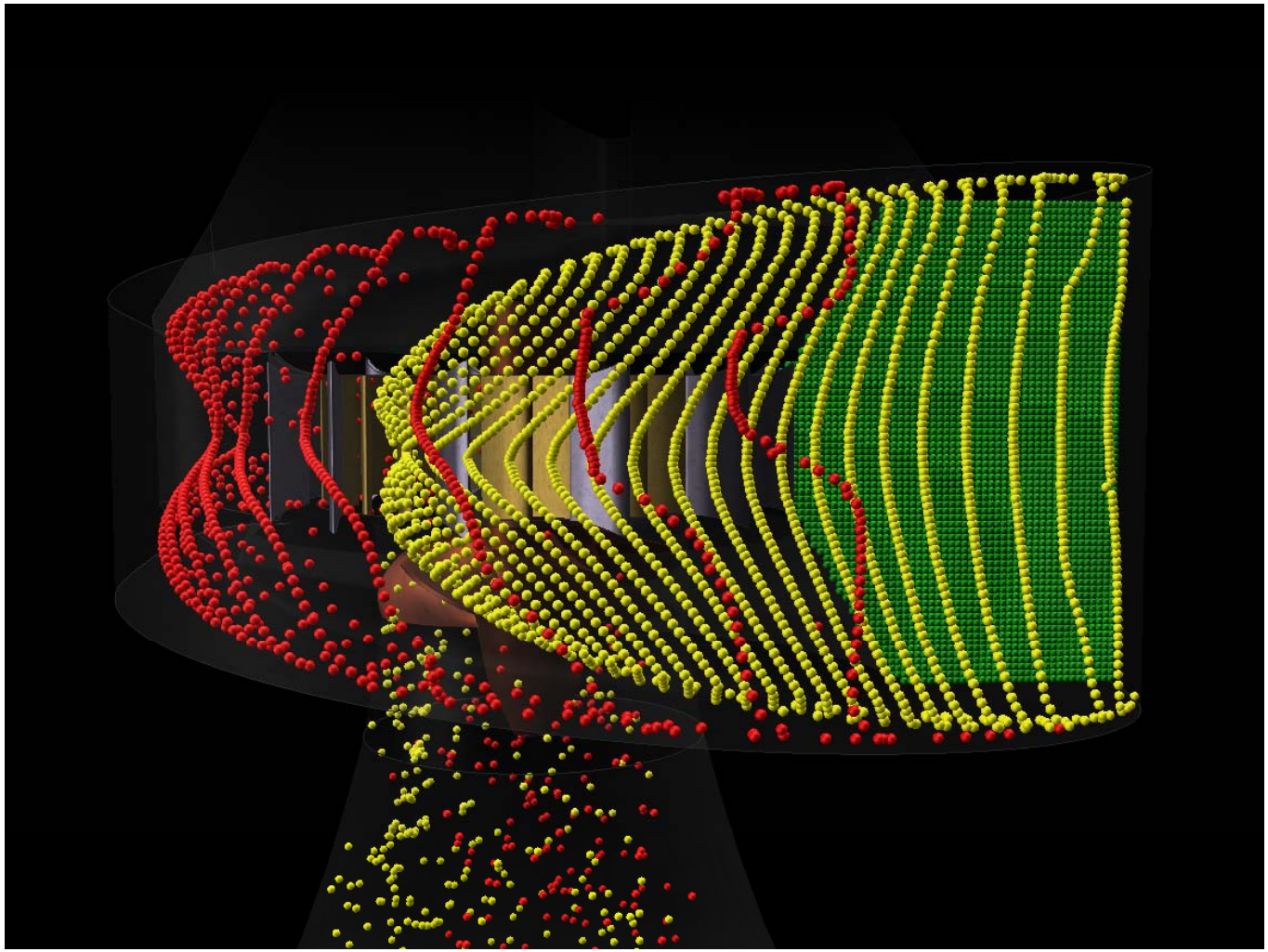

Streamlines
Time Step: 102
Streaklines
Time Step: 114
Time Step: 128
Time Step: 141


Streamlines
Streaklines
Timelines

Further example of (discrete) streaklines:



Example of discrete time surfaces:



Streamline integration

Outline of algorithm for numerical streamline integration (with obvious extension to pathlines):

Inputs:

- static vector field $\mathbf{v}(\mathbf{x})$
- seed points with time of release $(\mathbf{x}_0, t_0)$
- control parameters:
  - step size (temporal, spatial, or in local coordinates)
  - step count limit, time limit, etc.
  - order of integration scheme

Output:

- streamlines as "polylines", with possible attributes (interpolated field values, time, speed, arc length, etc.)

Preprocessing:

- set up search structure for point location
- for each seed point:
  - global point location: Given a point **x**,
    find the cell containing **x** and the local coordinates (ξ, η, ζ)
    or ir the grid is structured:
    find the computational space coordinates $(i + \xi,\ j + \eta,\ k + \zeta)$
  - If **x** is not found in a cell, remove seed point

Integration loop, for each seed point **x**:

- interpolate ***v*** trilinearly to local coordinates (ξ, η, ζ)
- do an integration step, producing a new point **x'**
- incremental point location: For position **x'** find cell and local coordinates (ξ', η', ζ') making use of information (coordinates, local coordinates, cell) of old point **x**

Termination criteria:

- grid boundary reached
- step count limit reached
- optional: velocity close to zero
- optional: time limit reached
- optional: arc length limit reached

Integration step: widely used integration methods:

- Euler (used only in special speed-optimized techniques, e.g. GPU-based texture advection)

$$\mathbf{x}_{new} = \mathbf{x} + \mathbf{v}(\mathbf{x}, t) \cdot \Delta t$$

- Runge-Kutta, 2nd or 4th order

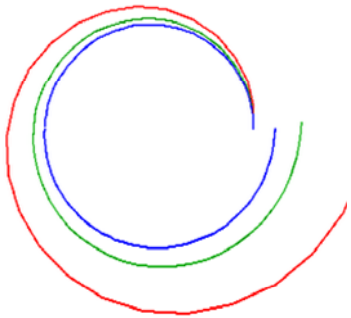
Higher order than 4th?

