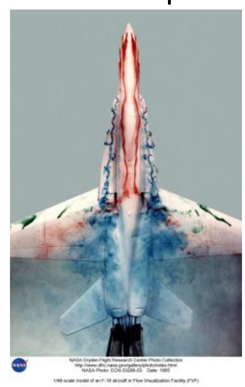
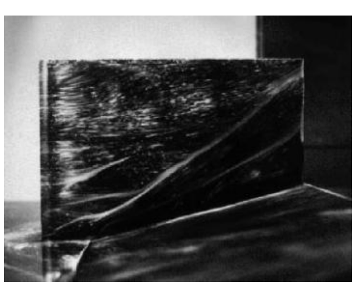
2D Vector Field Visualization

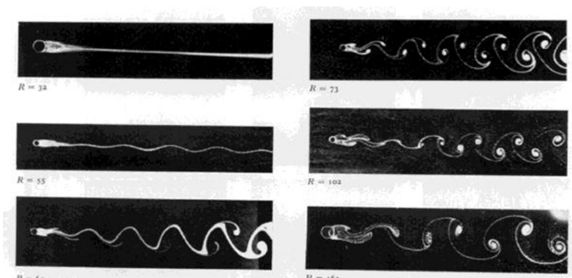
Experimental Flow Vis



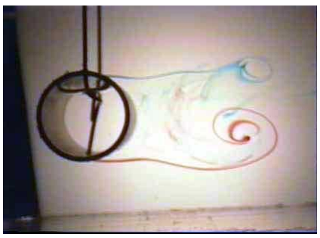
Experimental Flow Vis



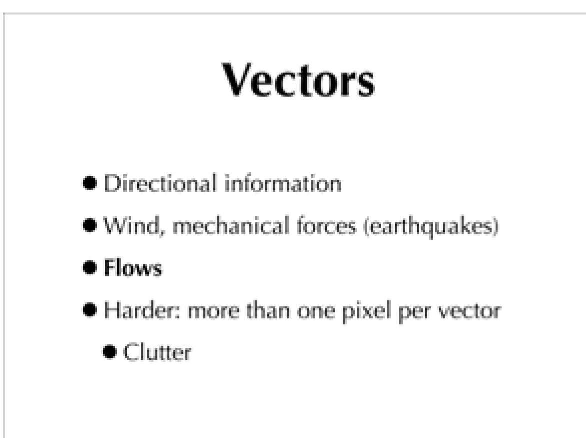
Experimental Flow Vis



Experimental Flow Vis



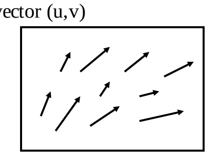
Why would we not stick with these?



Vector Field Visualization

-A vector field: F(U) = V

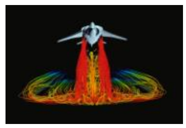
U: field domain (x,y) in 2D
V: vector (u,v)

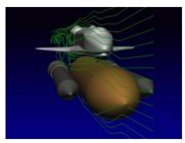


-Like scalar fields, vectors are defined at discrete points:
-interpolation issues

Visualization techniques

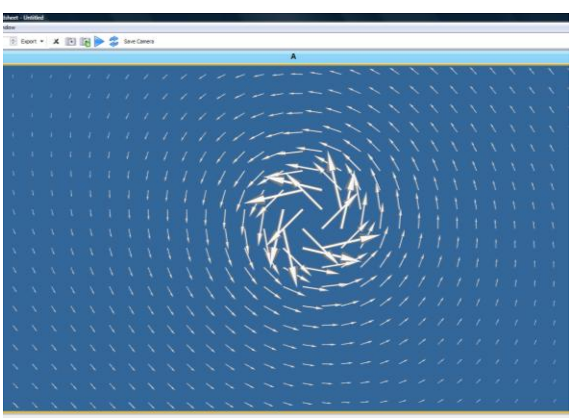
- Geometry-based methods: rendering primitives built from particle trajectories
  - Glyphs
  - streamlines
  - pathlines
  - streaklines
  - topology
  - LIC
  - …….

