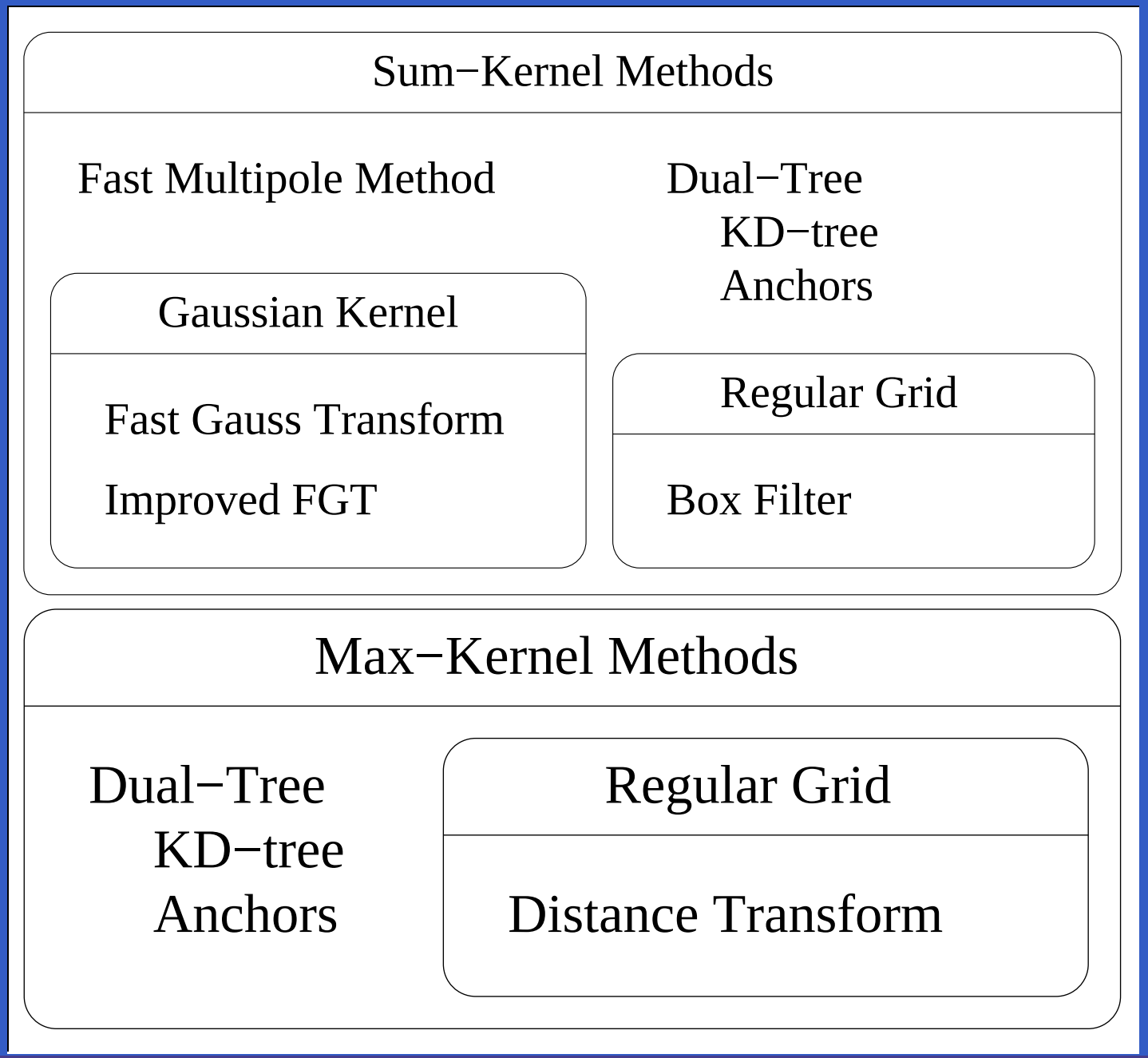
Empirical Comparisons of Fast Methods

Dustin Lang and **Mike Klaas**

{dalang, klaas}@cs.ubc.ca

University of British Columbia

December 17, 2004



A Map of Fast Methods

We claim that to be useful for other researchers, Fast Methods need:

- guaranteed, adjustable error bounds: users can set the error bound low during development stage, then experiment once they know their code works.
- no parameters that need to be adjusted by users (other than error tolerance).
- documented error behaviour: we must explain the properties of our approximation errors.

The Role of Fast Methods

We tested:

Sum-Kernel: $$f_j = \sum_{i=1}^{N} w_i \exp\left(-\frac{\|x_i - y_j\|_2^2}{h^2}\right)$$

Max-Kernel:

$$x_j^* = \underset{i=1}{\overset{N}{\operatorname{argmax}}} \left[w_i \exp\left(-\frac{\|x_i - y_j\|_2^2}{h^2}\right) \right]$$

Gaussian kernel, fixed bandwidth h,
non-negative weights w_i, $j = 1 \ldots N$.