- often too slow for visualization
- study (Yeung/Pope 1987) shows that, when using standard trilinear interpolation, interpolation errors dominate integration errors.

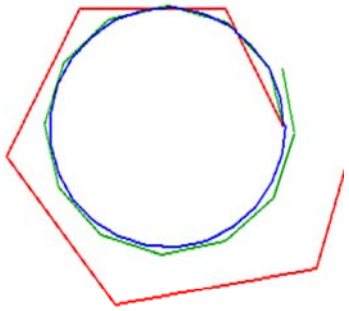
Example: Velocity field of rigid rotation

$$\mathbf{v}(\mathbf{x}) = (-\omega y, \omega x)$$

- **v** is linear, hence bilinear interpolation is exact
- observed errors are integration errors



Euler, small steps:
$\Delta t = 1/4, 1/8, 1/16$



RK2, larger steps:
$\Delta t = 1, 1/2, 1/4$

Choosing the step size:

Several options: Δt

- fixed time step Δs
  - used for animated particles
- fixed spatial step
  - time step derived from spatial step $\Delta t = \Delta s \,/\, \|\mathbf{v}(\mathbf{x})\|$, iteratively corrected
  - used for methods such as LIC
- adaptive
  - adapting to grid resolution, i.e. cell size, or
  - adapting to data variation (Runge-Kutta-Fehlberg method)
  - used for interactive viewing (with zooming)

The stencil walk algorithm

Incremental point location is nontrivial for curvilinear and unstructured grids.

Buning's stencil walk algorithm solves this problem.

Given:

- point with coordinates **x**
- cell (as three parameters (i, j, k) or as index *c,* resp.)
- local coordinates (ξ, η, ζ)
- coordinates of a new point **x'**

Wanted:

- new cell, as (i', j', k') or *c'* resp.
- new local coordinates (ξ', η', ζ')

In a first phase the algorithm finds the cell containing **x'** by doing iteratively:

- take the difference vector $\Delta\mathbf{x} = \mathbf{x}' - \mathbf{x}$
- intersect the ray $\mathbf{x} + t\Delta\mathbf{x}$ with the cell boundary, giving a *t* value
- if $t \geq 1$ the point **x'** lies in the current cell and iteration can be stopped
- otherwise move to the neighbor cell adjacent at the intersection point
- if no such cell exists terminate with failure
- set the cell centroid as the new **x** for the next iteration

The sub-problem of intersecting a ray with the cell boundary is solved as follows:

- Linearize the coordinate transform $\varphi : (\xi, \eta, \zeta) \mapsto (x, y, z)$

  in the point $\mathbf{x} = (x, y, z)$, i.e. compute the Jacobian

$$\mathbf{J} = \frac{\partial(x, y, z)}{\partial(\xi, \eta, \zeta)} = \left[\frac{\partial \varphi}{\partial \xi} \;\middle|\; \frac{\partial \varphi}{\partial \eta} \;\middle|\; \frac{\partial \varphi}{\partial \zeta} \right]$$

- Using **J** transform the difference vector $\Delta\mathbf{x} = \mathbf{x}' - \mathbf{x}$ into the local coordinate frame of the cell

$$(\Delta\xi, \Delta\eta, \Delta\zeta) = J^{-1}\Delta\mathbf{x}$$

- Find the intersection of the ray

$$(\xi,\eta,\zeta)+t(\Delta\xi,\Delta\eta,\Delta\zeta)$$

with the cell boundary:

  - having equations $\xi,\eta,\zeta = 0$
  - and for hex cell: $\xi,\eta,\zeta = 1$
    resp. for tet cell: $\xi+\eta+\zeta = 1$
  - and inequalities: $0 \le \xi,\eta,\zeta \le 1$

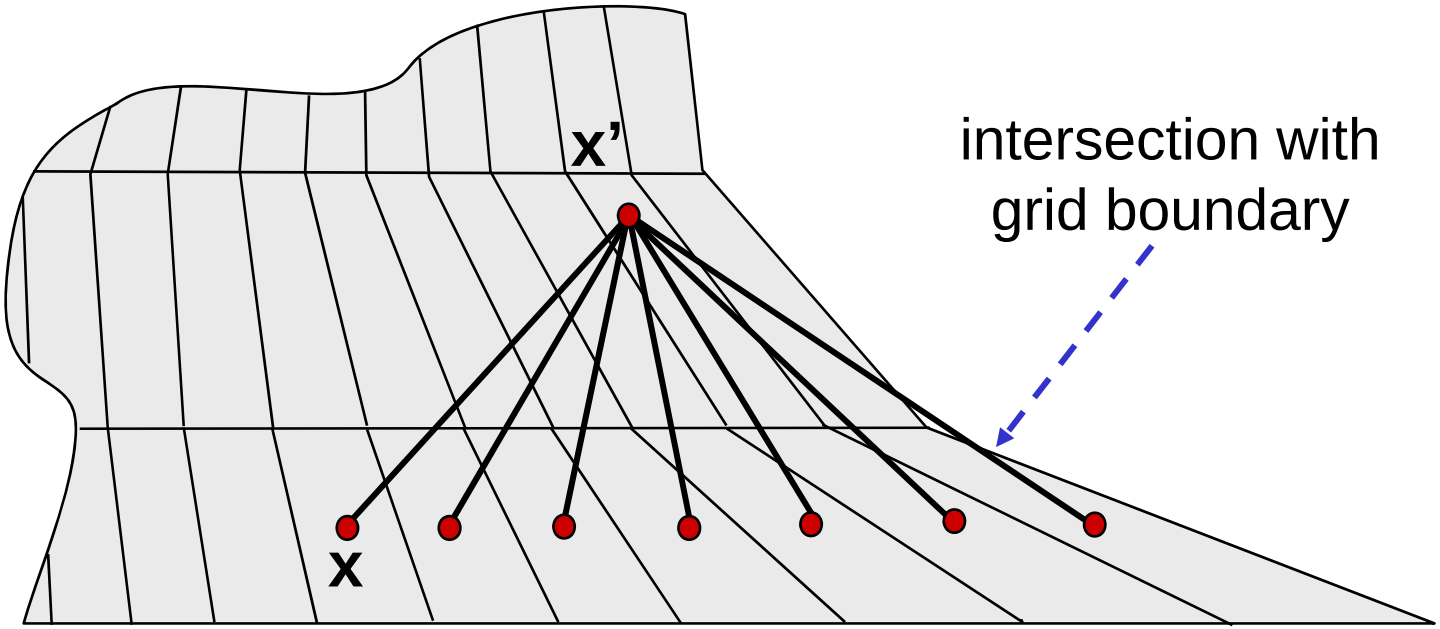
Due to linearization the point is not exact but in most cases (!) the correct neighbor cell is found.

