# Efficient Fluid Simulation Algorithms

Josh Slocum

# What’s so hard about making realistic fluids?

- Realistic rendering
  - Color
  - Transparency
  - Reflection and refraction

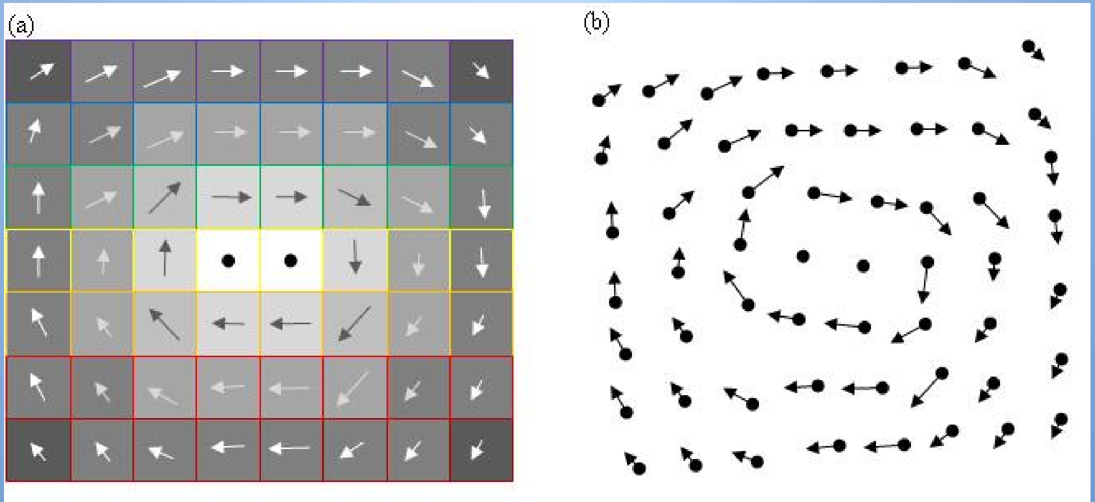
- Realistic physics
  - Foam and spray
  - Small rolling motions
  - Interference
  - Surface tension

# Simulation Comparison

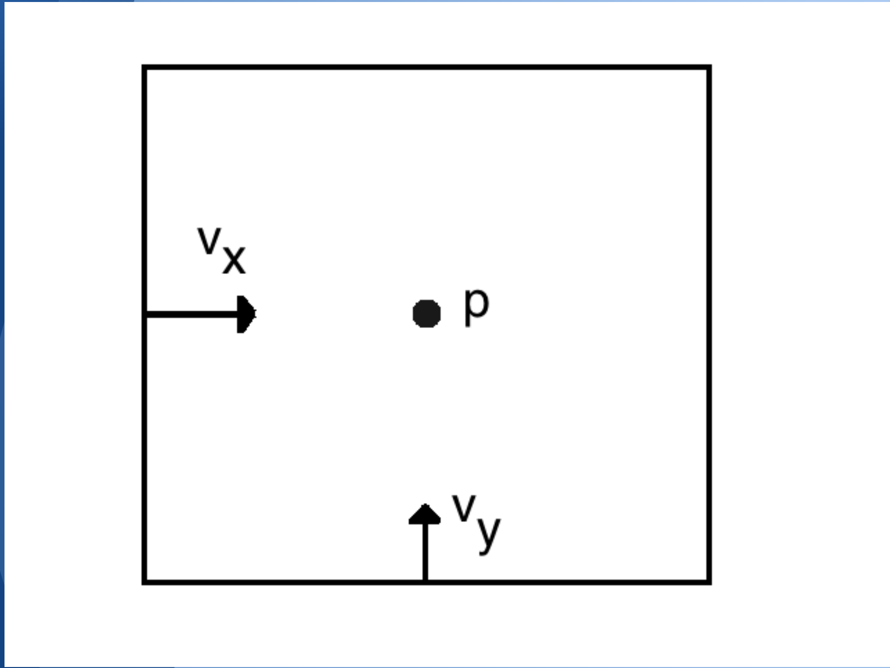
# Models of Computation

Eulerian (grid based), properties of fluid at each location

Lagrangian (particle based), each particle represents fluid



# Eulerian Methods: Defining the grid




MAC discretization

- pressure at centers
- velocity at faces
  - only stores perpendicular component

# Velocity Field

Defines the fluid and its motion, pressure and other values just used to properly construct velocity field.

# Defining the Fluid

$$\frac{\delta v}{\delta t} = -v \cdot \nabla v$$

Advection: Fluid continues to move forward in time

# Defining the Fluid

$$\frac{\delta v}{\delta t} = -v \cdot \nabla v + f$$

External forces (such as gravity)

# Defining the Fluid

$$\frac{\delta v}{\delta t} = -v \cdot \nabla v - \nabla p + f$$

$$\nabla \cdot v = 0$$

Incompressibility and pressure (internal forces)

# Incompressible Navier-Stokes Equations

$$\frac{\delta v}{\delta t} = -v \cdot \nabla v - \nabla p + f$$

$$\nabla \cdot v = 0$$

# Incompressible Navier-Stokes Equations

$$\frac{\delta v}{\delta t} = -v \cdot \nabla v - \nabla p + f$$

velocity

$$\nabla \cdot v = 0$$

# Incompressible Navier-Stokes Equations

$$\frac{\delta v}{\delta t} = -v \cdot \nabla v - \nabla p + f$$

time

$$\nabla \cdot v = 0$$

# Incompressible Navier-Stokes Equations

$$\frac{\delta v}{\delta t} = -v \cdot \nabla v - \nabla p + f$$

pressure

$$\nabla \cdot v = 0$$

# Incompressible Navier-Stokes Equations

$$\frac{\delta v}{\delta t} = -v \cdot \nabla v - \nabla p + f$$

external forces

$$\nabla \cdot v = 0$$

# Eulerian Implicit Solver

Each timestep:

1. Advect the velocity field by dt
2. Solve for the pressure
3. Project the velocity with the pressure

$$\frac{\delta v}{\delta t} = -v \cdot \nabla v - \nabla p + f \quad \nabla \cdot v = 0$$

# Eulerian Implicit Solver

Each timestep:

1. Advect the velocity field by dt
2. Solve for the pressure
3. Project the velocity with the pressure

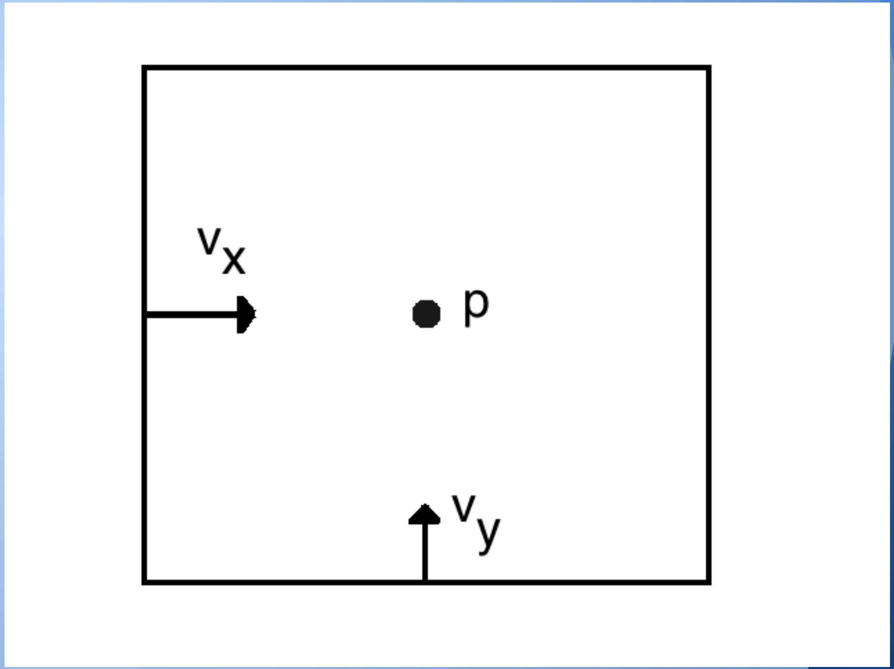
$$\frac{\delta v}{\delta t} = \underline{-v \cdot \nabla v} - \nabla p + f \quad \nabla \cdot v = 0$$

# 1. Velocity Advection

Moving the fluid forward in time by dt.

# 1. Velocity Advection - MAC

In the MAC method, we only store one of the velocity components at the face.

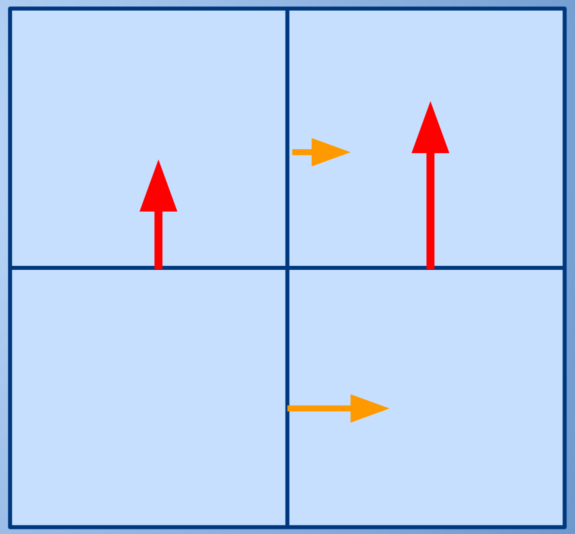


# 1. Velocity Advection - MAC

1. Compute Velocity at Corners
2. Average corners to face

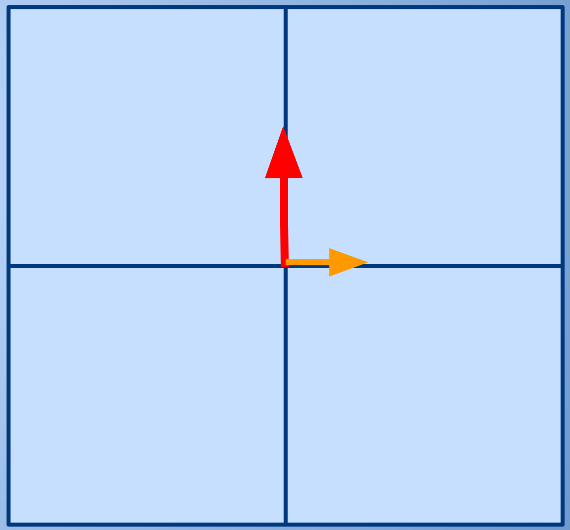
# 1. Velocity Advection - MAC

1. Compute Velocity at Corners
2. Average corners to face



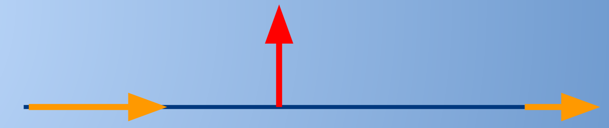
# 1. Velocity Advection - MAC

1. Compute Velocity at Corners
2. Average corners to face



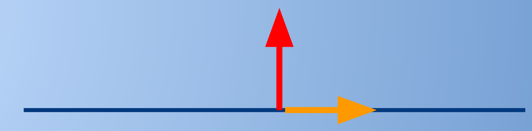
# 1. Velocity Advection - MAC

1. Compute Velocity at Corners
2. Average corners to face



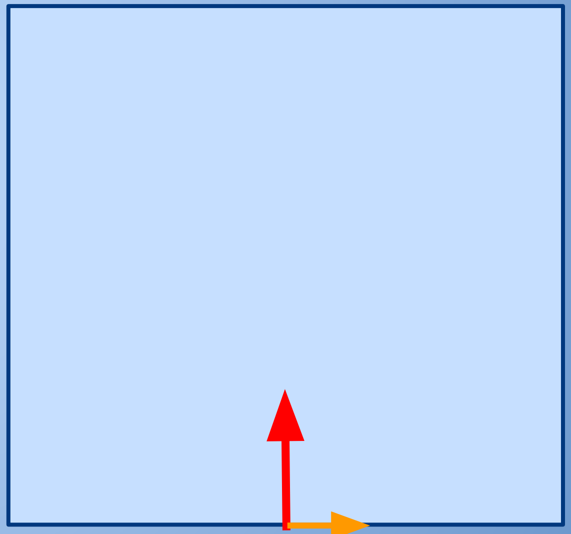
# 1. Velocity Advection - MAC

1. Compute Velocity at Corners
2. Average corners to face



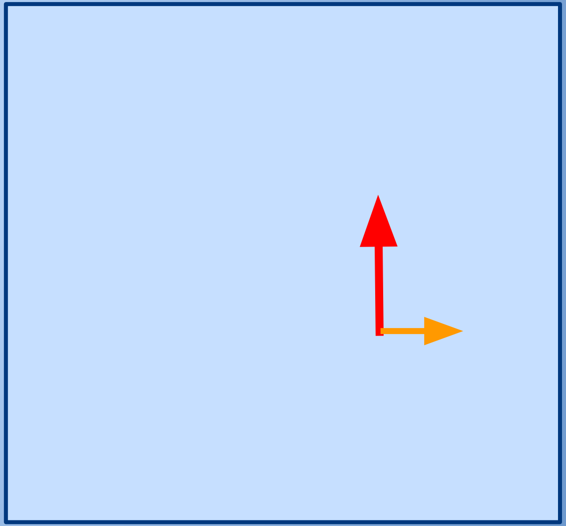
# 1. Velocity Advection - Simplest method

Move velocity v to new position and set position's velocity to v



# 1. Velocity Advection - Simplest method

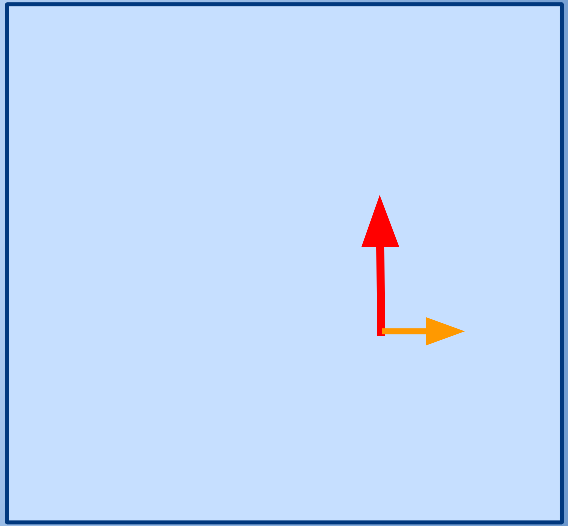
Move velocity v to new position and set position’s velocity to v



# 1. Velocity Advection - Simplest method

Move velocity v to new position and set position’s velocity to v

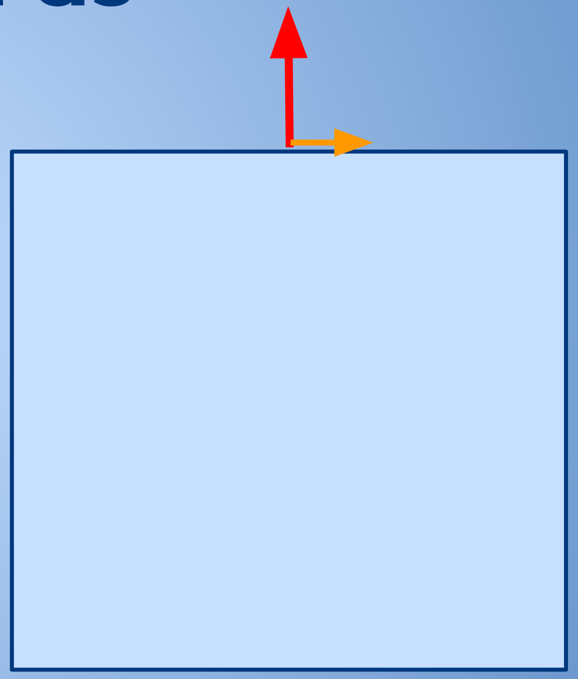
But, what face velocity do we update?