Testing Framework

For the Sum-Kernel problem, we allow a given error tolerance ϵ: $|f_j - f_{\text{true}}| \leq \epsilon$ for each j.

We tested:

- Fast Gauss Transform (FGT)
- Improved Fast Gauss Transform (IFGT)
- Dual-Tree with kd-tree (KDtree)
- Dual-Tree with ball-tree constructed via Anchors Hierarchy (Anchors)

Testing Framework (2)

Fast Gauss Transform (FGT) code by Firas Hamze of UBC.

KDtree and Anchors Dual-Tree code by Dustin.

The same Dual-Tree code was used for KDtree and Anchors.

Methods Tested

Ramani Duraiswami and Changjiang Yang generously gave their code for the Improved Fast Gauss Transform (IFGT).

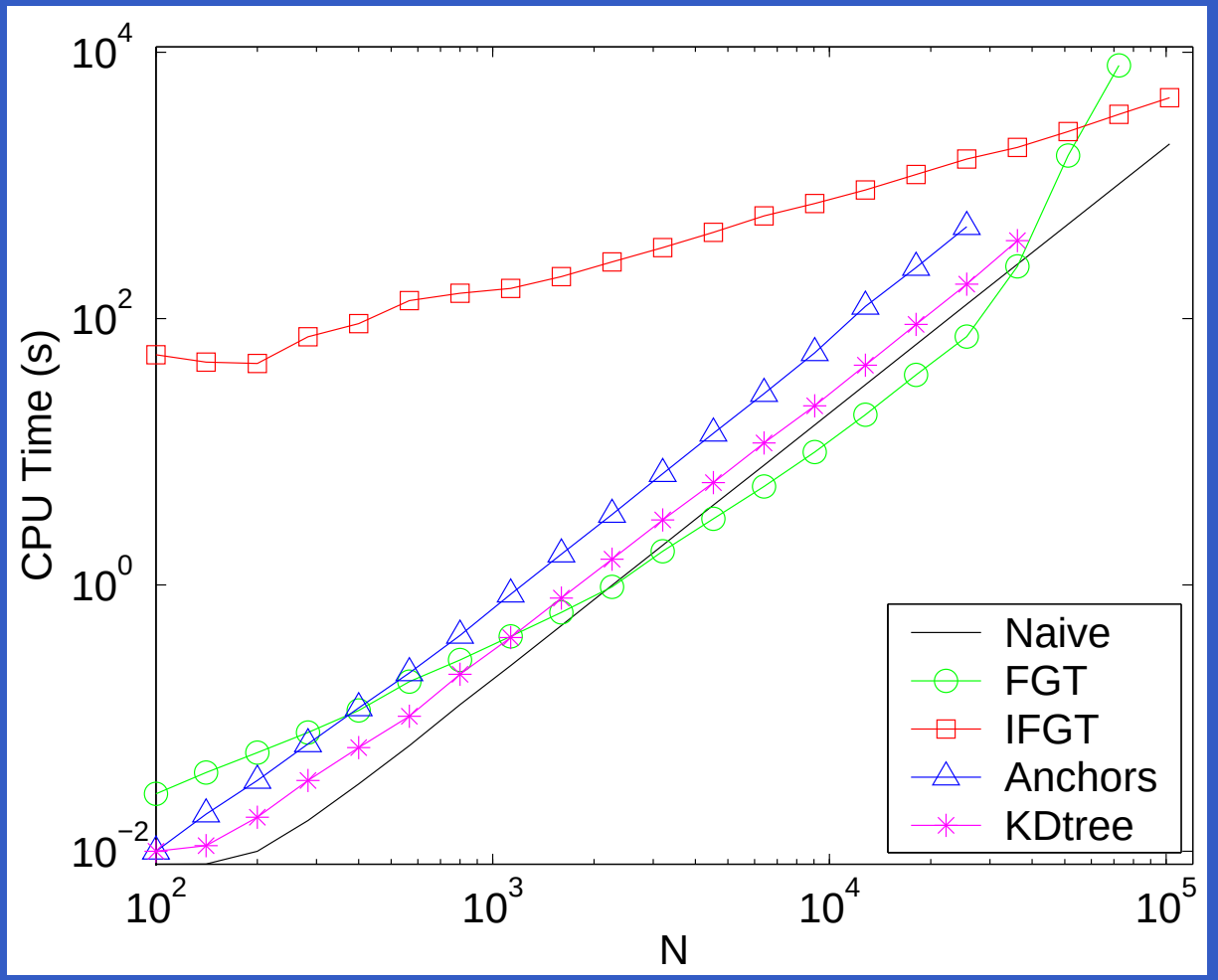
To make the IFGT fit in our testing framework, we had to devise a method for choosing parameters. Our method seems reasonable but is probably not optimal.