Problem of original stencil walk algorithm:

If cells are sufficiently skewed, the algorithm can walk away from the target cell.

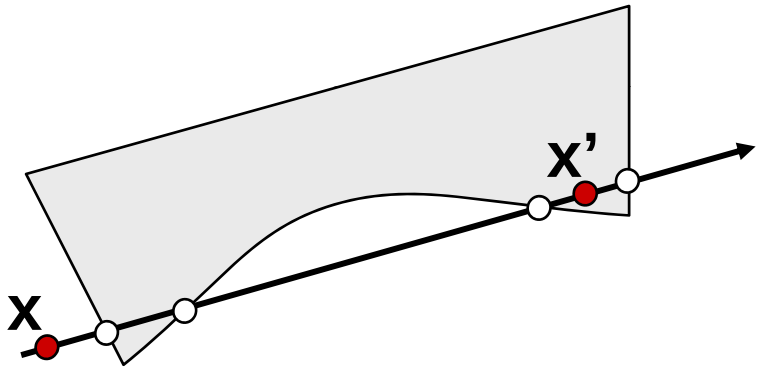
This happens also with correctly computed intersection points.

Example:



The problem can be solved with a modification of the algorithm:

- Keep ray (**x, x'**) unchanged, i.e. follow the ray from **x** to **x'**
- new problem: cell faces of type quadrangle (nonplanar!) can be intersected twice by the ray!



- therefore, exact intersection points of the ray with the bilinear surface patches must be calculated

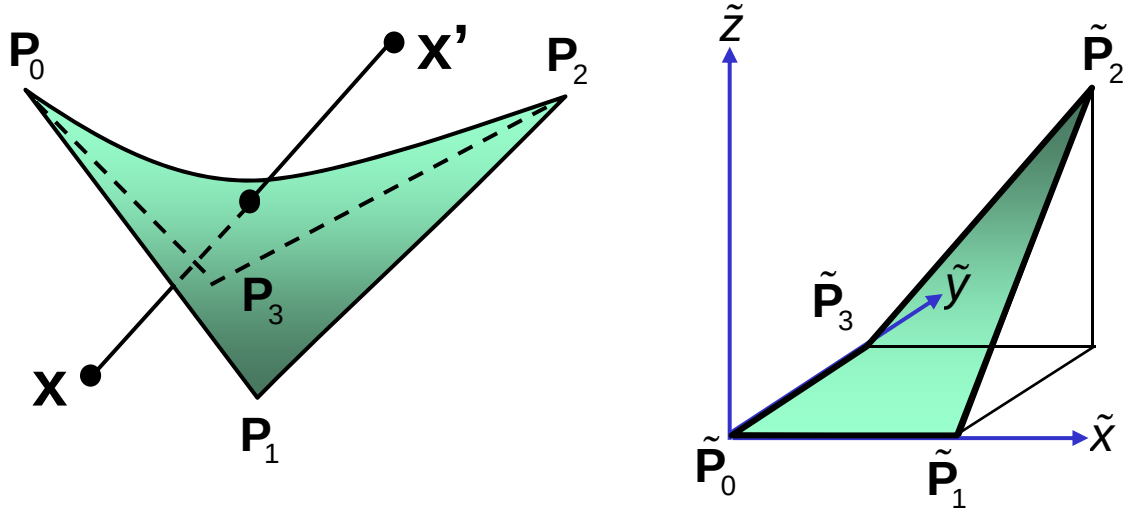
Exact intersection calculation:

- If the quadrangle is nonplanar, the four corners $\mathbf{P}_0$, $\mathbf{P}_1$, $\mathbf{P}_2$, $\mathbf{P}_3$ can be mapped to

$$\tilde{\mathbf{P}}_0 = (0,0,0),\ \tilde{\mathbf{P}}_1 = (1,0,0),\ \tilde{\mathbf{P}}_2 = (1,1,1),\ \tilde{\mathbf{P}}_3 = (0,1,0)$$

  by the affine transformation

$$\tilde{\mathbf{x}} = \left(\tilde{\mathbf{P}}_1 \middle| \tilde{\mathbf{P}}_2 \middle| \tilde{\mathbf{P}}_3\right)\left(\mathbf{P}_1 \middle| \mathbf{P}_2 \middle| \mathbf{P}_3\right)^{-1}(\mathbf{x} - \mathbf{P}_0)$$



- The bilinear surface containing $\mathbf{P}'_0, \mathbf{P}'_1, \mathbf{P}'_2, \mathbf{P}'_3$ is the hyperbolic paraboloid

$$z = xy$$

- Inserting the transformed view ray $\tilde{\mathbf{x}} + t\Delta\tilde{\mathbf{x}}$ leads to a quadratic equation for *t*:

$$\left(\tilde{z} + t\Delta\tilde{z}\right) = \left(\tilde{x} + t\Delta\tilde{x}\right)\left(\tilde{y} + t\Delta\tilde{y}\right)$$

- If real solutions with $0 < t < 1$ exist, the intersection points $\left(\tilde{x}_i, \tilde{y}_i, \tilde{z}_i\right)$ are computed and transformed back

In the second phase the stencil walk algorithm computes the local coordinates of the point in the cell known to contain it.