# 1. Velocity Advection - Working backwards

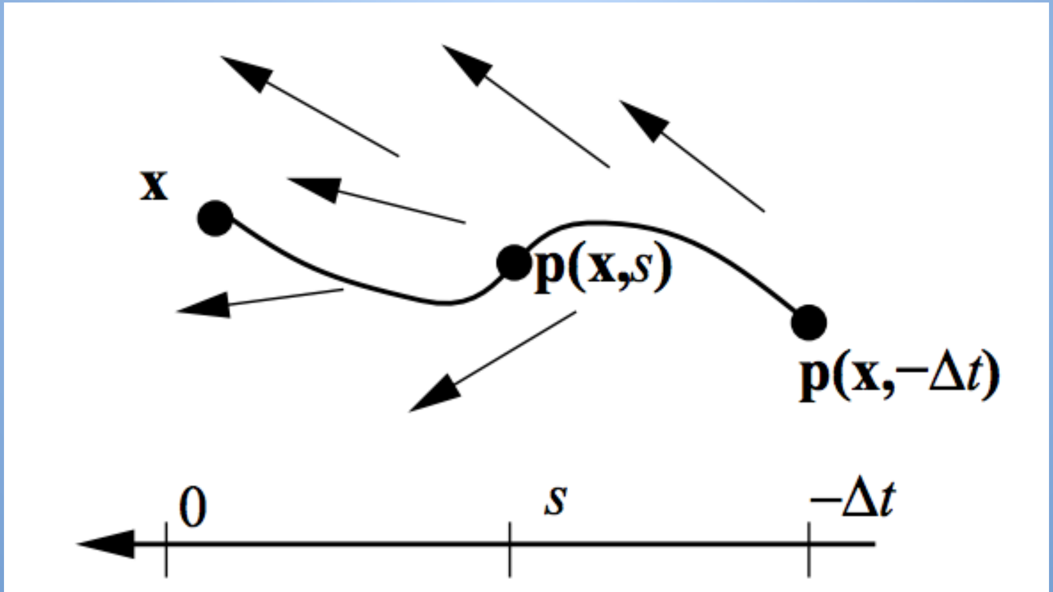
Want to set velocity at known place:

1. Go back in time
2. Figure out what velocity was there
3. Set face velocity to that



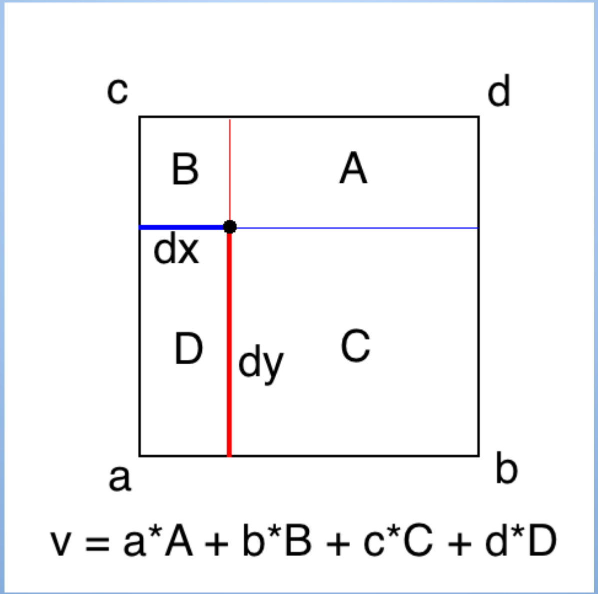
# 1. Velocity Advection - Semi-Lagrangian Advection

Use particle tracing algorithm to trace particle back from face.



# 1. Velocity Advection - Semi-Lagrangian Advection

Since point is inside cell, interpolate from corner velocities we know



# Eulerian Implicit Solver

Each timestep:

1. Advect the velocity field by dt
2. Solve for the pressure
3. Project the velocity with the pressure

$$\frac{\delta v}{\delta t} = -v \cdot \nabla v - \nabla \underline{p} + f \quad \nabla \cdot v = 0$$

# 2. Pressure Solver

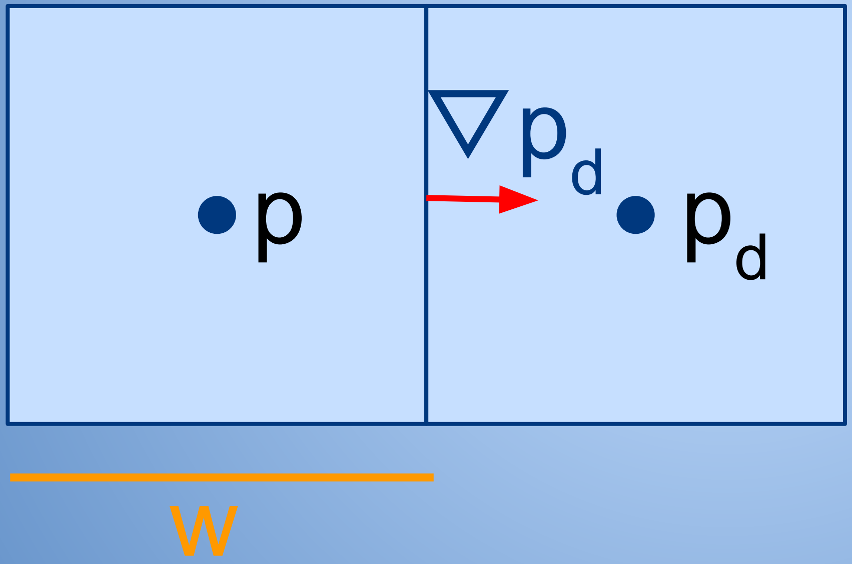
- Need the pressure for the second term
- With some manipulation, we can solve for the pressure with the equation $\nabla \cdot V = \nabla^2 p$.
- Form of the poisson equation $\nabla^2 \varphi = f$.

# 2. Pressure Solver - Discretizing the Equation

Using the divergence theorem on the previous equation:

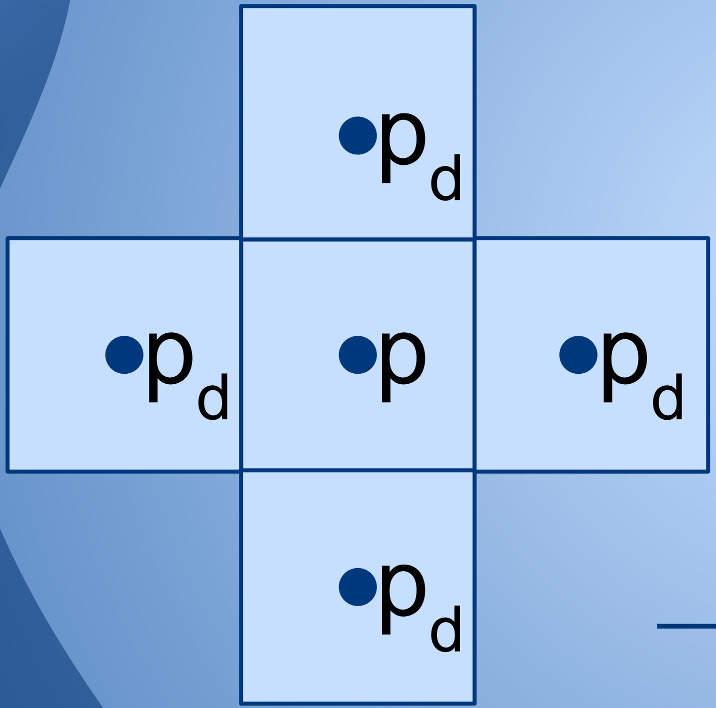
$$\nabla^2 p = \nabla \cdot V \rightarrow \sum_d \nabla p_d = h(\nabla \cdot V)$$

# 2. Pressure Solver - Calculating Pressure Gradient




$$\nabla p_d = (p_d - p)/w$$

# 2. Pressure Solver - Calculating Laplacian

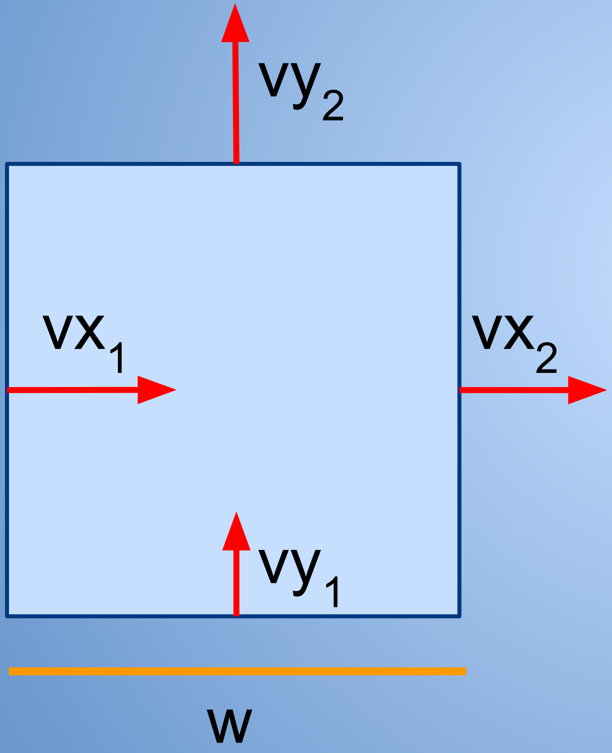



$$\sum_d \nabla p_d = h(\nabla \cdot V)$$

$$+ \quad \nabla p_d = (p_d - p)/w$$

$$\rightarrow \sum_d ((p_d - p)/w) = w(\nabla \cdot v)$$

# 2. Pressure Solver - Calculating Velocity Divergence




$$\sum_d \nabla p_d = h(\nabla \cdot V)$$

$\nabla \cdot v$ = dv/dx + dv/dy

= (vx2-xv1+vy2-vy1)/w

# 2. Pressure Solver - Solving the Equations

# 2. Pressure Solver - Sparse Linear Solver

- System of n equations and n unknowns
- Formulate as matrix multiply A * x = b
  - Many different algorithms to solve this
  - Use different properties of A to decompose it efficiently

# 2. Pressure Solver - Sparse Linear Solver

*Advantages*:

- Solves problem to machine precision
- Many libraries for it

*Disadvantages*:

- Performance dependent entirely on library and how you phrase the computation

# 2. Pressure Solver - Relaxation

$$\sum((p_d - p)/w) = w(\nabla \cdot v)$$

What if we knew all $p_d$ but not $p$?

# 2. Pressure Solver - Relaxation

$\sum((p_d - p)/w) = w(\nabla \cdot v)$

What if we knew all $p_d$ but not p?

→ Solve for p

$p = (\sum p_d - w^2(\nabla \cdot v))/4$

# 2. Pressure Solver - Relaxation

But, we don’t actually know any of the pressure values..

1. guess
2. repeatedly perform this assignment for each value until
3. both sides of original equation are “close enough” at each cell

new equation:

$p = (\sum p_d - w^2(\nabla \cdot v))/4$

original equation:

$\sum((p_d - p)/w) = w(\nabla \cdot v)$

# 2. Pressure Solver - Relaxation

*Advantages*:

- Simple, works well with varying topologies of grid
- Easy to implement, no libraries required

*Disadvantages*:

- Slower than Sparse Linear Solvers naively

# Eulerian Implicit Solver

Each timestep:

1. Advect the velocity field by dt
2. Solve for the pressure
3. Project the velocity with the pressure

$$\frac{\delta v}{\delta t} = -v \cdot \nabla v - \underline{\nabla p} + f \qquad \underline{\nabla \cdot v = 0}$$

# 3. Projection Method

Project the velocity field onto the space of divergence-free velocity fields.

Add the $-\nabla p$ term from the equation.

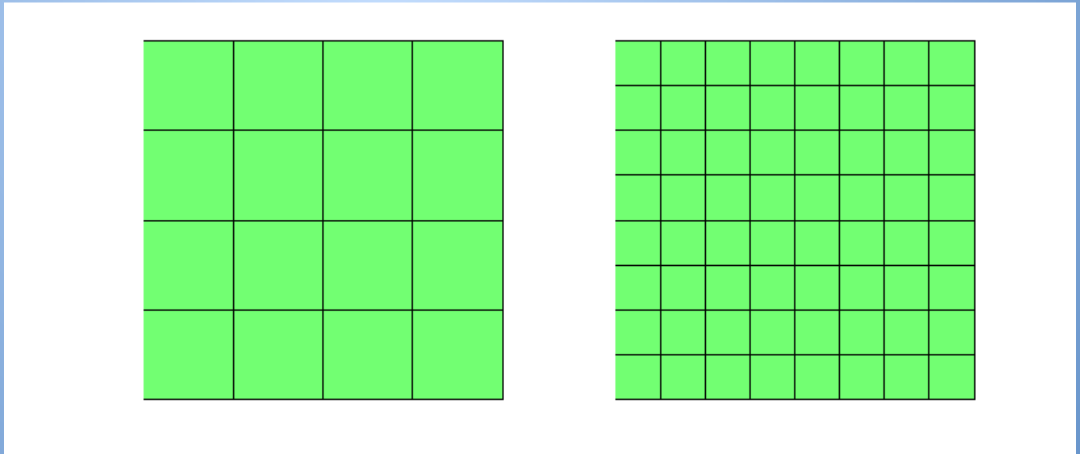
Since the velocity is at the faces and we already know how to calculate the pressure gradient at the faces, this is trivial.

# Performance of Eulerian Simulations

- Pressure solver is by far the most expensive step.
- Relaxation takes a while to converge on the final value.
- If we can get a better “initial guess”, we can perform fewer relaxations, and thus have greater performance.

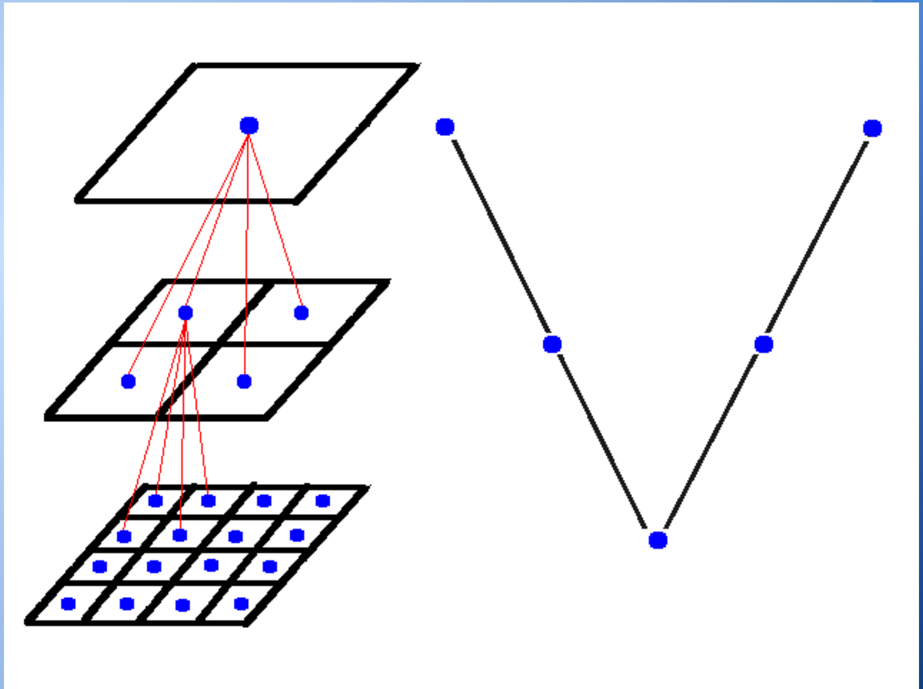
# Multigrid

- Optimization over uniform grids
- Relaxation starts with arbitrary initial guess
    - relax from initial guess on smaller grid
    - use solution to smaller grid as guess for bigger grid



# Multigrid V-cycle

- Multigrid can have more than 2 grids
- Use solution for each grid as guess for the next grid
- Average solutions back up from largest to smallest



# Problems of Eulerian Simulations

To get the high levels of computational resolution required, the grid must be very fine.