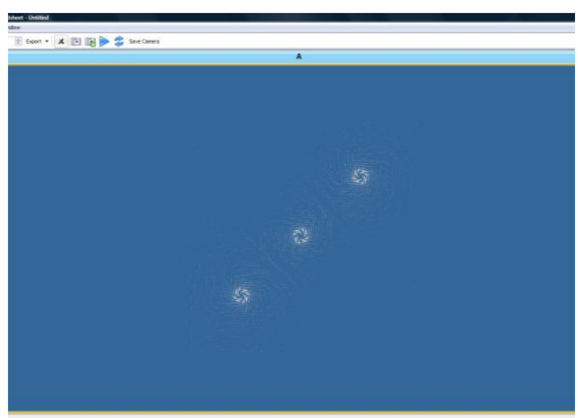


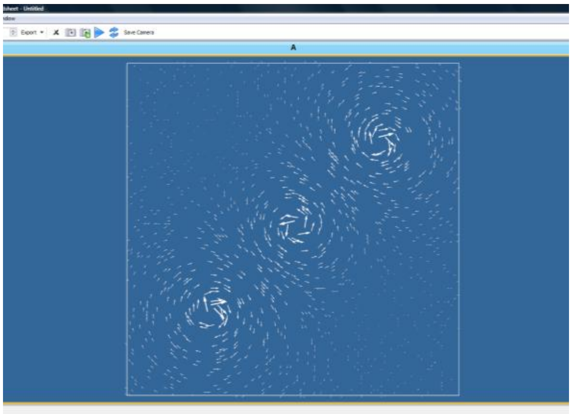


Glyphs

- Place symbols over vector field
  - Regularly spaced
  - Randomly spaced
  - Scale
- Watch out for clutter

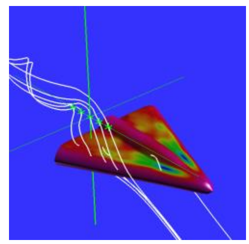


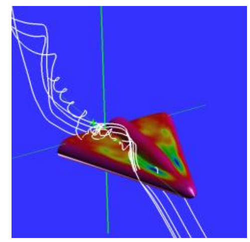


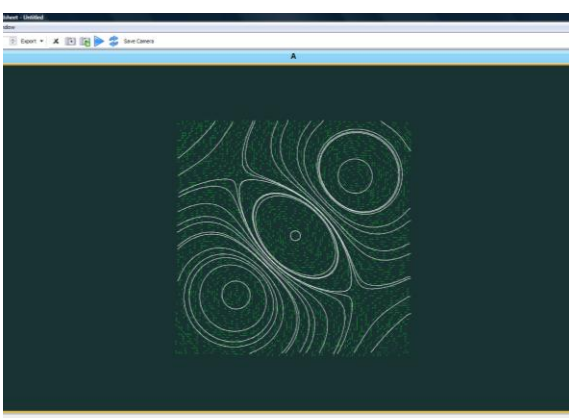


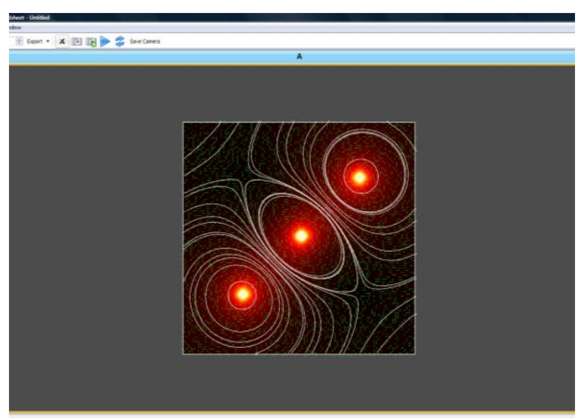
Streamlines

A curves that connect all the particle positions









Streamlines

- Lines that are everywhere tangent to the vector field
  - $f(0) = x_0,\ \dot{f}(x) = u(x)$
- That's a diff. eq.
- Solving for $f(x)$ is an **initial value problem**

Local technique - Particle Tracing

Visualizing the flow directions by releasing particles and calculating a series of particle positions based on the vector field

The motion of particle: dx/dt = v(x)

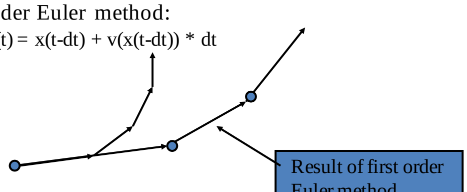
x: particle position (in 2D (x_1, x_2) position vector)
v(x): the vector (velocity) field

Use numerical integration to compute a new particle position

x(t) = x(t-dt) + Integration(v(x(t-dt)) dt)

Numerical Integration

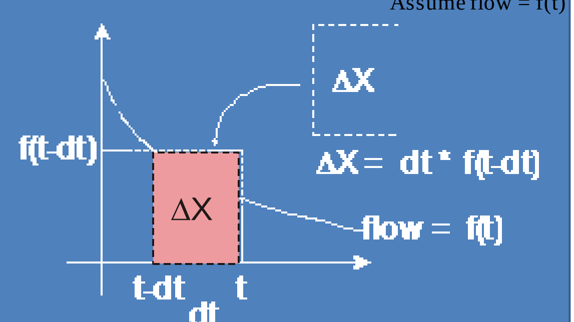
First Order Euler method:
x(t) = x(t-dt) + v(x(t-dt)) * dt