The local coordinates are the inverse of the coordinate function

$$\varphi : (\xi, \eta, \zeta) \mapsto (x, y, z)$$

evaluated at the given point $\mathbf{x} = (x, y, z)$

However, the trilinear function φ is a cubic polynomial and its inverse is a sixth-degree polynomial.

The problem is therefore solved with Newton's method.

Initialization:

- Let **x** be the cell centroid
- The local coordinates (ξ,η,ζ) are then:
  - for a hex cell: (0.5, 0.5, 0.5)
  - for a tet cell : (0.25, 0.25, 0.25)

Loop:

Repeat while the error $\Delta\mathbf{x} = \mathbf{x}' - \mathbf{x}$ is above a given threshold:

- Compute the (vector) coefficients **a**, **b**, ... of φ :

$$\varphi(\xi,\eta,\zeta) = \mathbf{a}\xi\eta\zeta + \mathbf{b}\xi\eta + \mathbf{c}\xi\zeta + \mathbf{d}\eta\zeta + \mathbf{e}\xi + \mathbf{f}\eta + \mathbf{g}\zeta + \mathbf{h}$$

- Evaluate the Jacobian in (ξ,η,ζ) :

$$\mathbf{J} = \frac{\partial(x,y,z)}{\partial(\xi,\eta,\zeta)} = \left(\frac{\partial\varphi}{\partial\xi} \;\middle|\; \frac{\partial\varphi}{\partial\eta} \;\middle|\; \frac{\partial\varphi}{\partial\zeta} \right)$$
$$= \left(\mathbf{a}\eta\zeta + \mathbf{b}\eta + \mathbf{c}\zeta + \mathbf{e} \;\middle|\; \cdots \;\middle|\; \cdots \right)$$

- Transform the error vector $\Delta\mathbf{x} = \mathbf{x}' - \mathbf{x}$ into local coordinates:

$$(\Delta\xi,\Delta\eta,\Delta\zeta) = J^{-1}\Delta\mathbf{x}$$

- Apply the correction

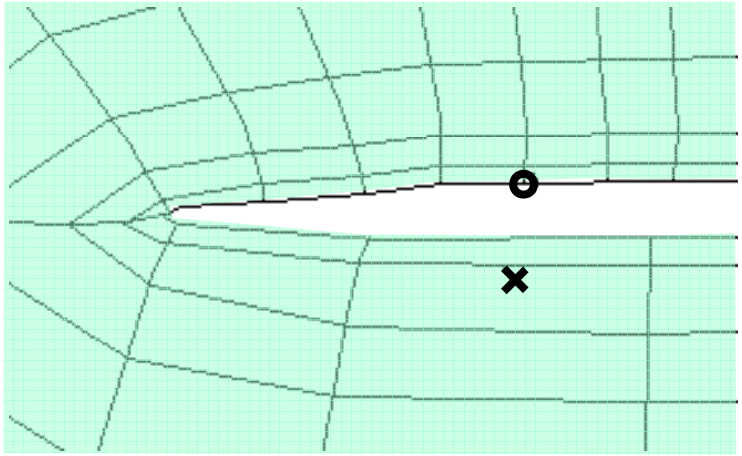
$$(\xi,\eta,\zeta)_{\text{new}} = (\xi,\eta,\zeta) + (\Delta\xi,\Delta\eta,\Delta\zeta)$$

Global point location

Global point location is more expensive. Many methods trade in safety for efficiency. A few methods are:

(1) Search for the point in every grid cell, using Newton's method. Hypothetic "brute force" method, safe.

(2) Buning's method: Do incremental point location starting from a boundary cell. Problem: Node (o) nearest to given point (x) is not necessarily adjacent to the cell containing it. Furthermore, the straight line between the two points can leave the grid.



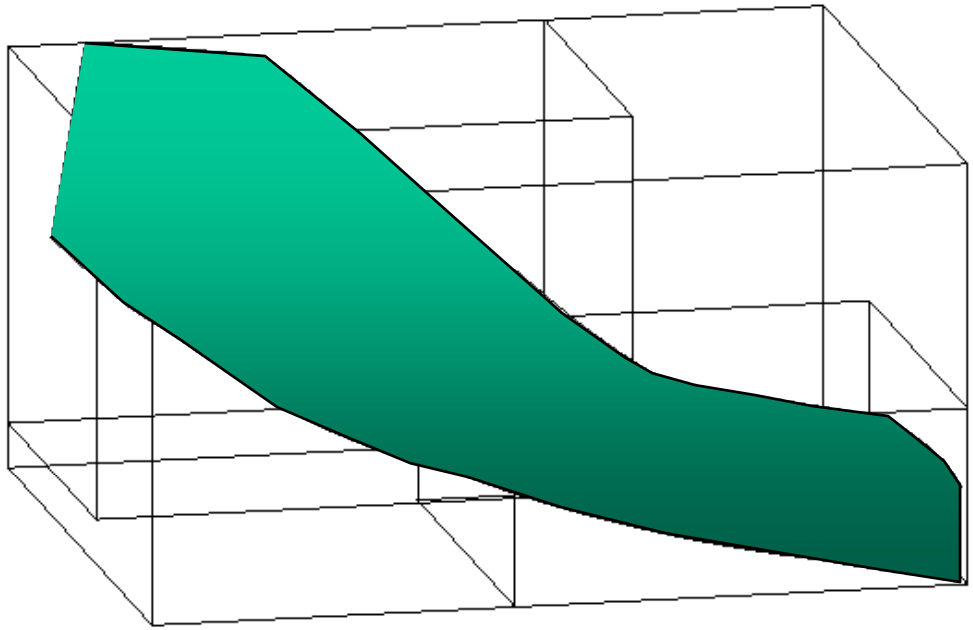
Buning's method is safe if incremental search is repeated with a different boundary cell as long as the point is not found. Instead of using all boundary cells, a subset can be precomputed which guarantees to find all points within the grid.