How can this be improved on?

# Adaptive Frameworks

# Adaptive Multigrid

- Increase the discretization of a computation only in certain areas
- Define refined “patches” that are even finer than the original grid
- Refinement can use any heuristic

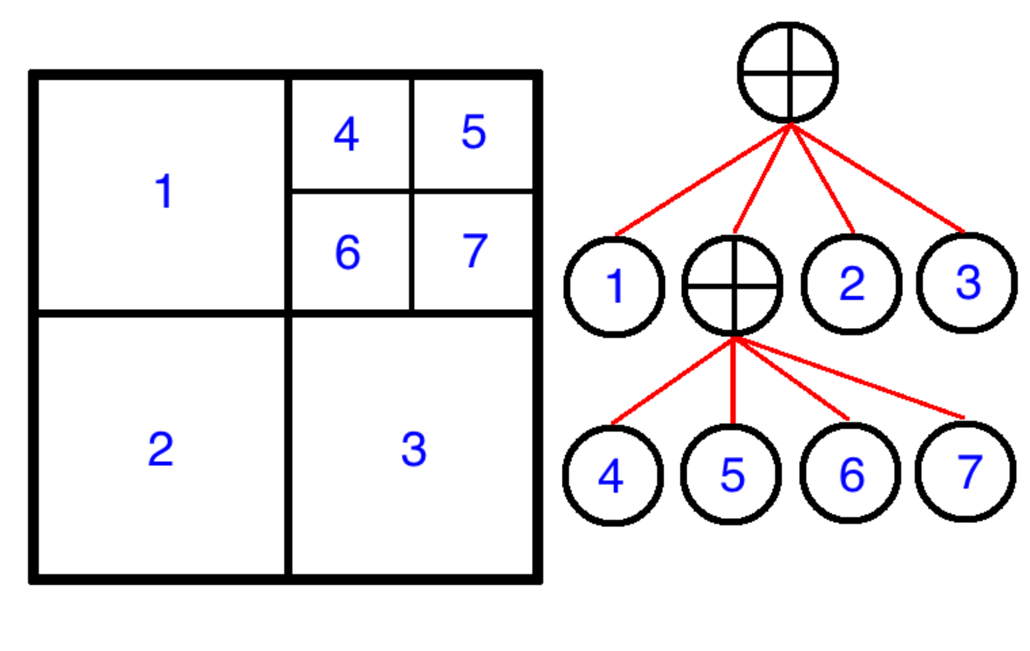
# Adaptive multigrid

Keeping track of refined patches is either expensive and complicated, or wastes a ton of memory.

What hierarchical data structure can we use?

# Adaptive quadtree

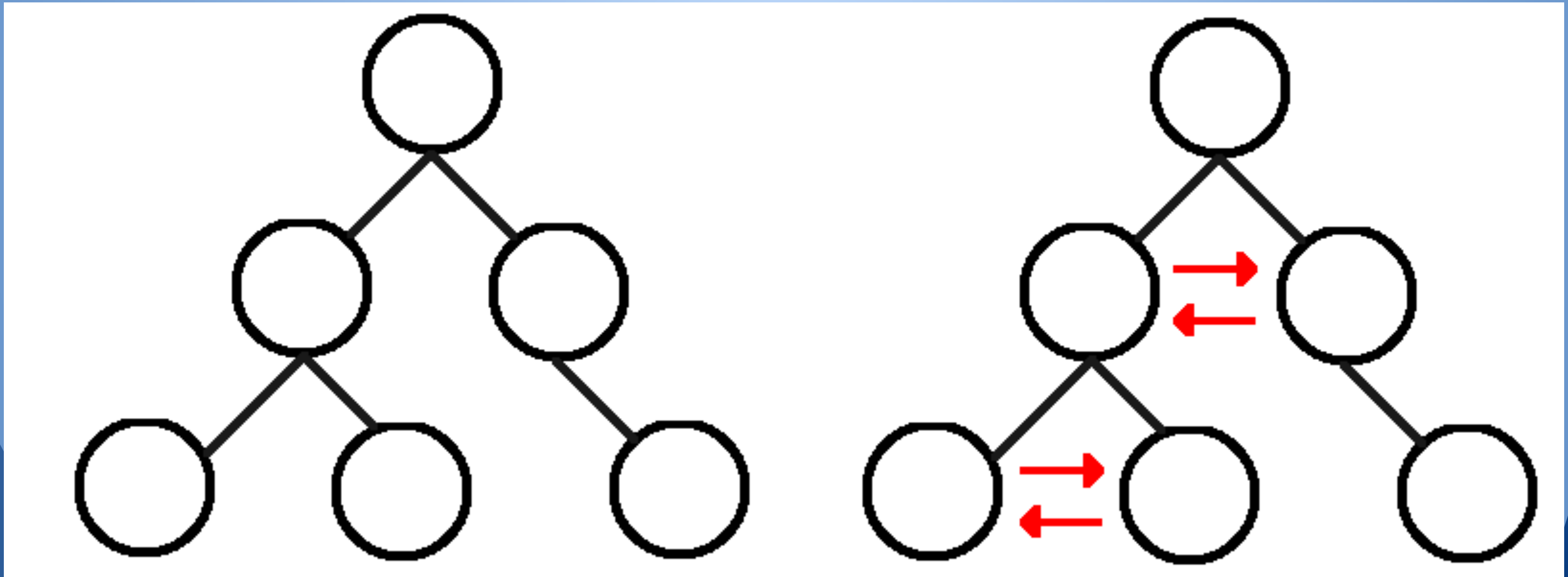
- Same data, different structure
- Finding neighboring cells more complicated
- Physics is harder to discretize



# Fully Threaded Neighbor Optimization

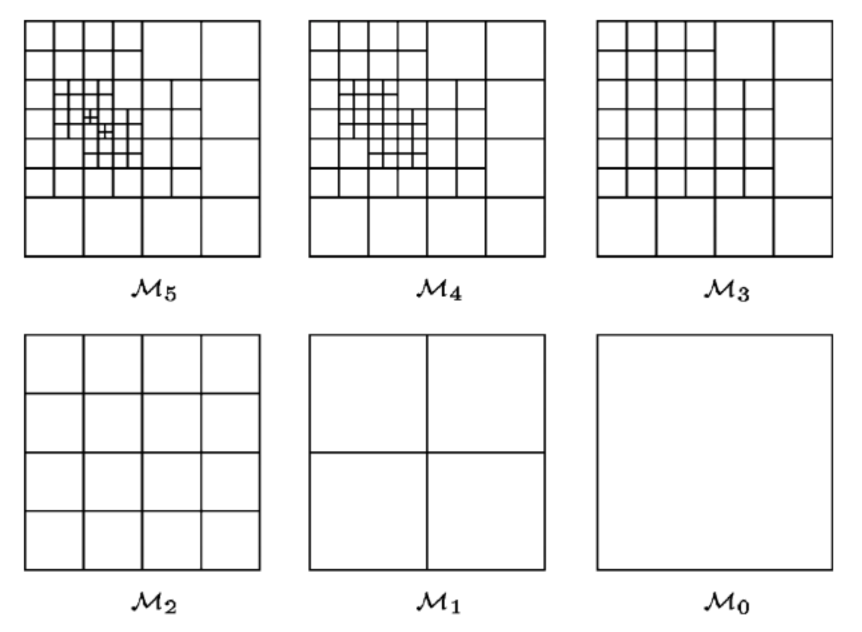
Non-threaded

Threaded



# Multilevels

The "finest" grid consists of the union of leaves in the tree.
The leaves are not all at the same level,
so we must define a "multilevel" as the following:




ML(L) = {n | n.leaf ^ n.level < L} U {n | n.level = L}

# Current State of the Art

What can we improve on?

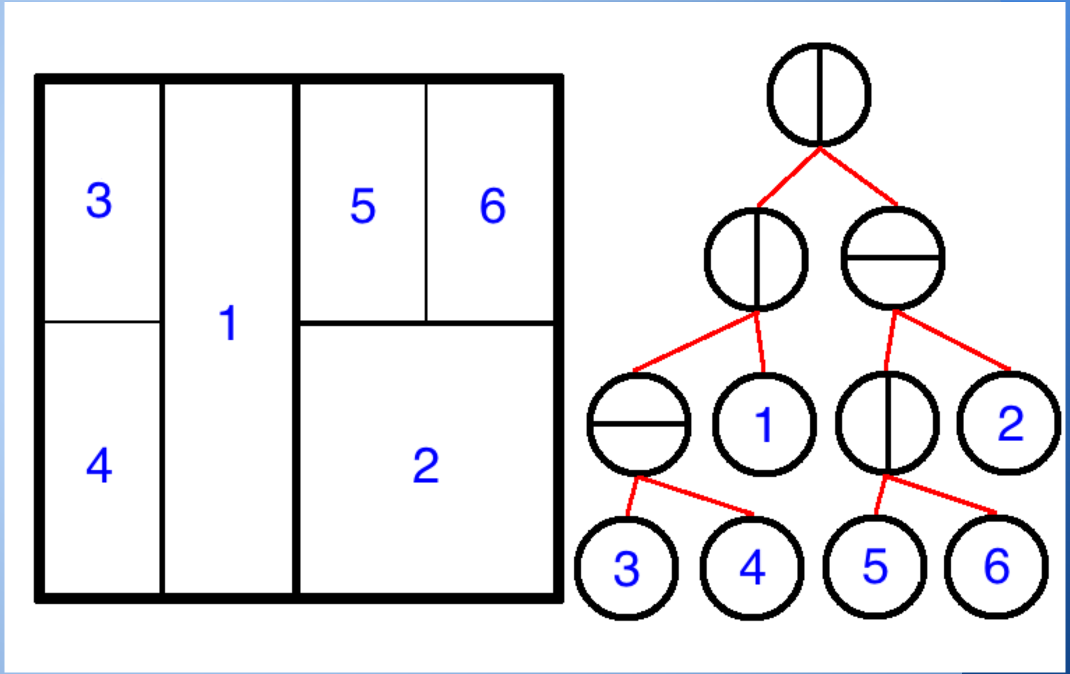
# Current State of the Art

What can we improve on?

Key insight: Fluids are very non-uniform
We have been discretizing them in a uniform way.

# K-d Tree

- K-d trees used in many other fields for non-uniform hierarchical algorithms



# My work

Built an adaptive quadtree and comparing it to an adaptive K-d tree.

Used Semi-Lagrangian advection and a multilevel relaxation solver, with the MAC discretization, but abstracted to work with non-uniformity.

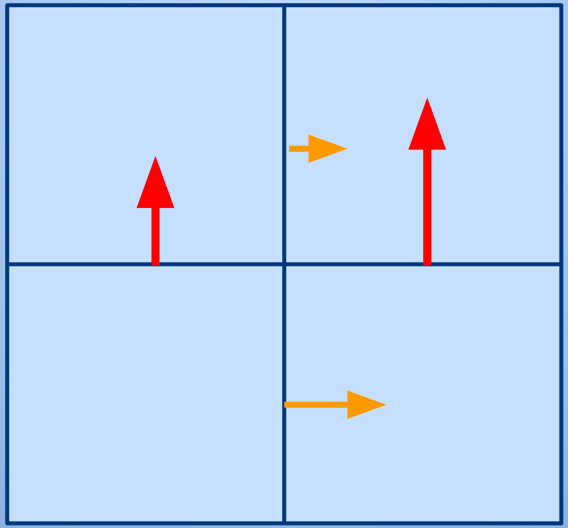
# Implementation

- Implemented similarly to perform useful comparisons
    - Abstracted quadtree methods to work on k-d tree and implemented both
- Main Large differences
    - Adaptivity
    - Computing values to feed to equations

# Common Elements: Advection

Still want to use same algorithm as before, but averaging face velocities to nodes is much harder.

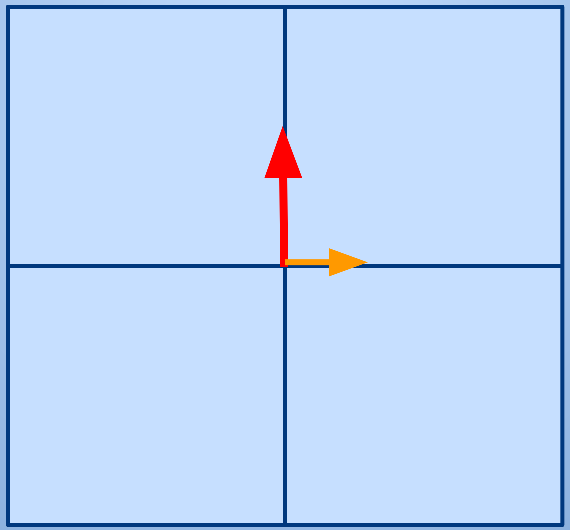
Simple Case:



# Common Elements: Advection

Still want to use same algorithm as before, but averaging face velocities to nodes is much harder.

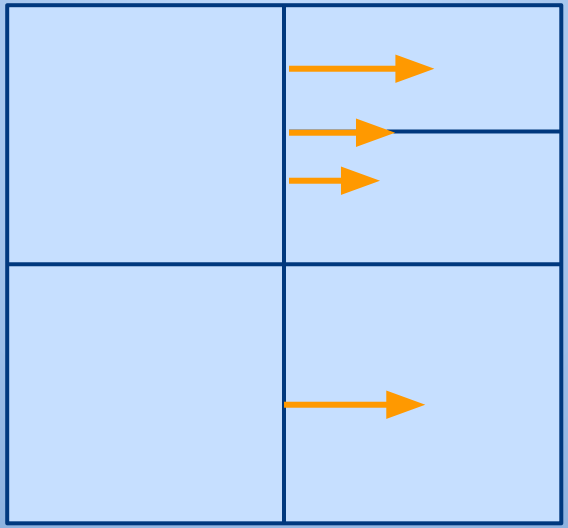
Simple Case:



# Common Elements: Advection

Still want to use same algorithm as before, but averaging face velocities to nodes is much harder.

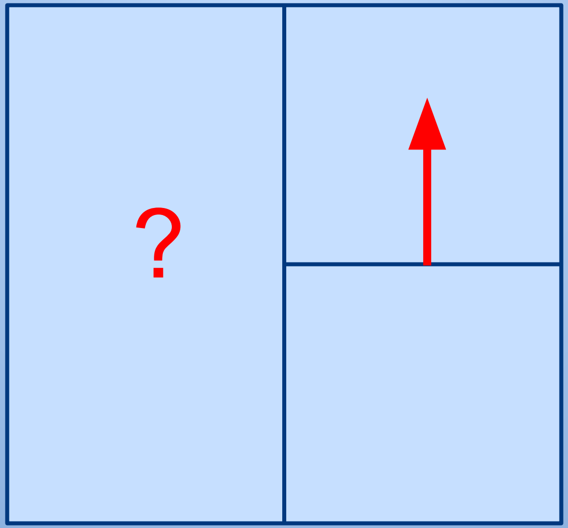
What if?