- Not very accurate, but fast
- Other higher order methods are avilable: Runge-Kutta second and fourth order integration methods (more popular due to their accuracy)

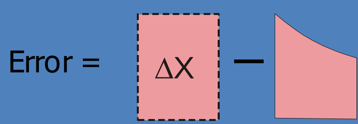
Euler's Method

Assume flow = f(t)



Euler Integration Error

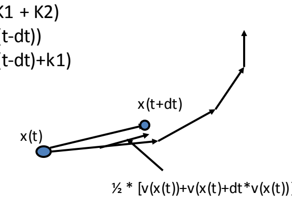
- Error = ΔX - area under flow curve



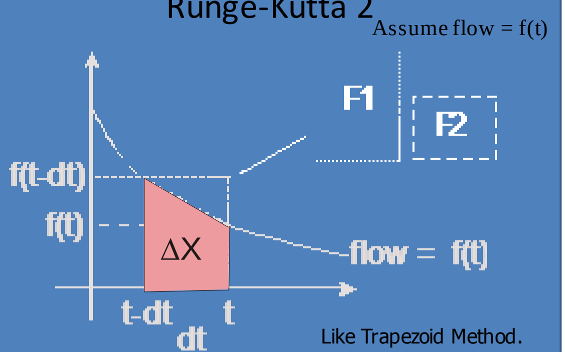
Numerical Integration (2)

Second Runge-Kutta Method

x(t) = x(t-dt) + ½ * (K1 + K2)
k1 = dt * v(x(t-dt))
k2 = dt * v(x(t-dt)+k1)



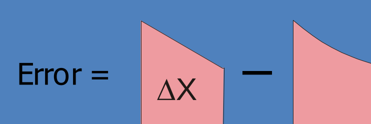
Runge-Kutta 2



Like Trapezoid Method.

RK2 Integration Error

- Error = ΔX - area under flow curve



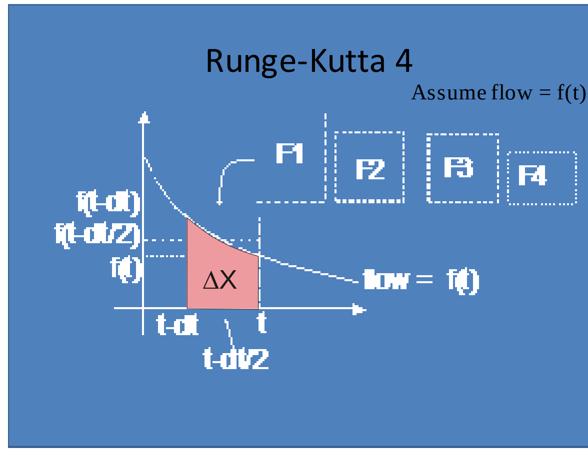
Numerical Integration (3)

Standard Method: Runge-Kutta fourth order

$$x(t) = x(t-dt) + 1/6\,(k_1 + 2k_2 + 2k_3 + k_4)$$

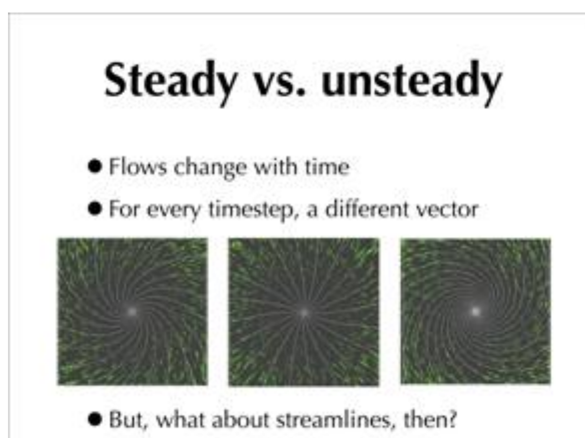
$k_1 = dt * v(t-dt)$; $k_2 = dt * v(x(t-dt) + k1/2)$

$k_3 = dt * v(x(t-dt) + k2/2)$; $k_4 = dt * v(x(t-dt) + k3)$



What Method to Use?

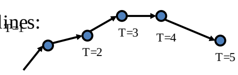
- RK2 and RK4 are <u>more accurate</u> for same dt than Euler
- Euler works poorly for <u>oscillatory</u> systems
- RK2 and RK4 work well for <u>continuous</u> systems
- RK2 and RK4 work poorly with <u>discrete</u> systems



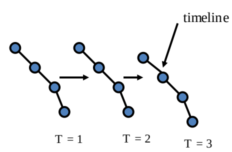
Pathlines, Timelines, and Streaklines

-Extension of streamlines for time-varying data

Pathlines:

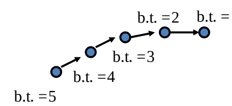


Timelines:

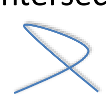


Streaklines

- Continuously injecting a new particle at each time step, advecting all the existing particles and connect them together into a *streakline*



When should we expect self-intersections?