(3) Do incremental point location, starting a node near grid center. Simple method, safe only for star-shaped grids.

(4) Efficient methods use a search structure (uniform grid, octree, kd-tree) for nodes or cell centers:

- Point query not sufficient, need range query, with range determined by cell size.
- Problem: cells (especially from CFD) can have extreme aspect ratios.

(5) Use a bounding box hierarchy for recursively subdivided grid. Efficient and safe method. Easy for structured grids. More pre-processing required for unstructured grids.



Computational space streamline integration

In structured grids, point location can be avoided by using a different approach:

Integration can be done in computational space $\mathcal{C}$ instead of physical space $\mathcal{P}$.

Modification of the integration algorithm:

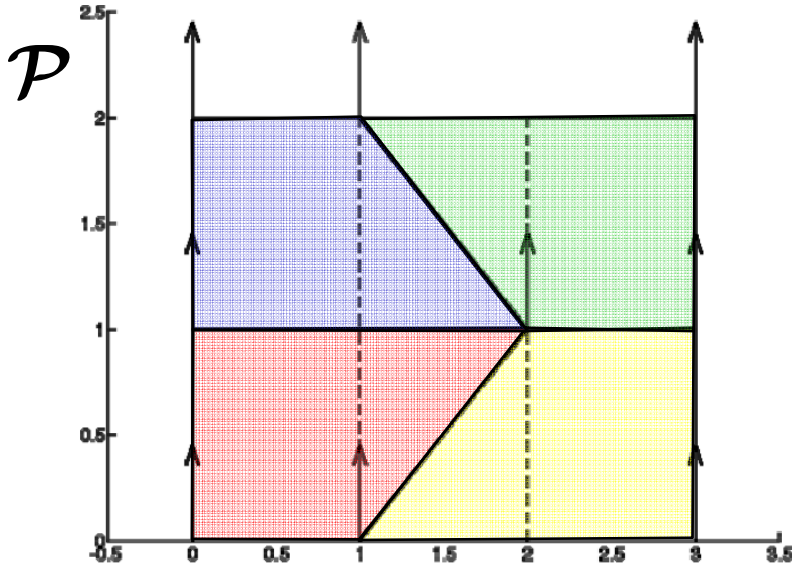
- before integration step:
  - Transform velocity $\mathbf{v}(\mathbf{x})$ to $\mathcal{C}$ by multiplying with $\mathbf{J}^{-1}$
- after integration step:

  (only if graphical output of this step is needed):
  - transform new position $\mathbf{x} = (i+\xi, j+\eta, k+\zeta)$ to $\mathcal{P}$ by trilinear interpolation.

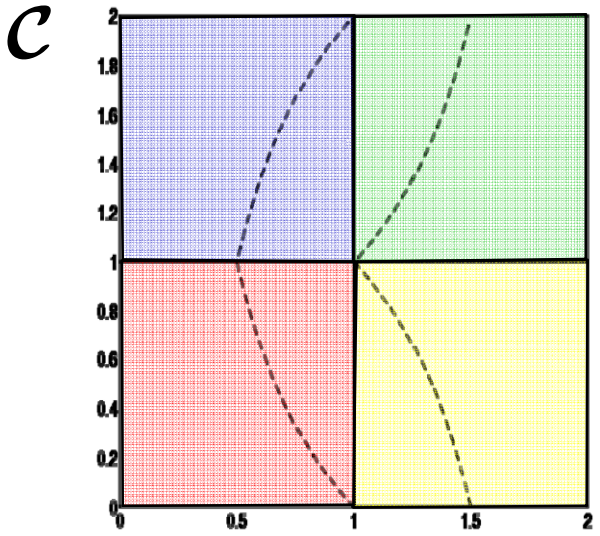
Main problem of integration in $\mathcal{C}$:

- cordinate function $\varphi : (i+\xi, j+\eta, k+\zeta) \mapsto (x, y, z)$ is only C^0 continuous at cell boundaries
- therefore **J** is discontinuous

Example (Sadarjoen 1994): Four cells with constant velocity field.



straight streamlines (dashed)



jagged streamlines (dashed)

Two sources of error:

- Integration steps across cell boundaries:
  - can be avoided by shortening such steps
- Use of a (precomputed) single transformed field vector per node:
  - can be fixed by transforming all eight vectors of a cell on the fly when entering a new cell.

The main advantage of integration in $\mathcal{C}$ is algorithmic simplicity.

If done correctly (avoiding above errors) it can be slower than integration in $\mathcal{P}$.

Skin friction lines

Velocity fields of fluid flow can have grid boundaries which are walls, i.e. solid material surfaces. At walls the velocity vector is usually zero as a result of the no-slip boundary condition.