# Common Elements: Advection

Still want to use same algorithm as before, but averaging face velocities to nodes is much harder.

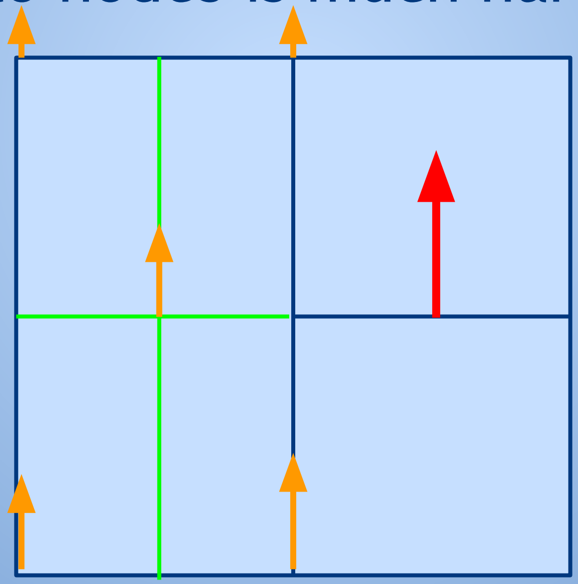
Or even worse:



# Common Elements: Advection

Still want to use same algorithm as before, but averaging face velocities to nodes is much harder.

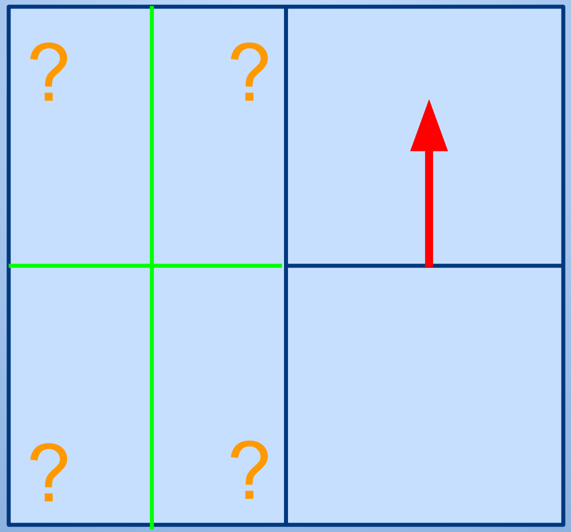
We can interpolate:



# Common Elements: Advection

Still want to use same algorithm as before, but averaging face velocities to nodes is much harder.

But then we're using nodal velocities to compute nodal velocities!

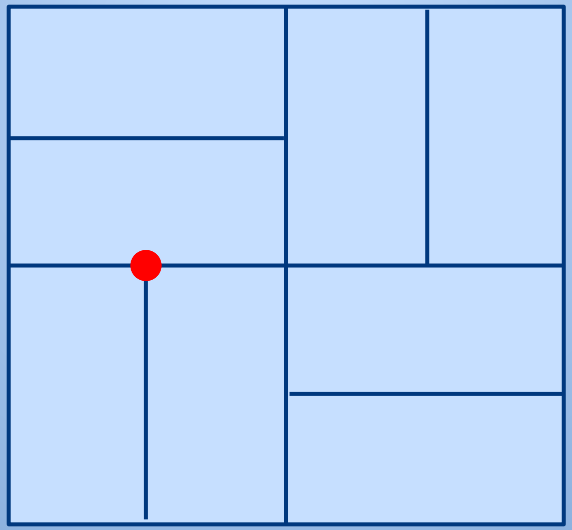


# Common Elements: Advection

Still want to use same algorithm as before, but averaging face velocities to nodes is much harder.

quadtree fine

recursive dependencies in k-d tree

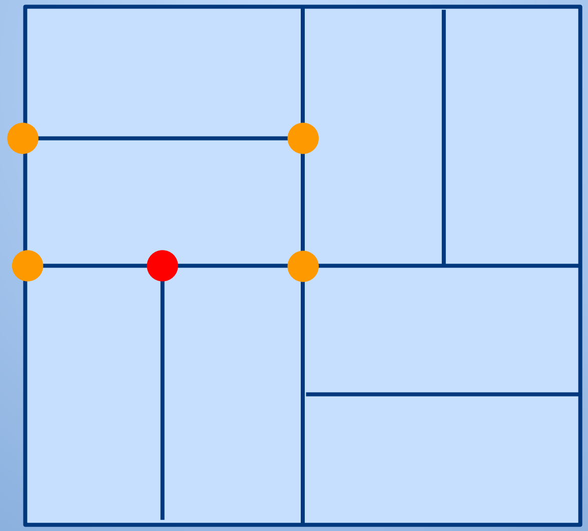


# Common Elements: Advection

Still want to use same algorithm as before, but averaging face velocities to nodes is much harder.

quadtree fine

recursive dependencies in k-d tree

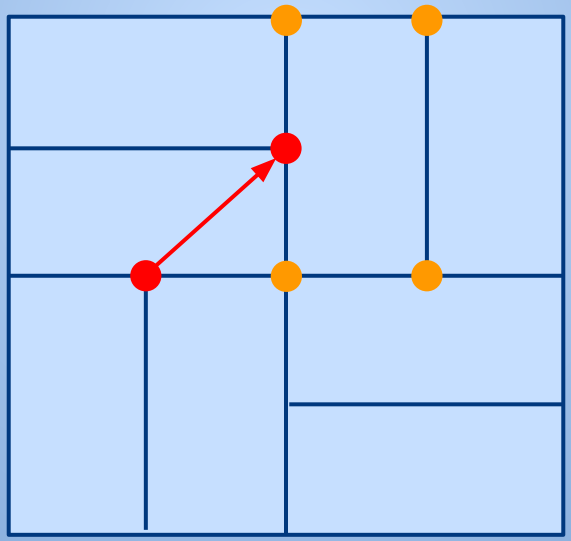


# Common Elements: Advection

Still want to use same algorithm as before, but averaging face velocities to nodes is much harder.

quadtree fine

recursive dependencies in k-d tree

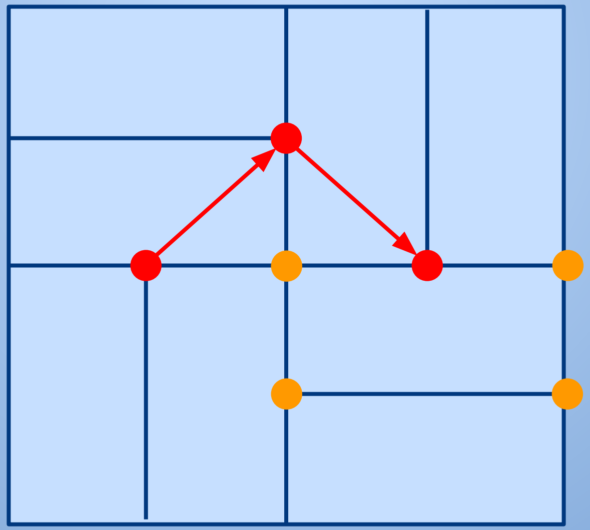


# Common Elements: Advection

Still want to use same algorithm as before, but averaging face velocities to nodes is much harder.

quadtree fine

recursive dependencies in k-d tree

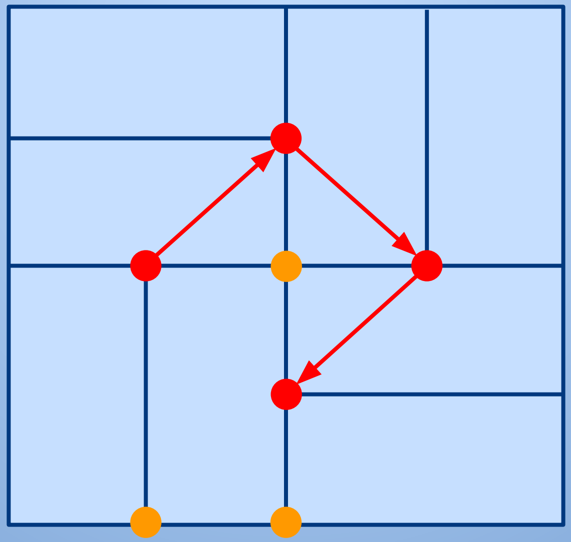


# Common Elements: Advection

Still want to use same algorithm as before, but averaging face velocities to nodes is much harder.

quadtree fine

recursive dependencies in k-d tree

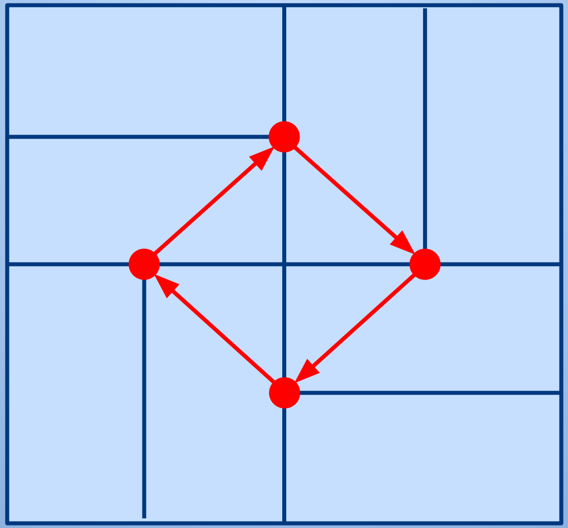


# Common Elements: Advection

Still want to use same algorithm as before, but averaging face velocities to nodes is much harder.

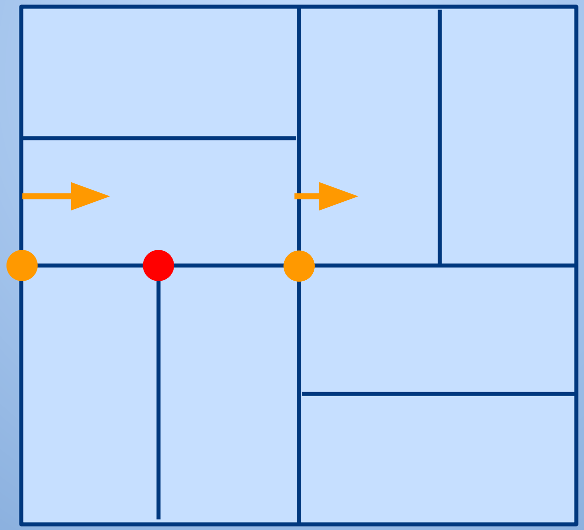
quadtree fine

recursive dependencies in k-d tree



# Common Elements: Advection

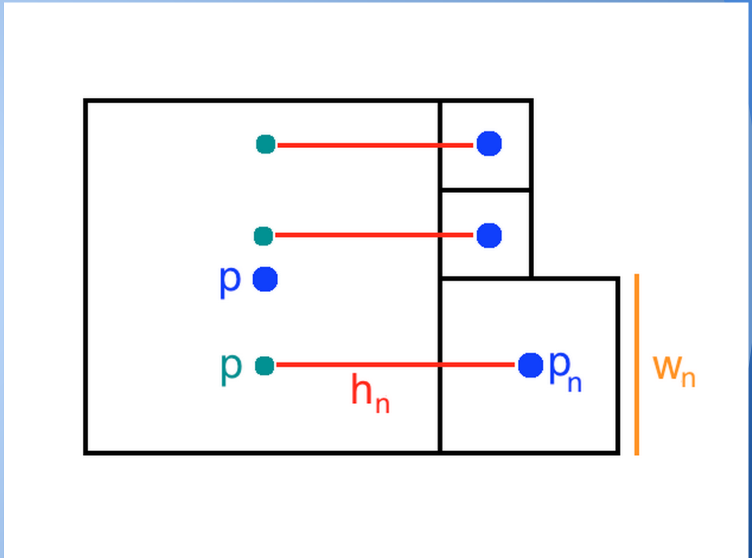
Remove dependency on possible T-junctions by using only corners of adjacent face.



# Common Elements: Projection

- Need pressure gradients for laplacian and projection
- Take weighted average of neighbor gradients at T-junction

$$\nabla_d p = \frac{1}{width} \sum w_n \frac{p_n - p}{h_n}$$



# Common Elements: Adaptivity

- Given adaptivity function f, can expand or contract nodes by comparing f(node) to threshold
- Many choices of adaptivity function f
  - Normalized vorticity
  - Velocity norm

# Different Elements - K-d tree adaptivity

- K-d tree must choose dimension to split
- Use dimension with larger velocity gradient

# Theoretical tradeoffs

- Quadtree is more compressed than K-d tree
  - If fluid ends up uniformly discretized, quadtree will be more efficient and have less error
- If the K-d tree can fit the problem better, it will be more efficient
  - Fewer leaves
  - Less error

# Obtaining Results

Two tests:

1. How well do the K-d tree and quadtree discretize to a given input?
2. How much computation do the K-d tree and quadtree require to get the same amount of error in the poisson solution?

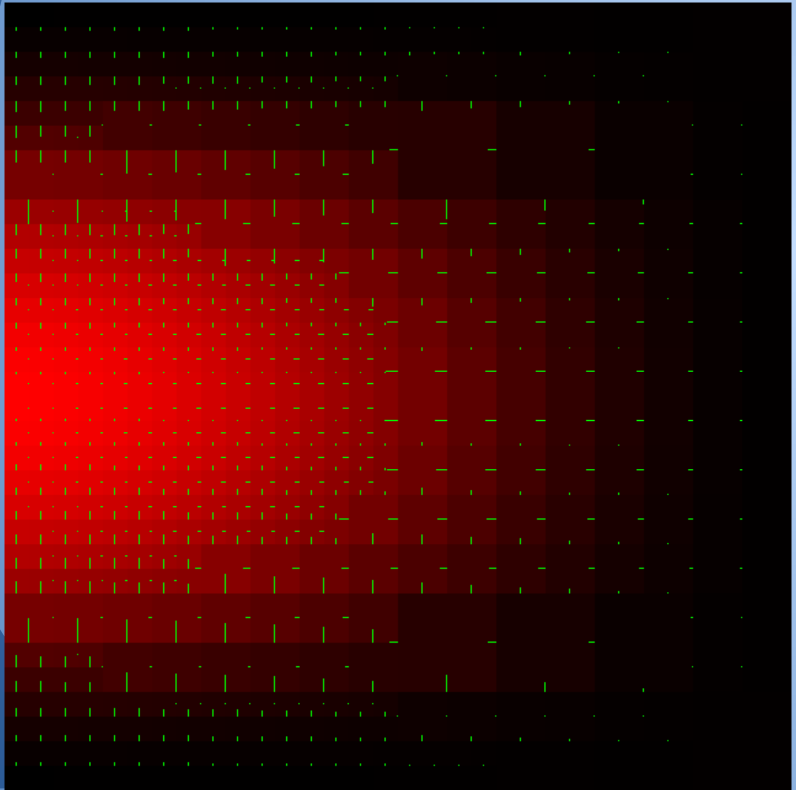
# 1. Adaptivity to a Function

Same input and same adaptation criteria on both trees

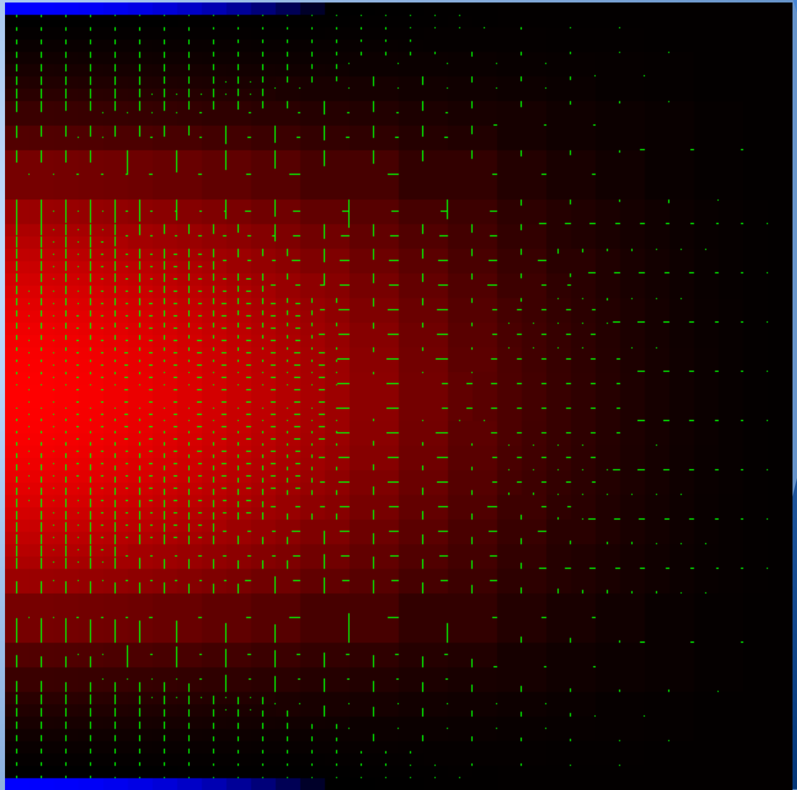
To input a velocity field, I input a function f(x,y) and set <vx, vy> to <df/dx, df/dy>.