- Streamlines
- Pathlines
- Streaklines

Pathlines and Streaklines

- Streamlines do not cross
- Streaklines still never cross
- Pathlines do cross

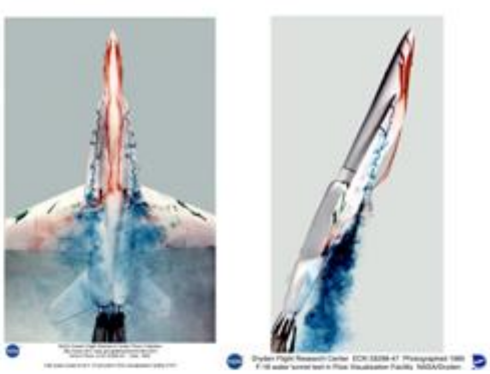
Pathlines and Streaklines

- Streamlines **should** not cross
- Streaklines still **seldom** cross
- Pathlines do cross

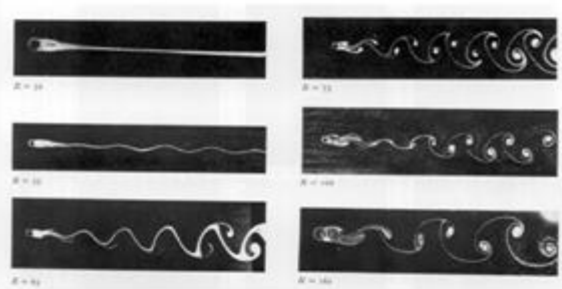
Seed Placement

- The placement of seeds directly determines the visualization quality
  - Too many: scene cluttering
  - Too little: no pattern formed
- It has to be the right number at the right places!!!

Streaklines in real life



Streaklines in real life



Line Integral Convolution

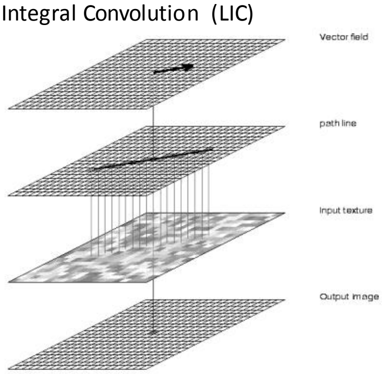
- Basic idea: Integrate noise along streamlines
- demo: http://www.javaview.de/demo/PaLIC.html

Rendering - LIC

- embed a noise texture under the vector field
- integrates along a streamline

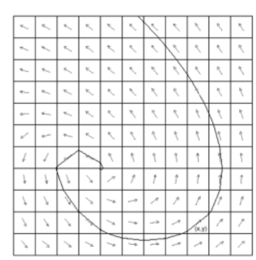
Line Integral Convolution (LIC)

- LIC:
- convolve a
- random texture
- along the
- streamlines

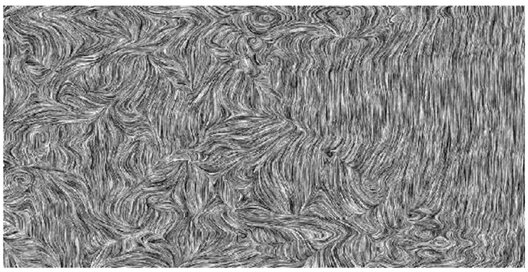


Line Integral Convolution (LIC)

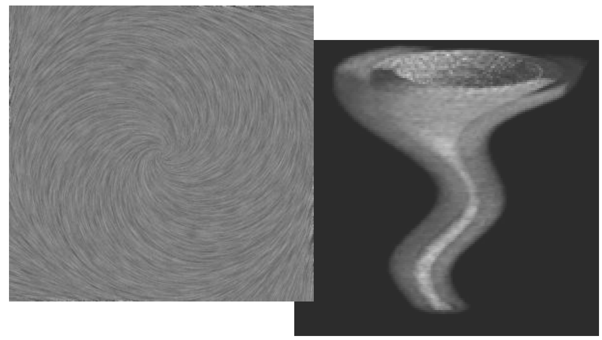
- Assume input texture, vector and output images are all the same resolution.
- For each output pixel/voxel, generate a streamline both forwards and backwards of a fixed length.
- Integrate the intensity that the streamline passes through



Line Integral Convolution (LIC)



Line Integral Convolution (LIC)



Comparison (LIC and Streamlines)