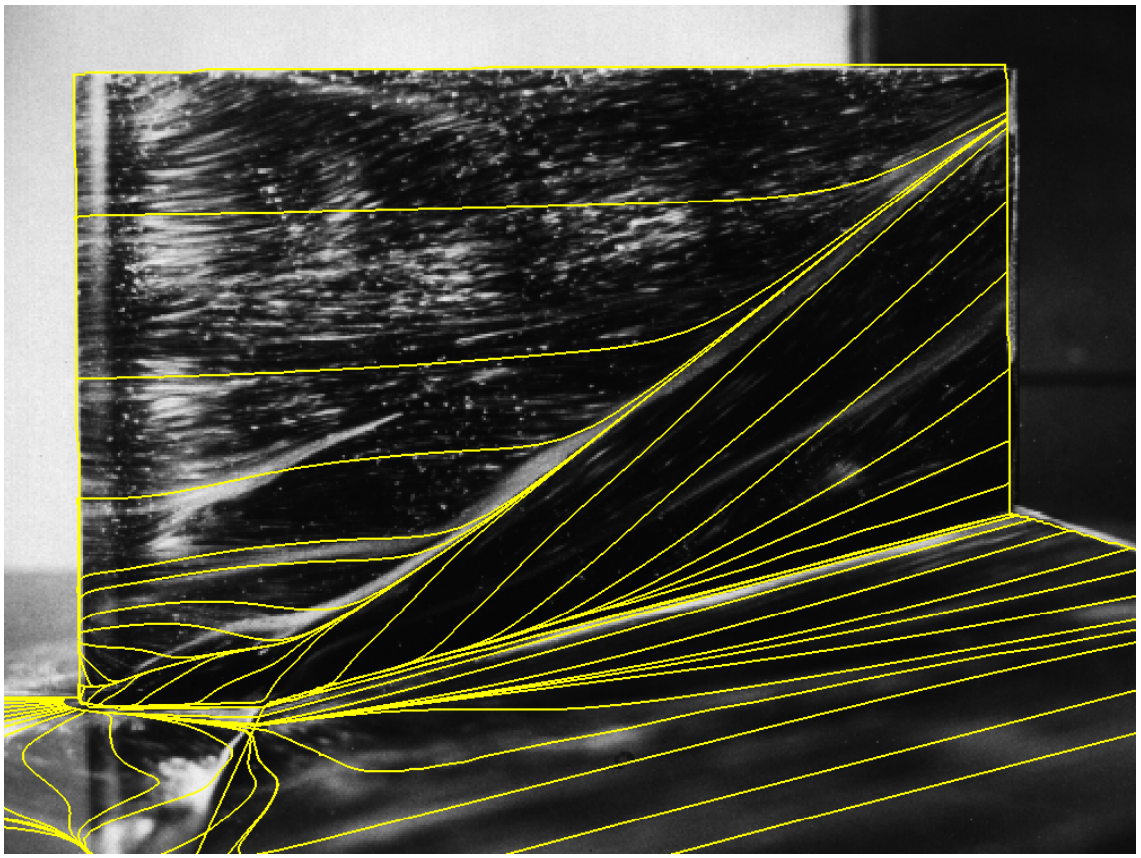
Therefore, a derived vector field is often used, namely the wall shear stress:

$$\tau_w = \mu \frac{\partial \mathbf{v}_w}{\partial s}$$

obtained as the limit of the wall-parallel velocity component $\mathbf{v}_w$ divided by the wall distance *s* and multiplied with the dynamic viscosity μ (a material constant). This limit is typically nonzero except at isolated points.

Streamlines of τ_w are called skin friction lines. They are an example of a vector field defined on a surface in 3-space.

Example (Pagendarm and Walter 1994): Skin friction lines from numerical simulation superposed over experimental oil-flow pattern.



Streamline placement

Problem of visualization by streamlines:

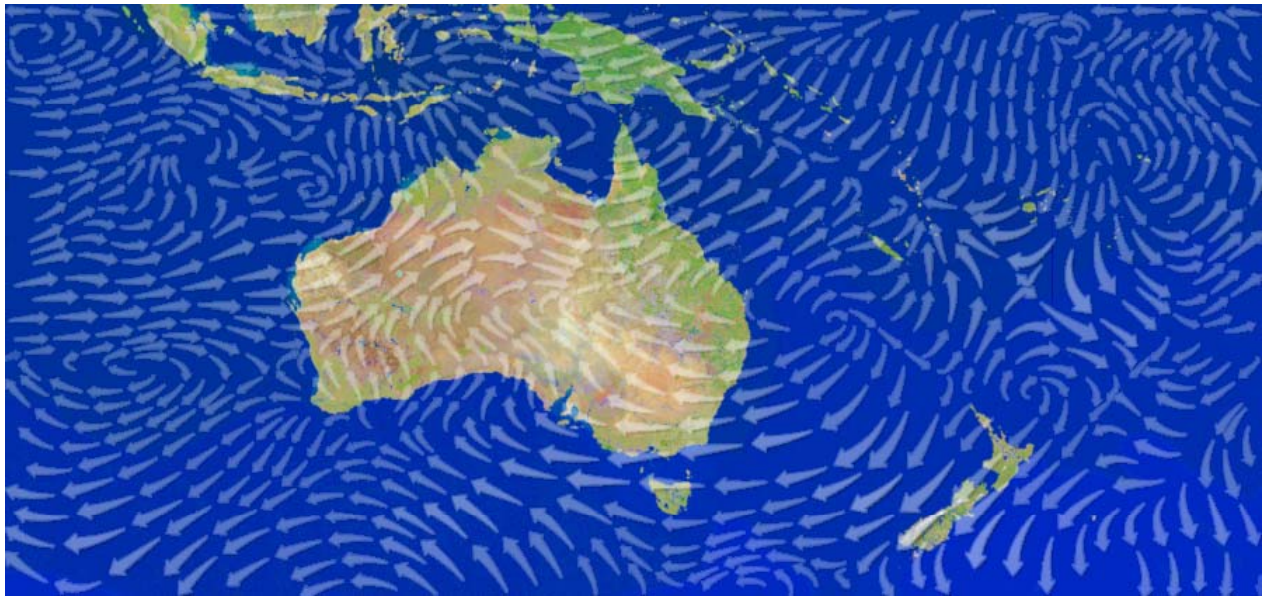
- dependency on seed points
- density of streamlines can be largely inhomogeneous

Solution: streamline placement, i.e. automatic, optimized choice of seed points.