f(x,y) = (2x^3 - 3x^2 + 1) *((2y-1)^4 - 2*(2y-1)^2 + 1)

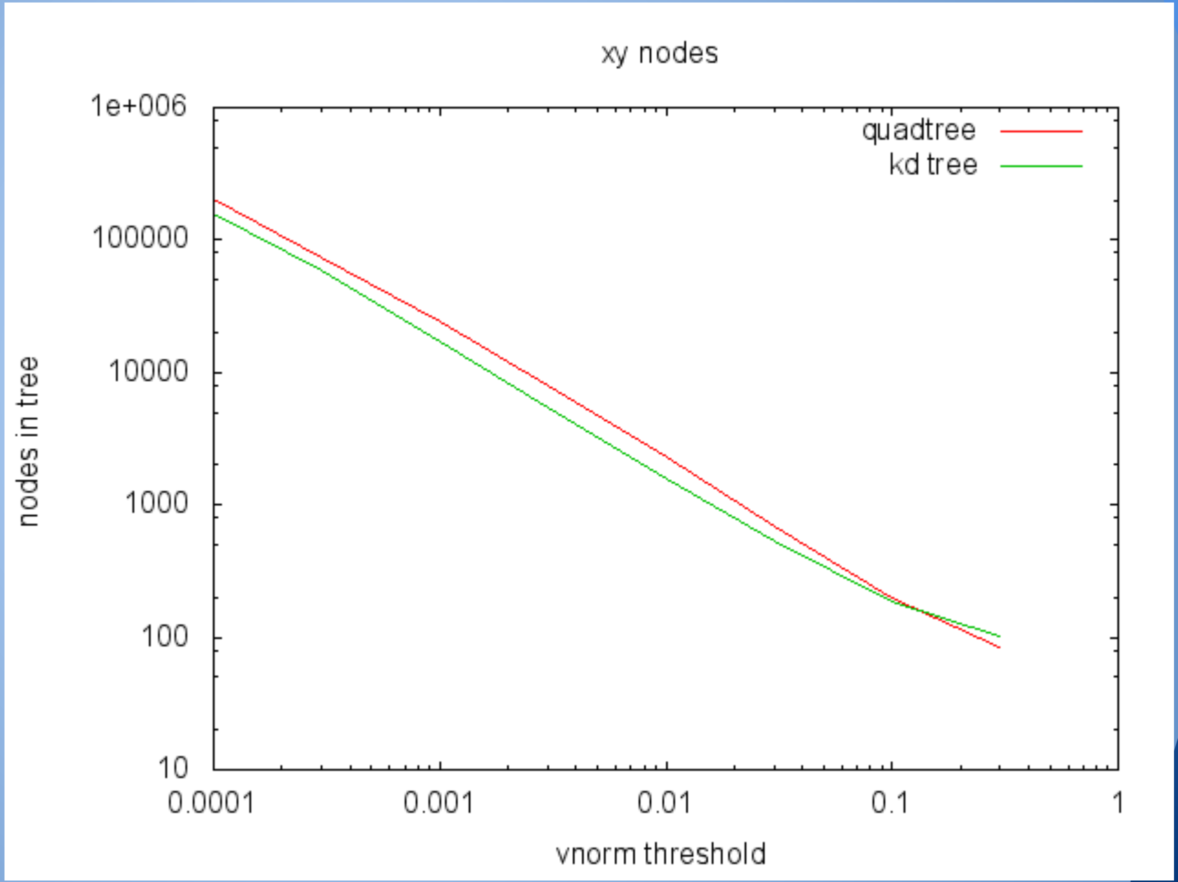
## Quadtree



## K-d Tree



# f(x,y) = (2x^3 - 3x^2 + 1) *((2y-1)^4 - 2*(2y-1)^2 + 1)



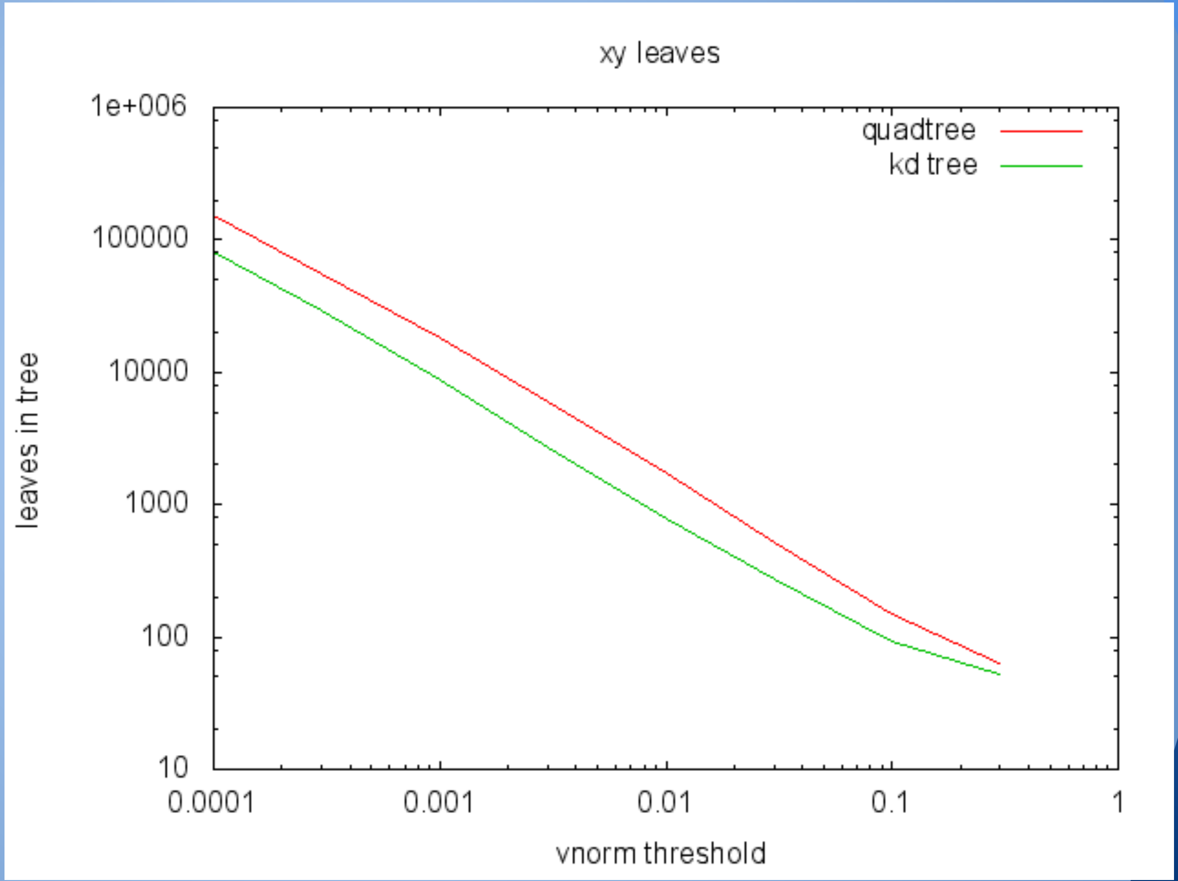

## nodes:

kd: 5% to 33% fewer nodes

$f(x,y) = (2x^3 - 3x^2 + 1) *((2y-1)^4 - 2*(2y-1)^2 + 1)$

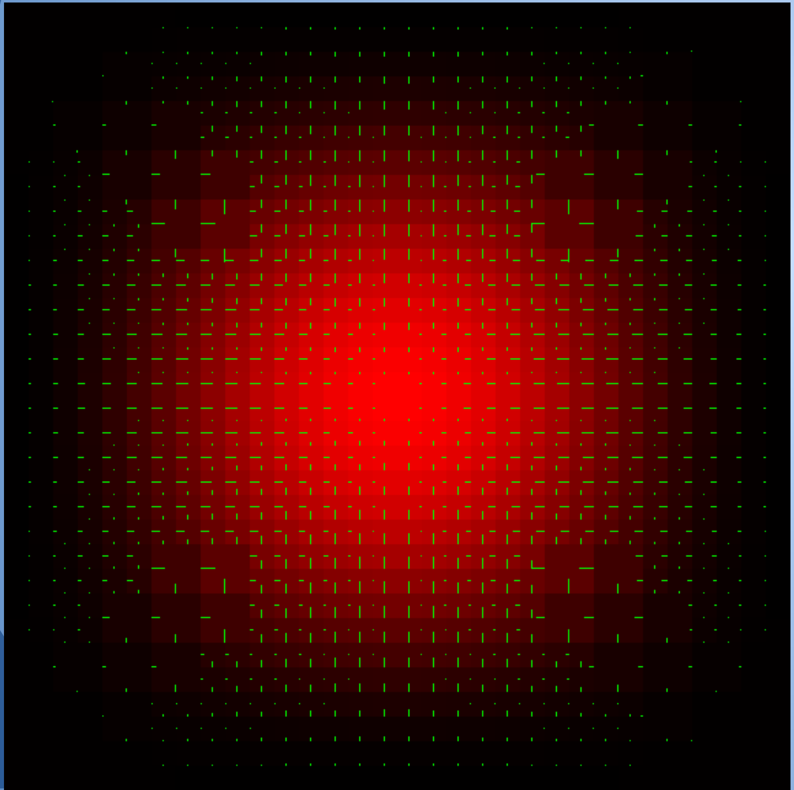
leaves:

kd: 36% to 55% fewer leaves

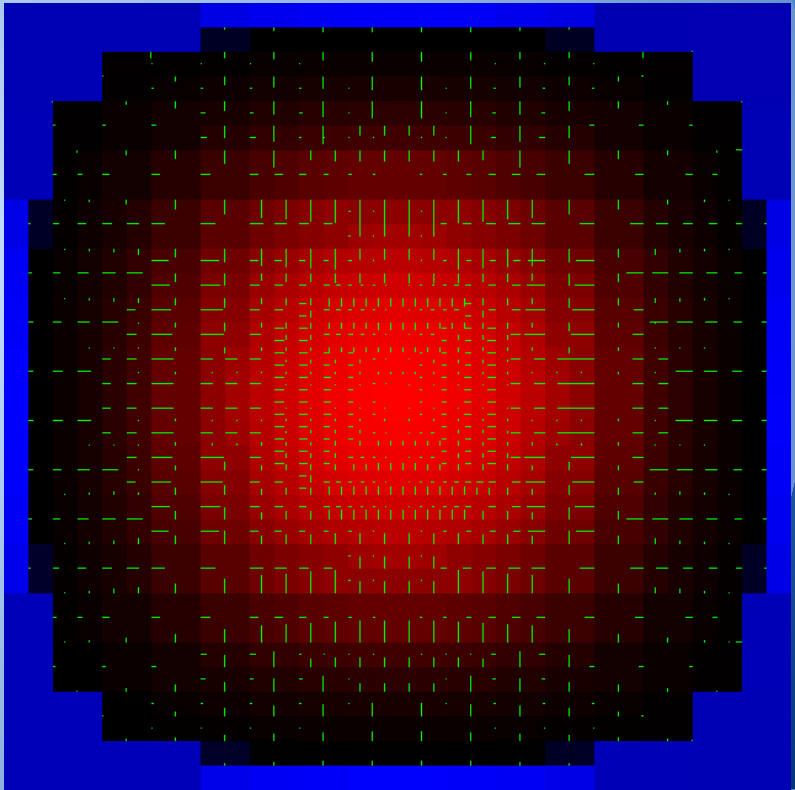


$$f(x,y) = (1-\cos(2\pi x))(1-\cos(2\pi y))$$

## Quadtree



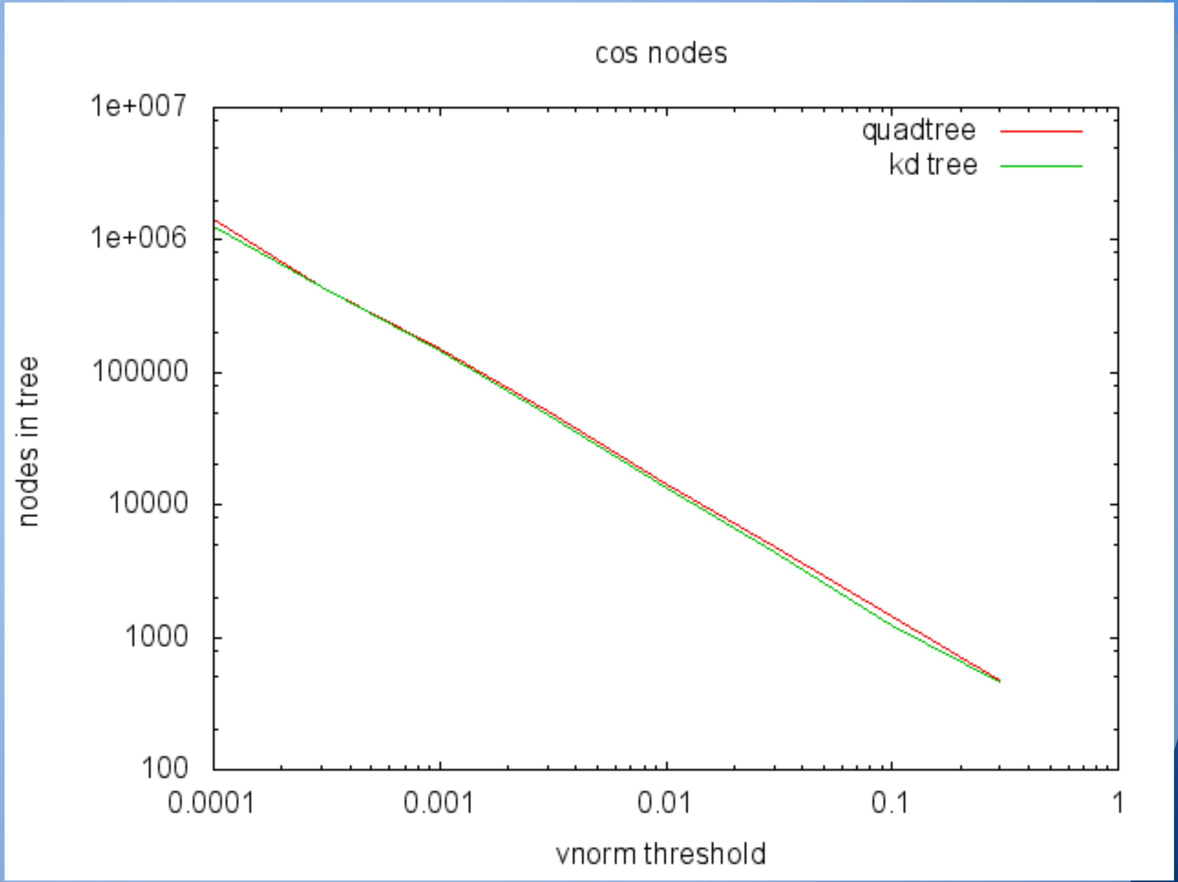
## K-d Tree



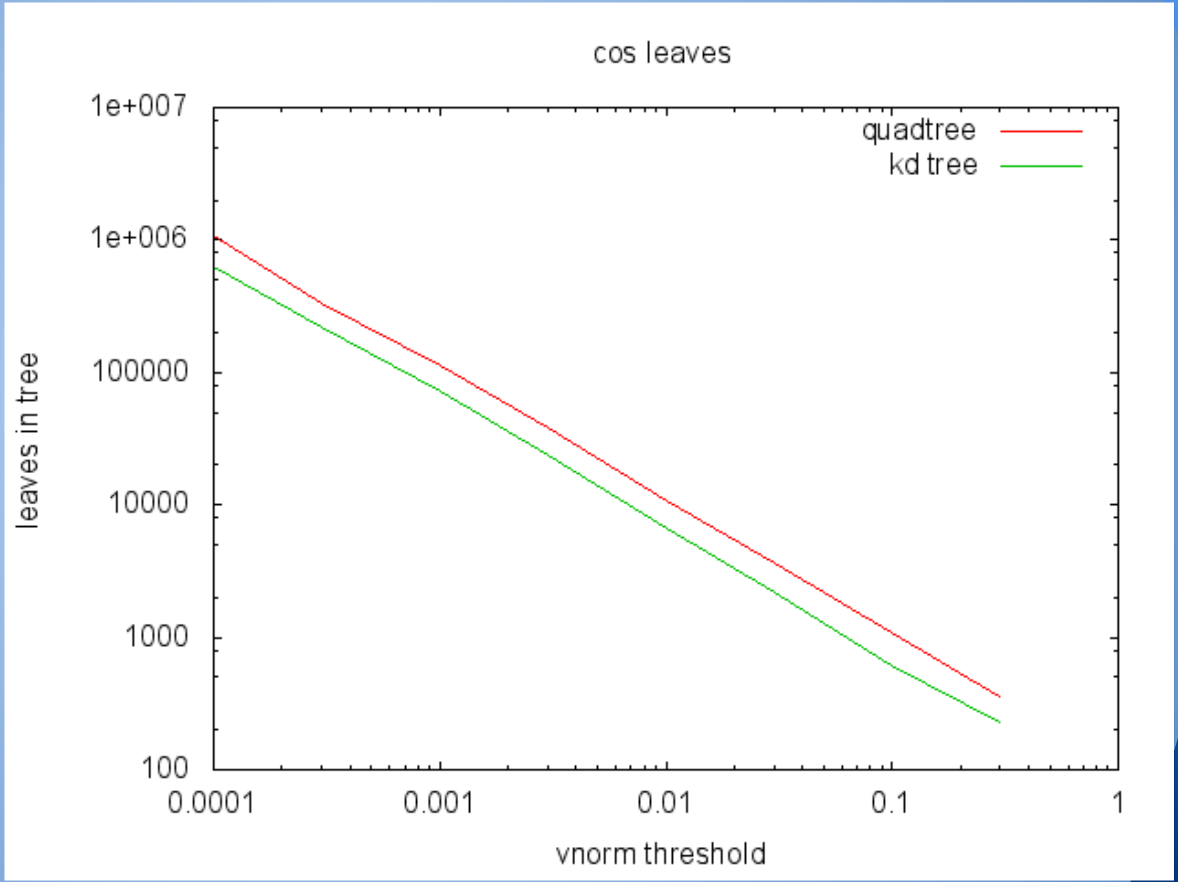
# $f(x,y) = (1-\cos(2\pi x))(1-\cos(2\pi y))$

nodes:

kd: 2% to 15% fewer nodes



# f(x,y) = (1-cos(2πx))(1-cos(2πy))

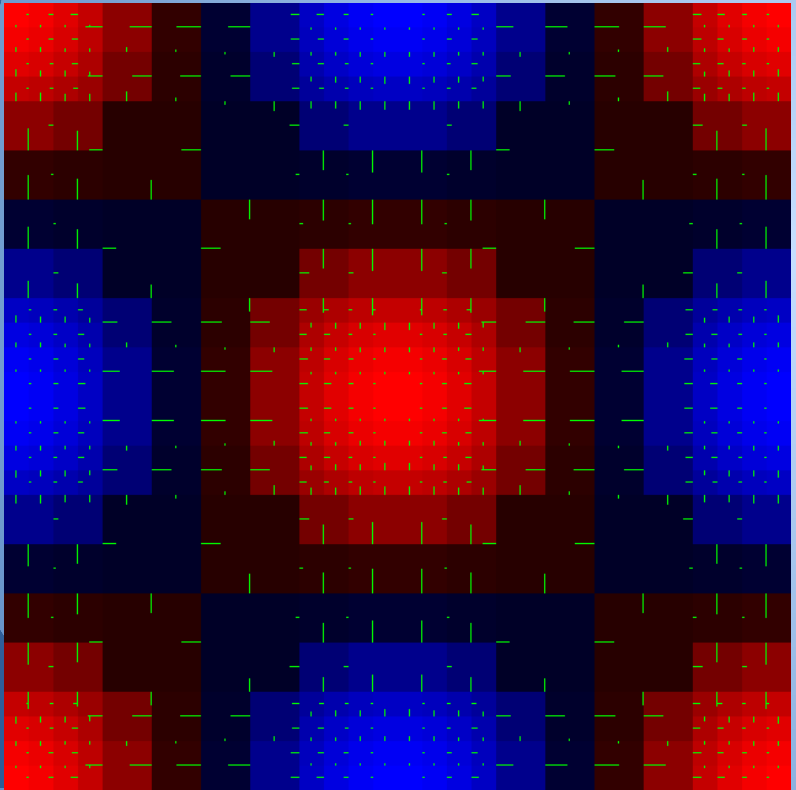



## leaves:

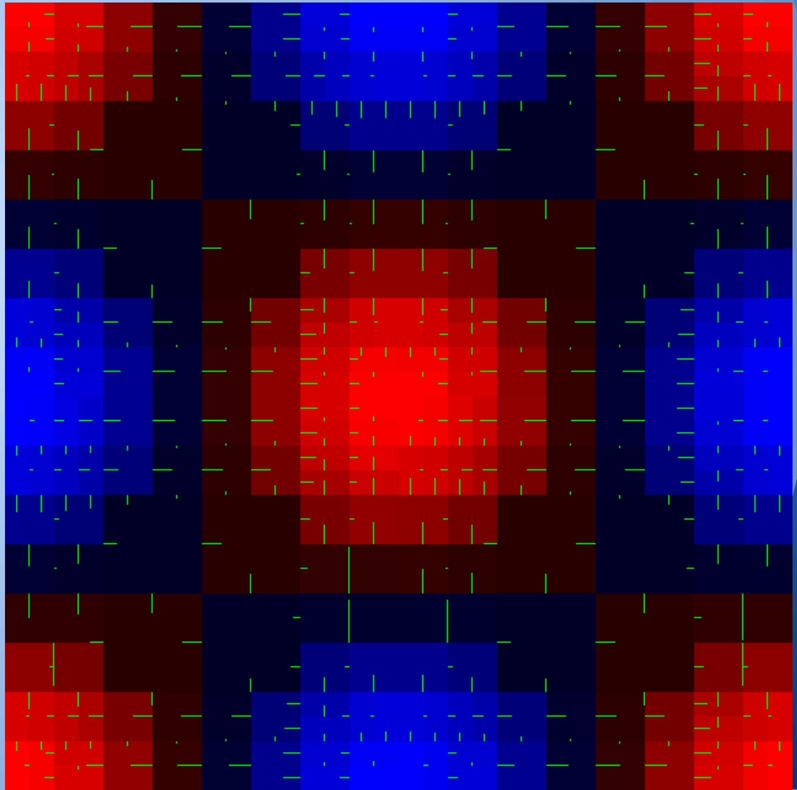
kd: 32% to 43% fewer leaves

$f(x,y) = \cos(\pi kx)\cos(\pi ly)$, $k=2$, $l=2$

Quadtree



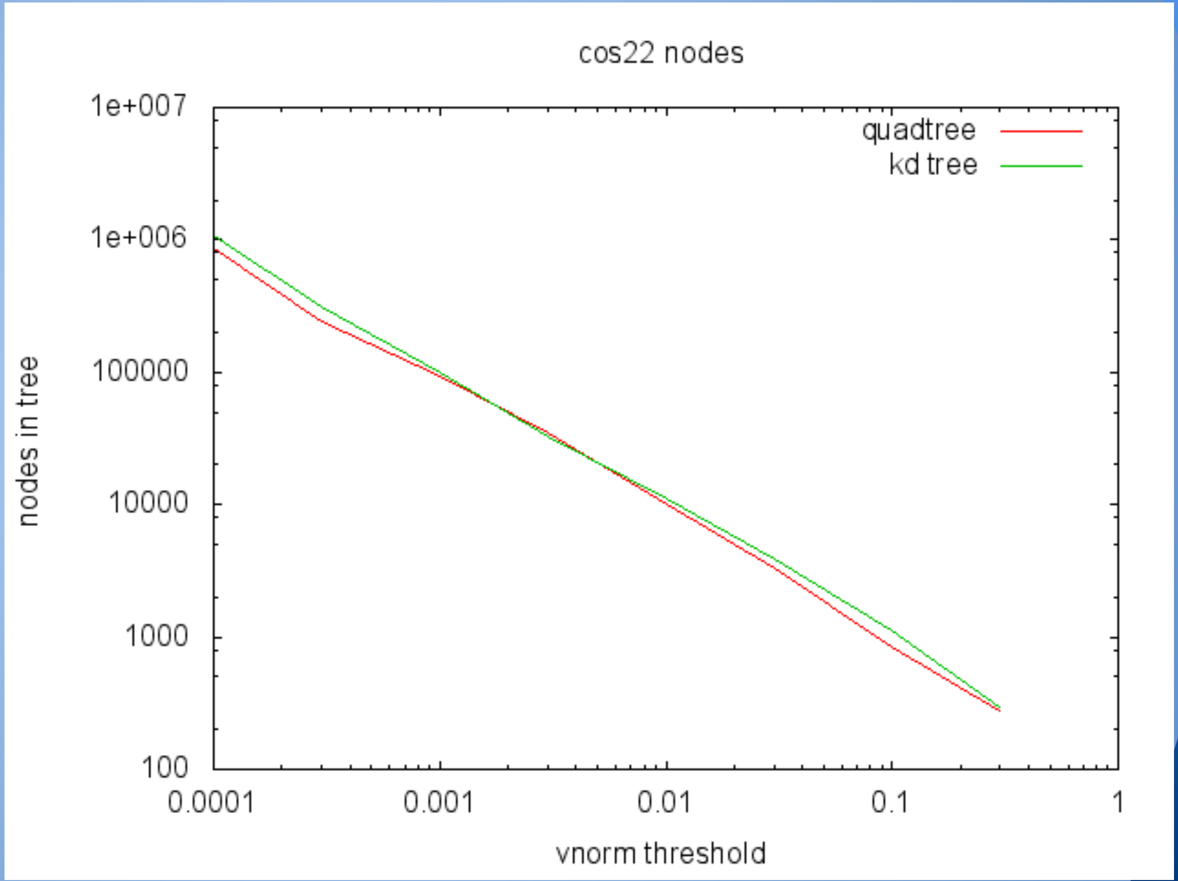
K-d Tree



# $f(x,y) = \cos(\pi kx)\cos(\pi ly)$, k=2, l=2

nodes:

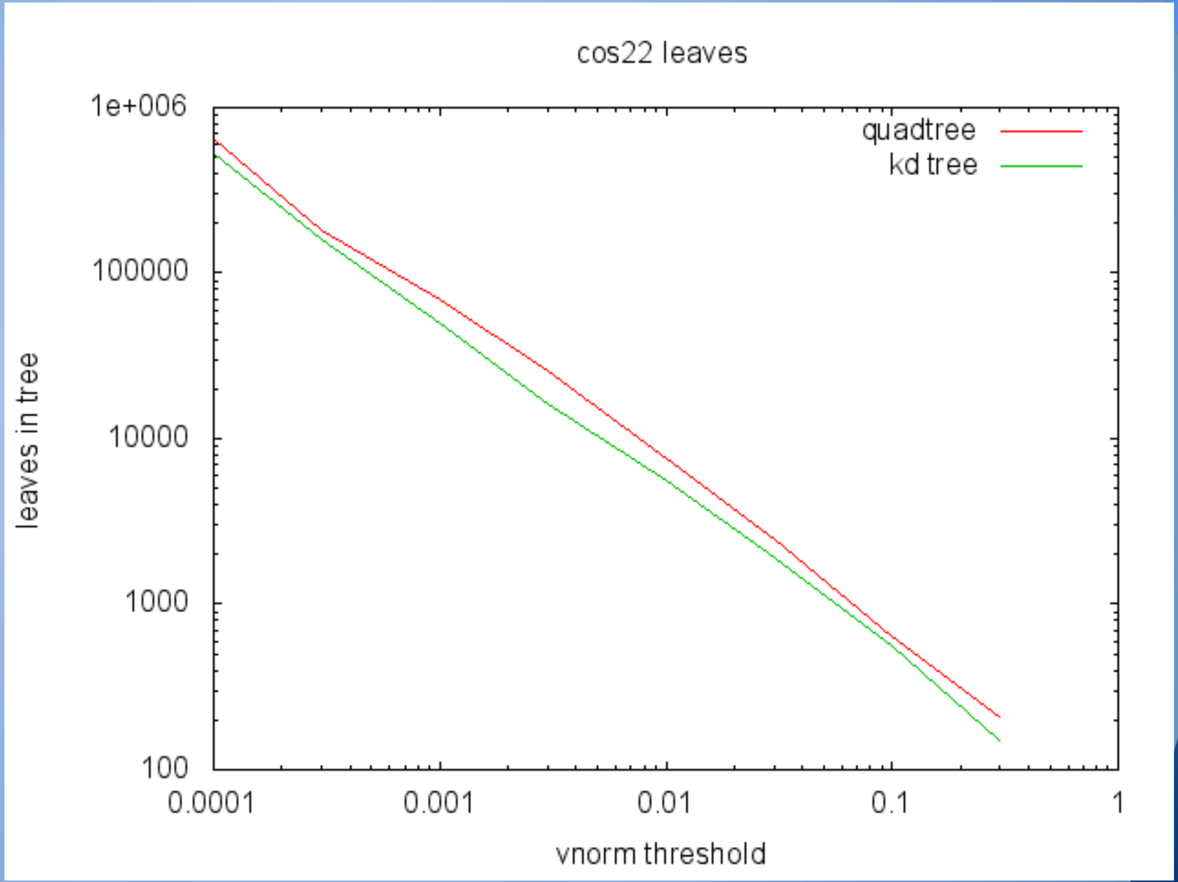
kd: 6% to -32% fewer nodes



# $f(x,y) = \cos(\pi kx)\cos(\pi ly)$, $k=2$, $l=2$

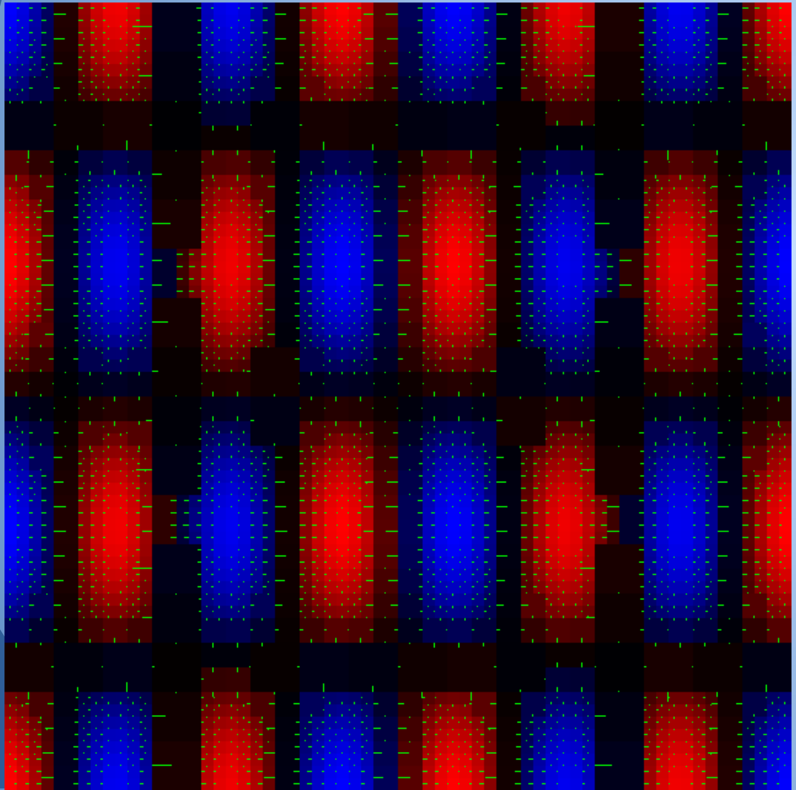
leaves:

kd: 11% to 32% fewer leaves

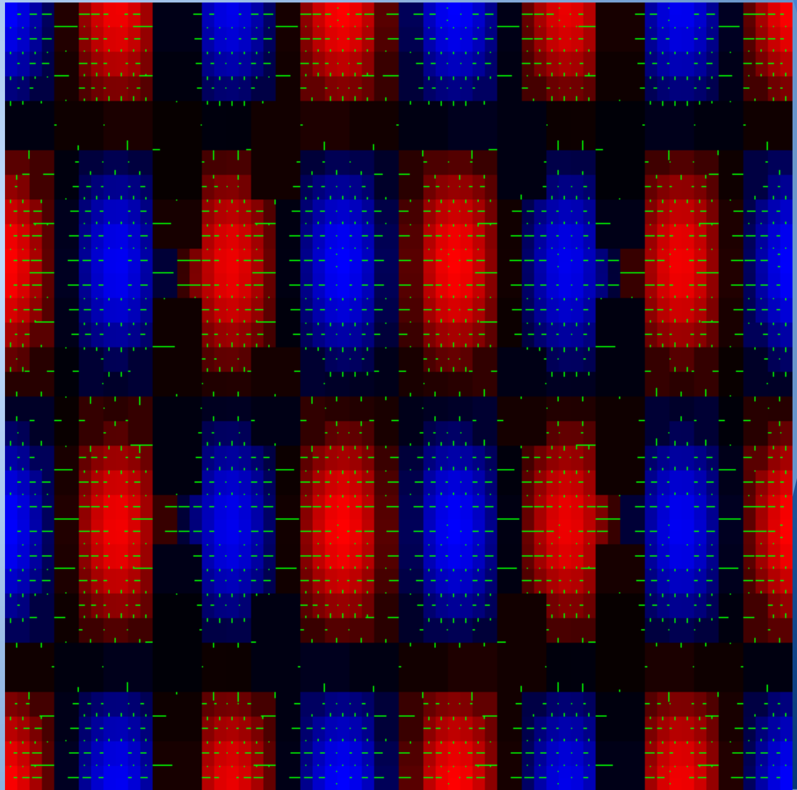


# $f(x,y) = \cos(\pi kx)\cos(\pi ly)$, $k=7$, $l=3$

Quadtree



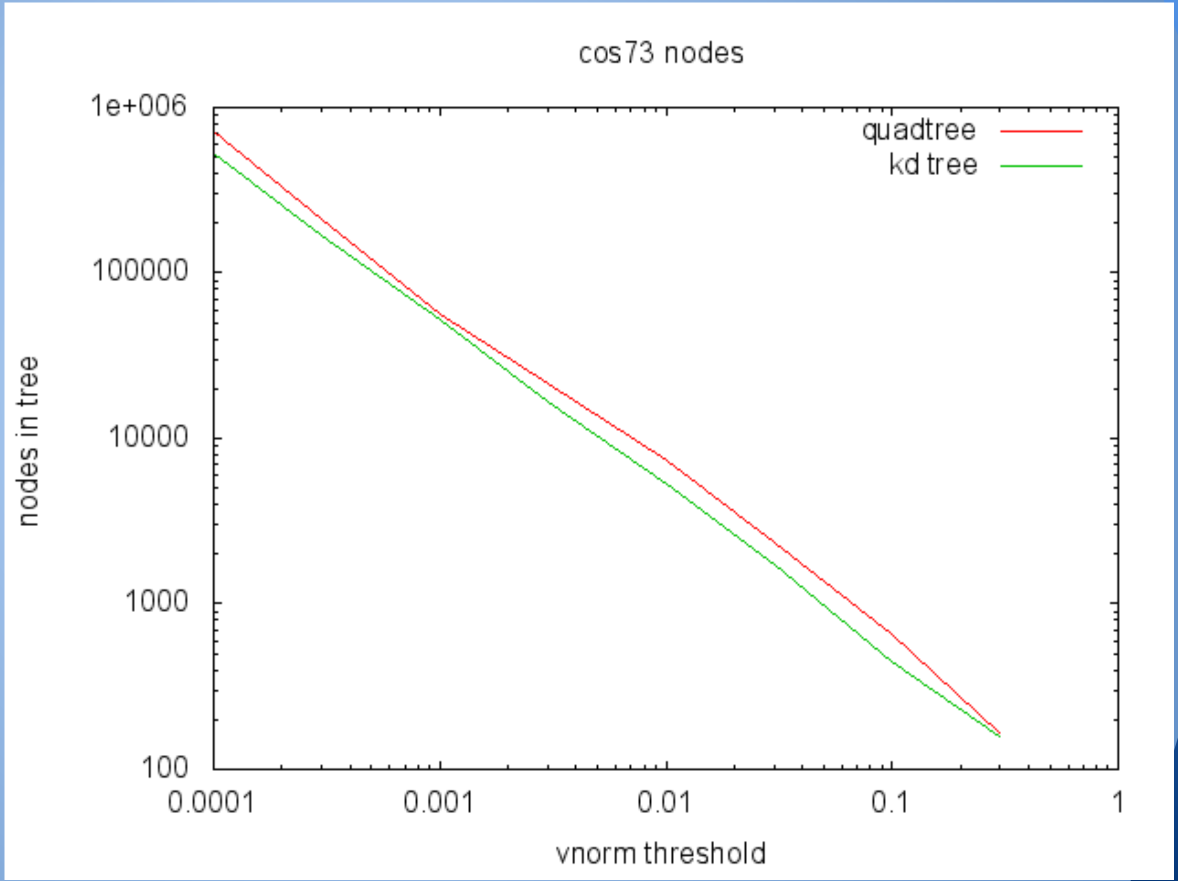
K-d Tree



# f(x,y) = cos(πkx)cos(πly), k=7, l=3

nodes:

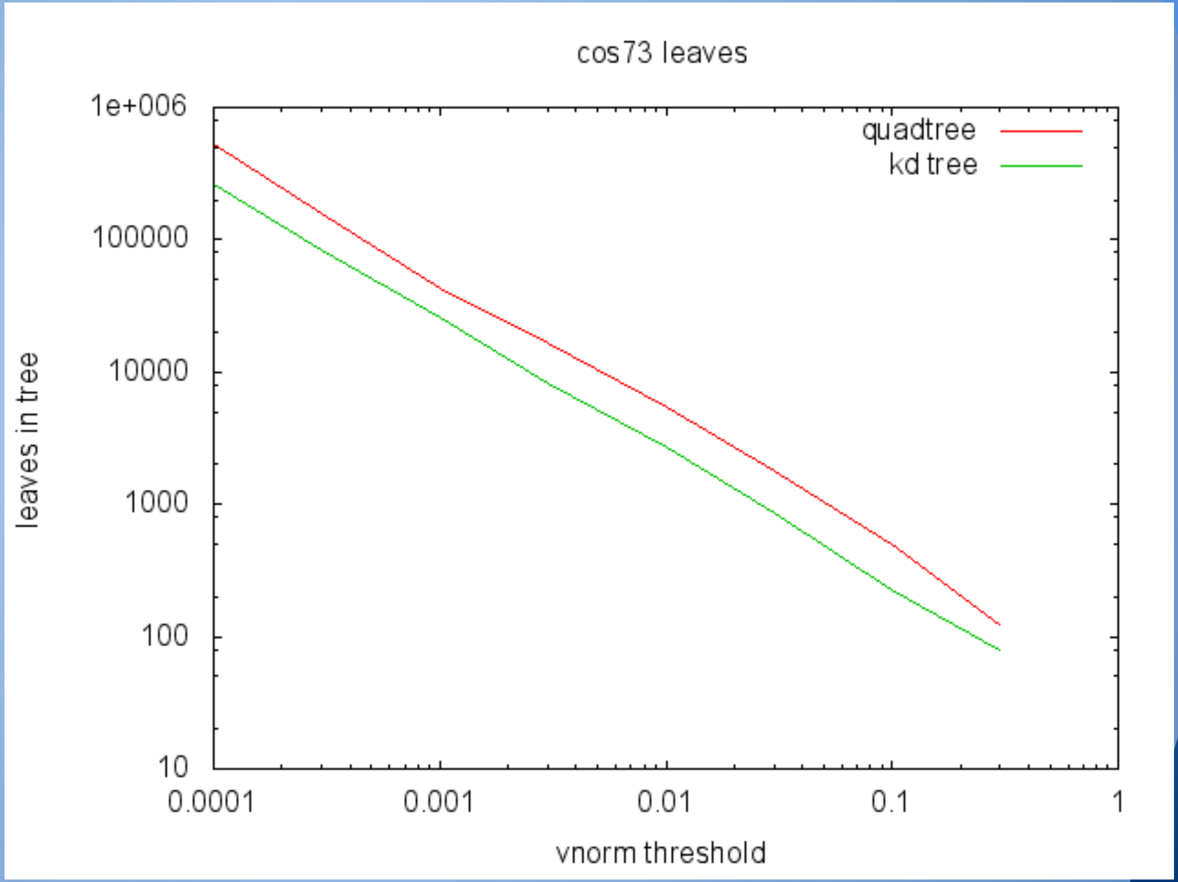
kd: 4% to 32% fewer nodes



# $f(x,y) = \cos(\pi kx)\cos(\pi ly)$, $k=7$, $l=3$

leaves:

kd: 35% to 55% fewer leaves



# 1. Adapting to a Function

Clearly the K-d tree is able to represent the same topology with fewer nodes and leaves

The K-d tree has a bigger advantage when the function is less uniform between the x and y dimensions

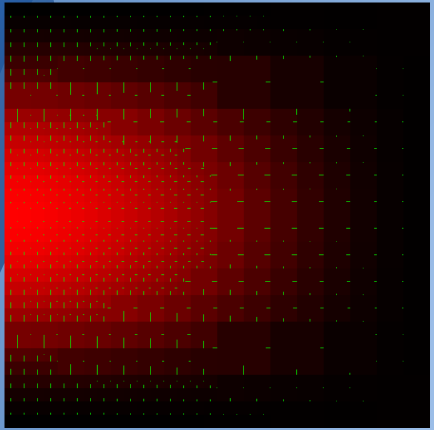
# 2. Performance vs Error

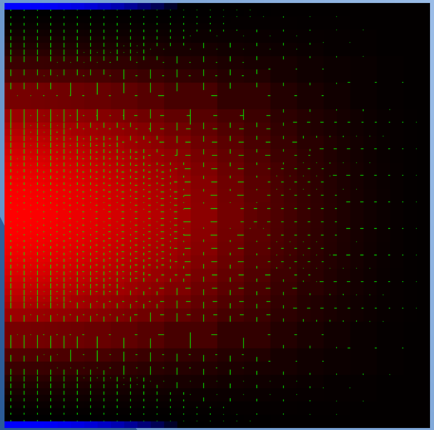
If the K-d tree is in fact better than the quadtree, it should achieve the same amount of error in the solution to the poisson equation, with less computation.

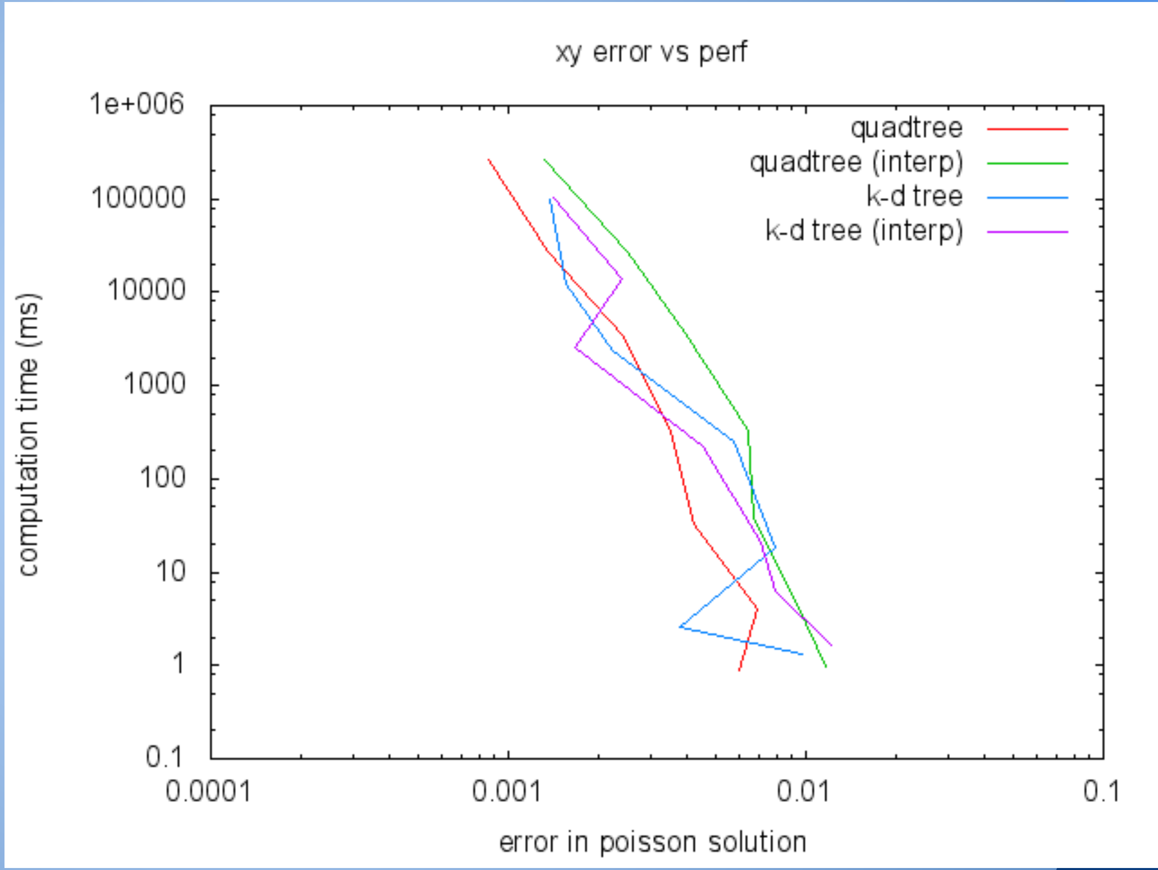
# Determining Error

- Use same velocity input as previous function
- Running poisson solver on input where velocity is the gradient of f returns f
- Can compare output to true value of f to compute error