Method 1: streamlets (short streamline segments)

- length is proportional to velocity magnitude (obtained automatically by using fixed integration time)
- start with uniform grid and make spacing roughly even by locally adapting (displacing, inserting, removing) seeds

Example of streamlets:

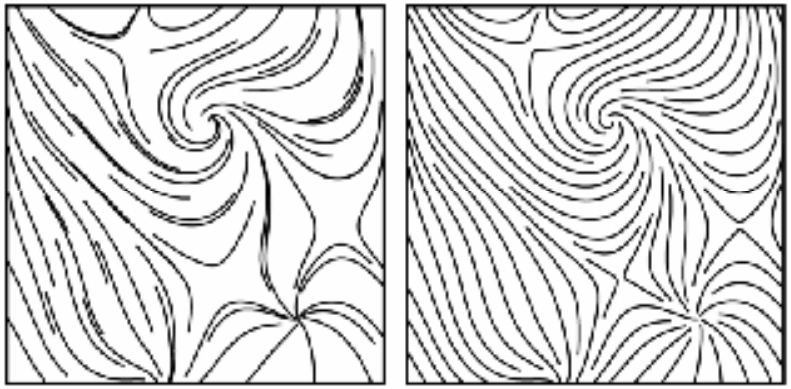


Discussion: Compared to arrows:

- additional curvature information
- no overlapping

Method 2, Algorithm by Turk and Banks (for longer streamlines):

- Objective: Create a streamline image which when low-pass filtered has a uniform grey level
- Optimize: seed positions and integration lengths
- Operations:
  - delete, insert, move, lengthen, shorten
- Apply operations either randomly or based on oracles.



Seeds on regular grid vs. Turk/Banks method

Streamsurfaces

Definition of a stream surface:

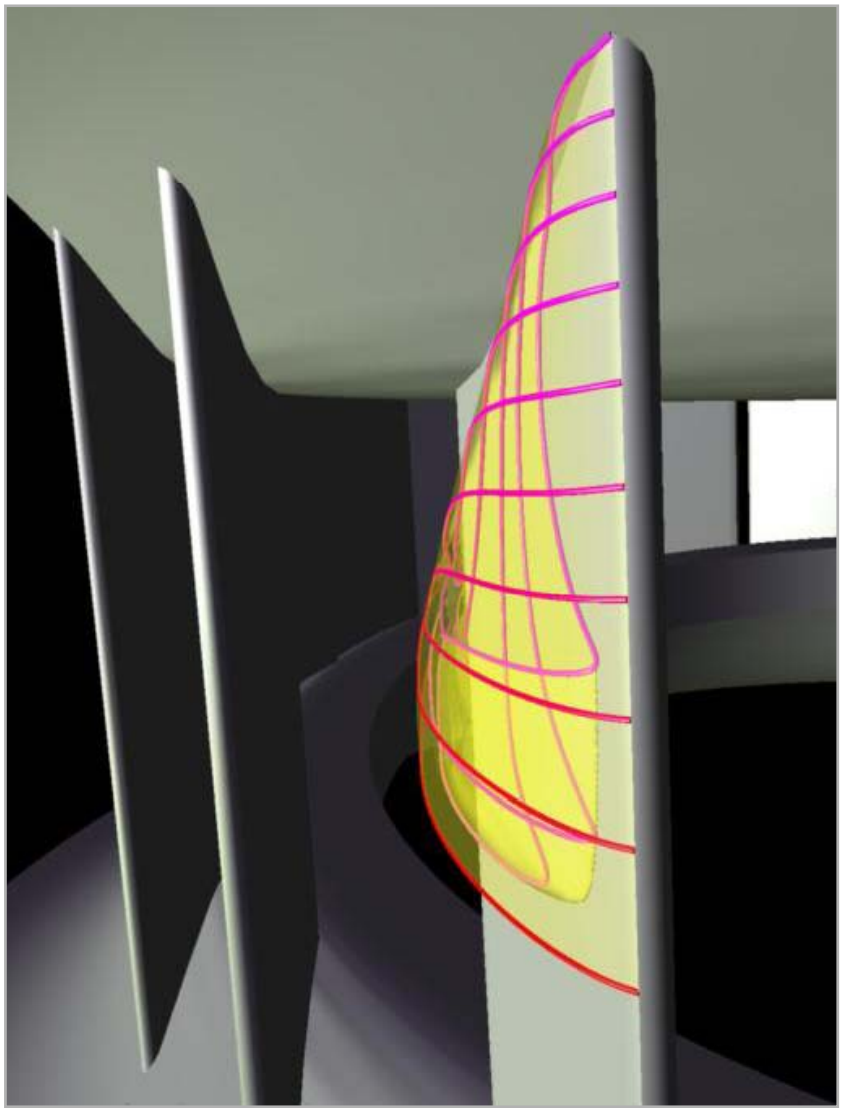
Union of streamlines seeded densely on a curve, e.g. straight line or circle.

Advantage for visualization:

more structured, better spatial perception.

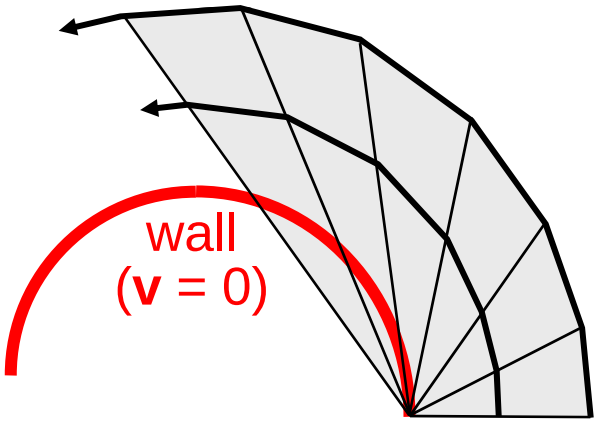
Naïve algorithm:

- start integration at discrete samples on the seed curve
- connect points of equal integration time, resulting in a quad mesh.