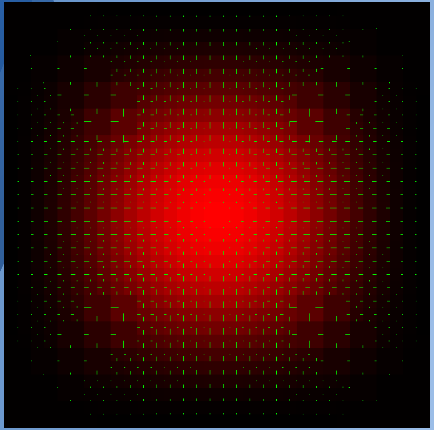
$$f(x,y) = (2x^3 - 3x^2 + 1) * ((2y-1)^4 - 2*(2y-1)^2 + 1)$$

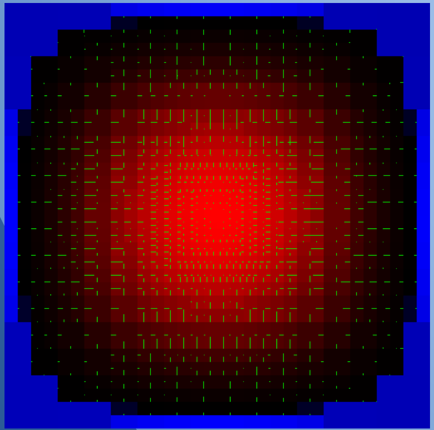


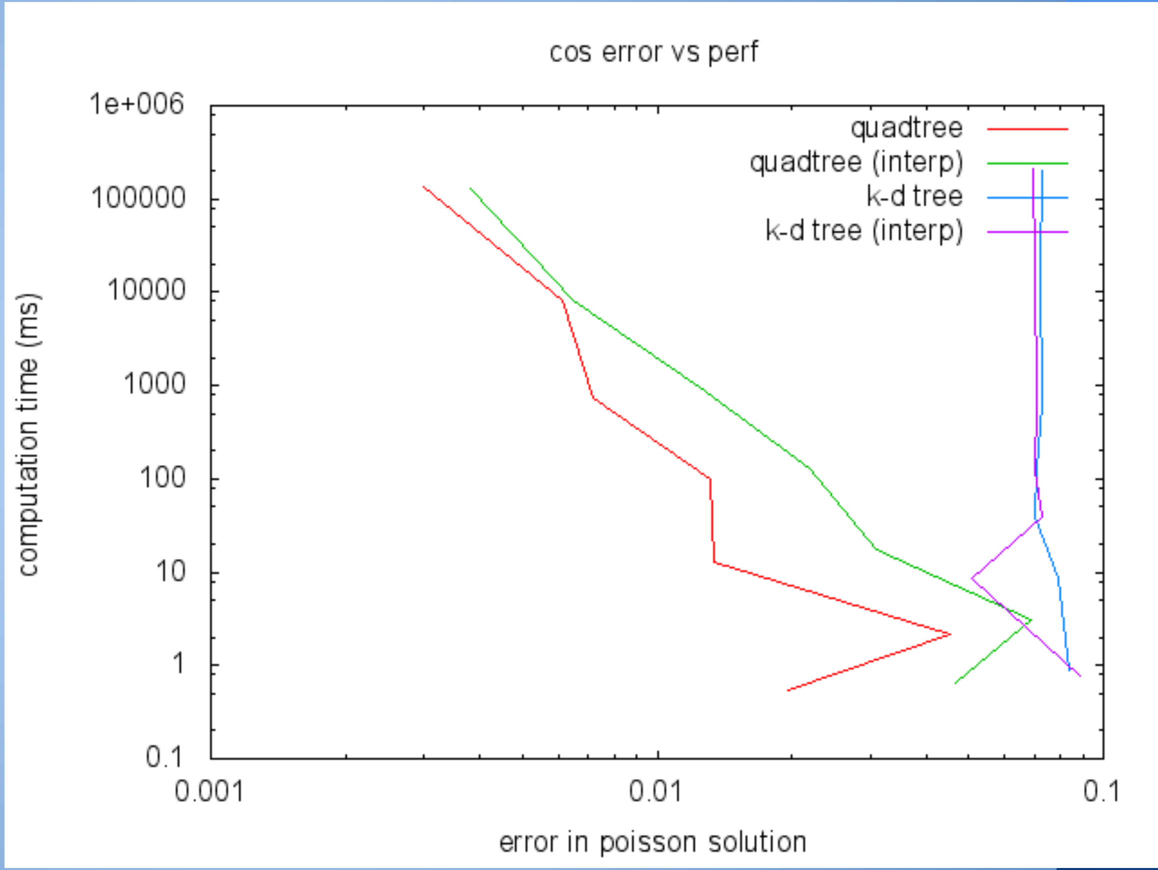




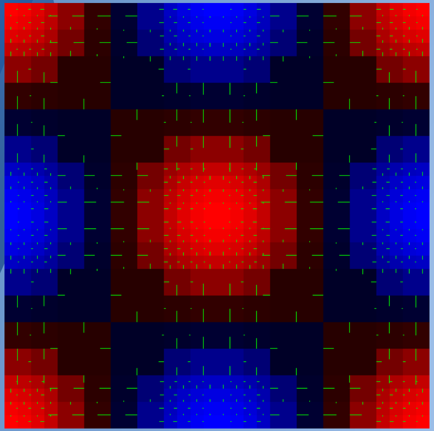
# $f(x,y) = (1-\cos(2\pi x))(1-\cos(2\pi y))$

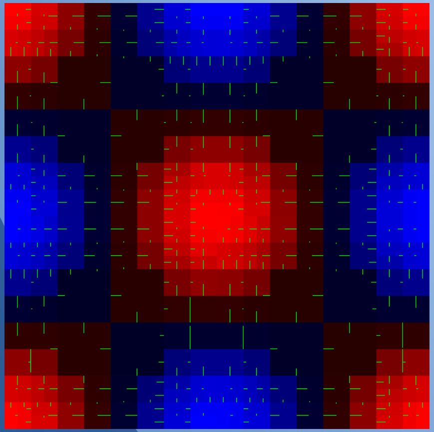


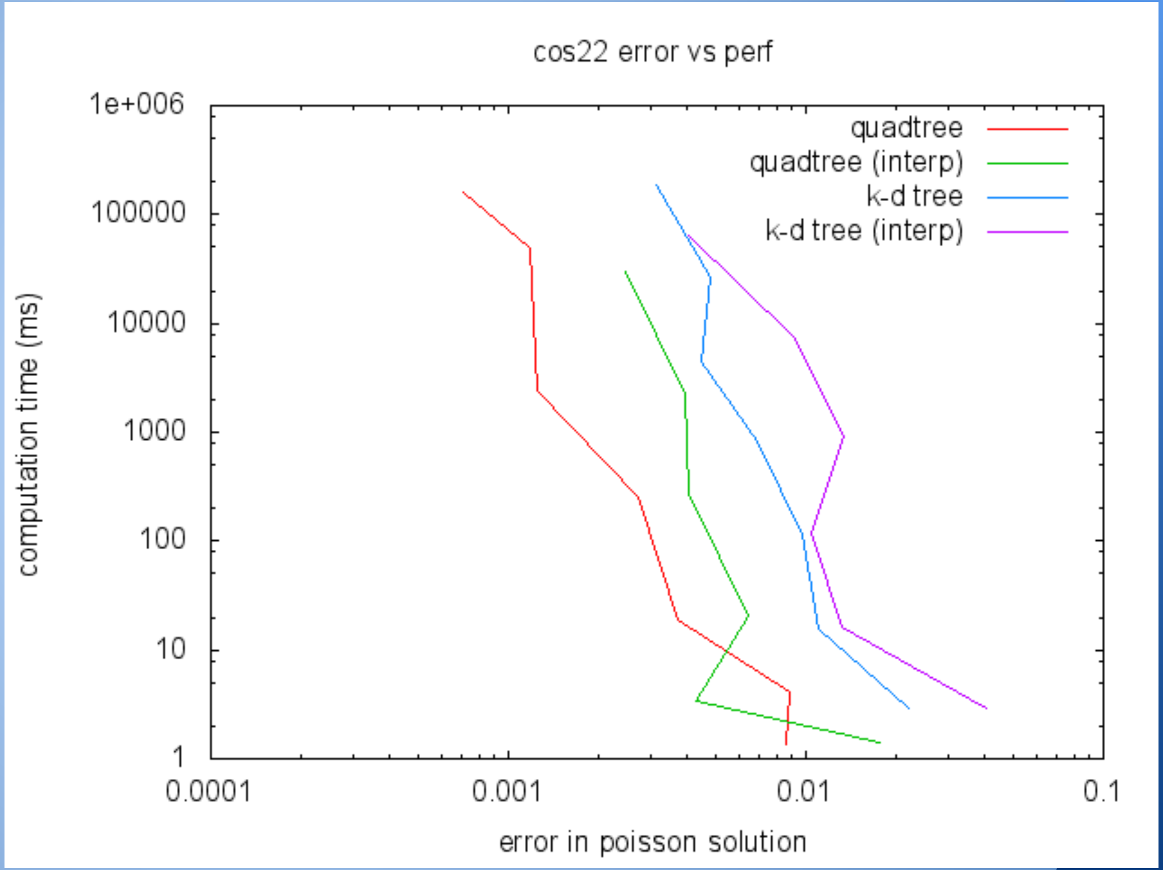




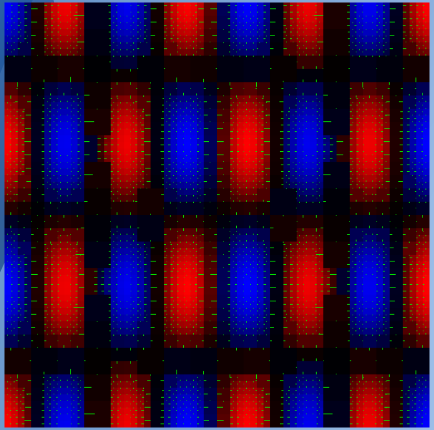
# $f(x,y) = \cos(\pi kx)\cos(\pi ly)$, k=2, l=2

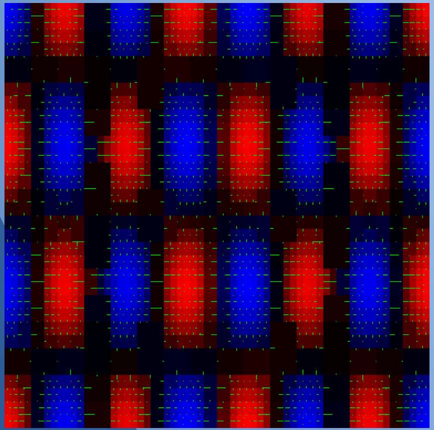


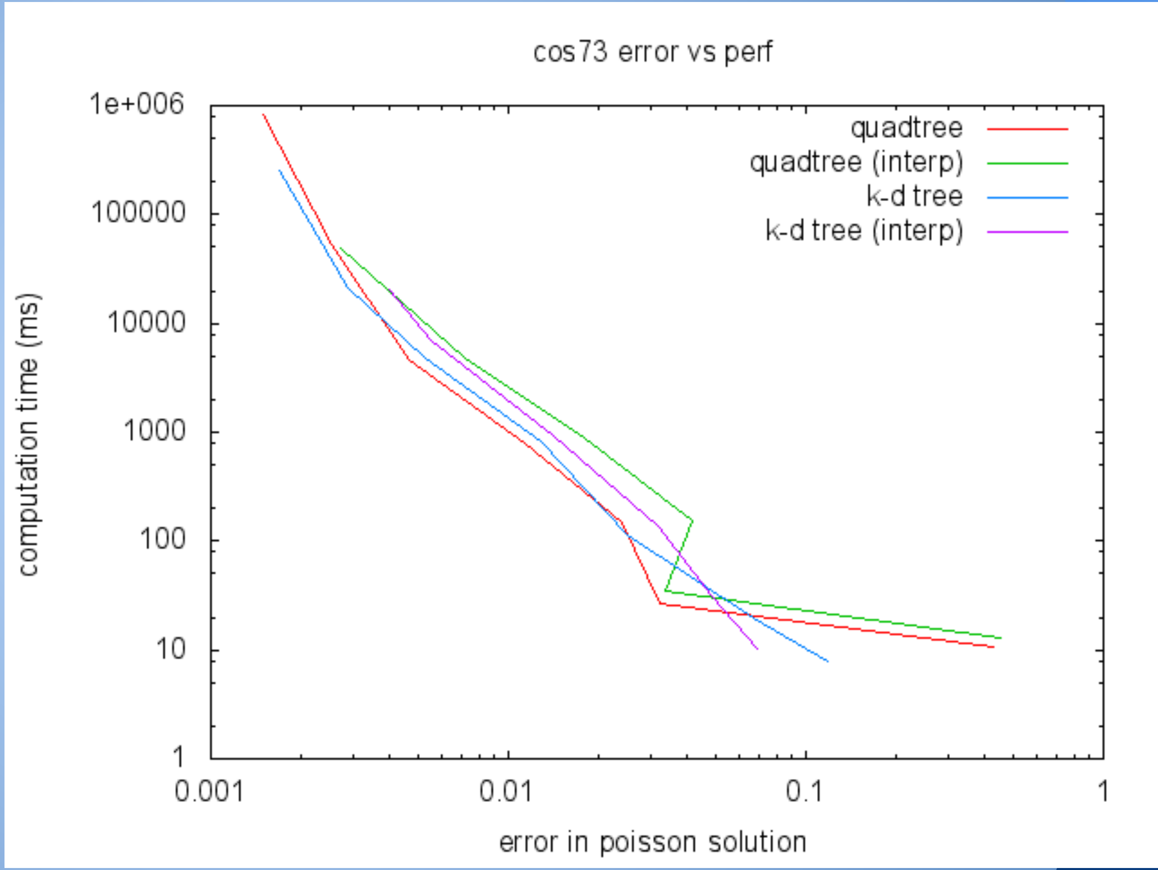




# f(x,y) = cos(πkx)cos(πly), k=7, l=3







# Conclusion

- K-d trees discretize the function better than quadtrees
- With my discretization, they perform worse on fluids close to uniform between the dimensions

# Future Work

- Find better discretization of pressure computations
- Liquid simulation
- 3 dimensions
- Add solid object interaction
- Different K-d tree splitting algorithms
    - Try not splitting on the median of the face
- Use Sparse Linear Solvers instead of relaxation

# Acknowledgements

Thanks to my advisor Dr. Fussell getting me started early and helping me with ideas, my second reader Dr. Vouga helping me to better understand the problem I was trying to solve, and my third reader Dr. van de Geijn.