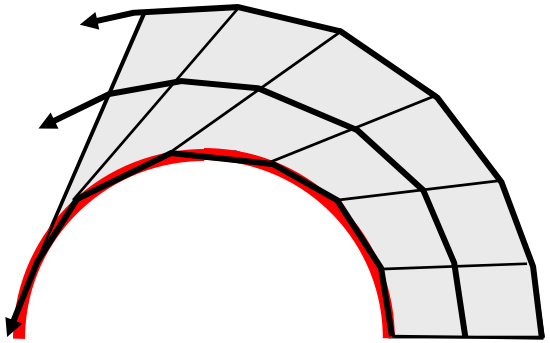


Problem: naïve algorithm fails if streamlines diverge or grow at largely different speeds.

Example of failure: seed curve which extends to no-slip boundary:



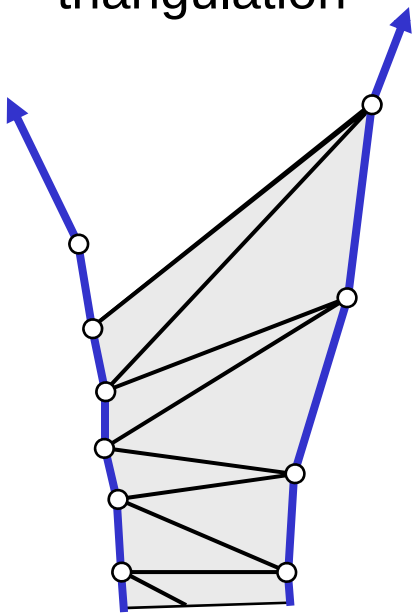
fixed time steps



fixed spatial steps
(slightly better)

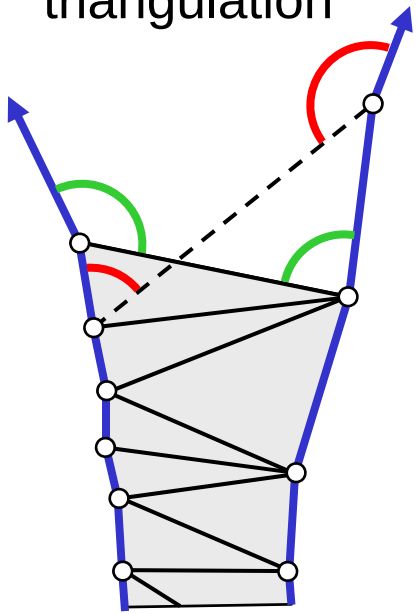
Hultquist's algorithm solves the problem of speed differences by optimized triangulation: Of two possible connections chose the one which is closer to orthogonal to both streamlines.

systematic triangulation



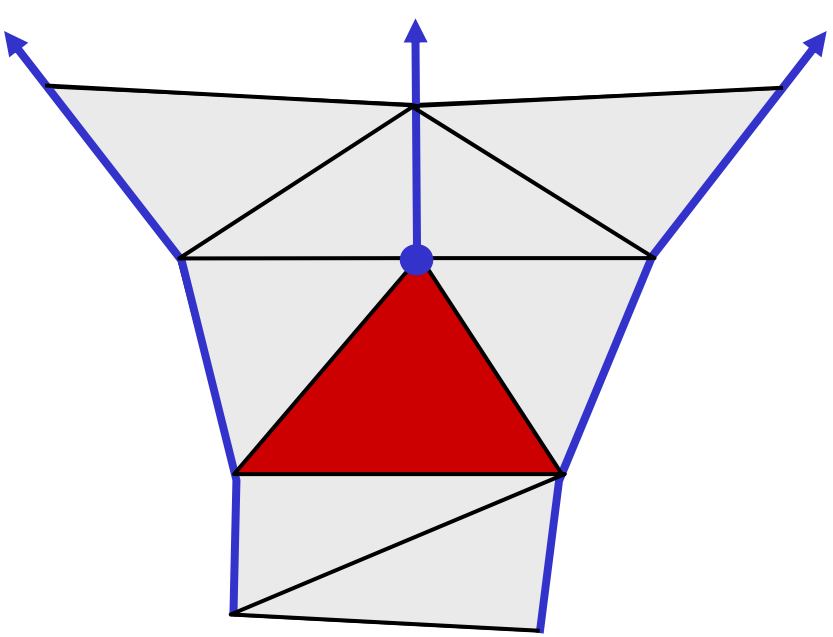
streamlines

optimized triangulation

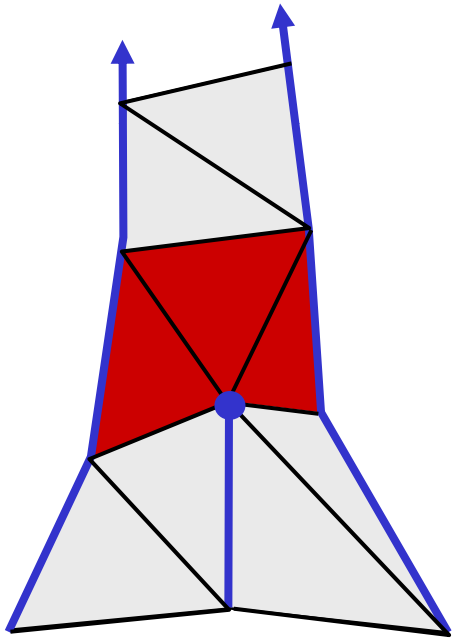


streamlines

The problem of divergence or convergence is solved by inserting or terminating streamlines.



inserted streamline



terminated streamline