All methods: in *C* with *Matlab* bindings.

Methods Tested (2)

Uniformly distributed points, uniformly distributed weights, 3 dimensions, large bandwidth $h = 0.1$, $\epsilon = 10^{-6}$: Time.

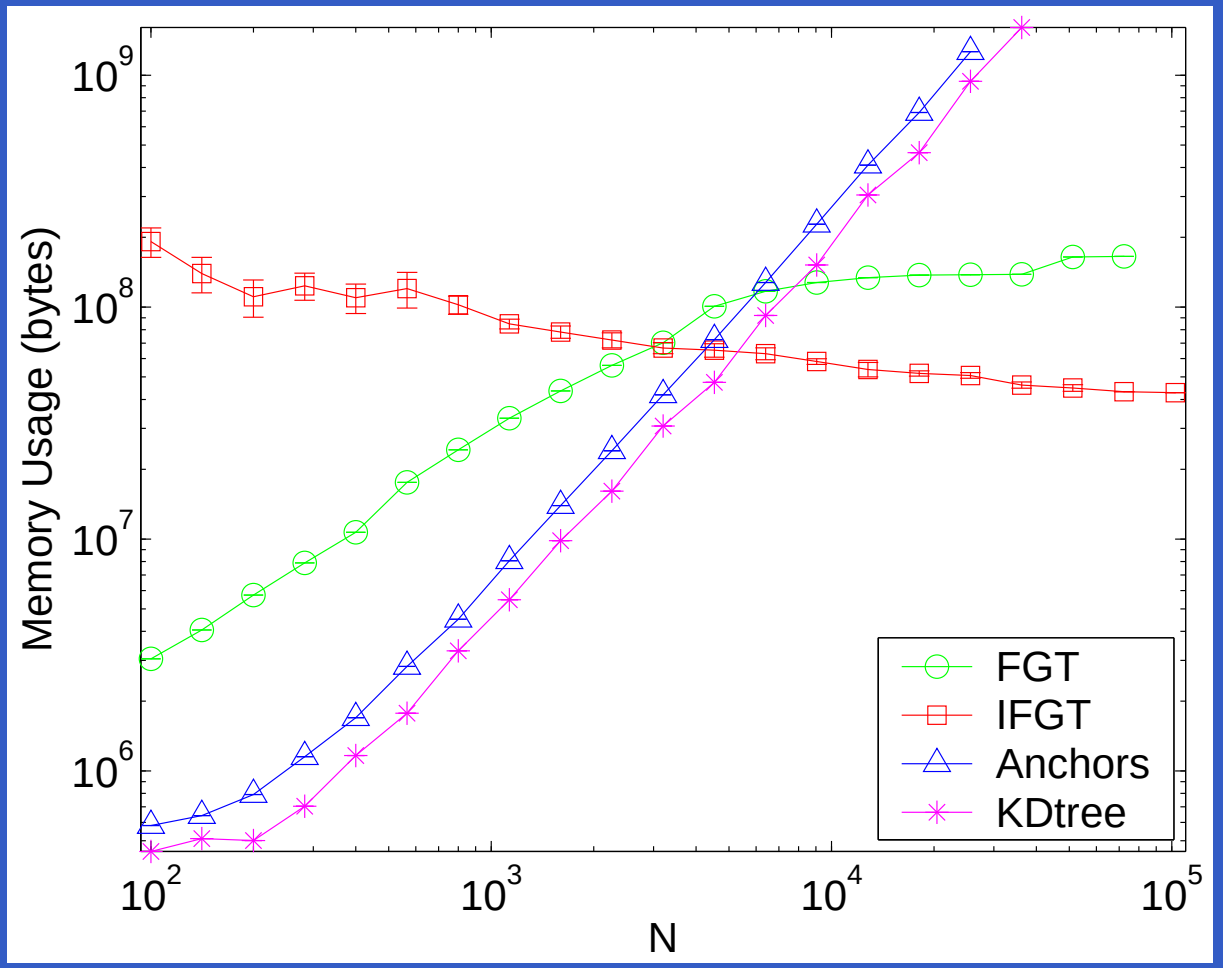
- Naive is usually fastest.
- Only FGT is faster - but only $\sim 3\times$.
- IFGT *may* become faster - after 1.5 *hours* of compute time.



Results (1): A Worst-Case Scenario

Uniformly distributed points, uniformly distributed weights, 3 dimensions, large bandwidth $h = 0.1$, $\epsilon = 10^{-6}$: Memory.

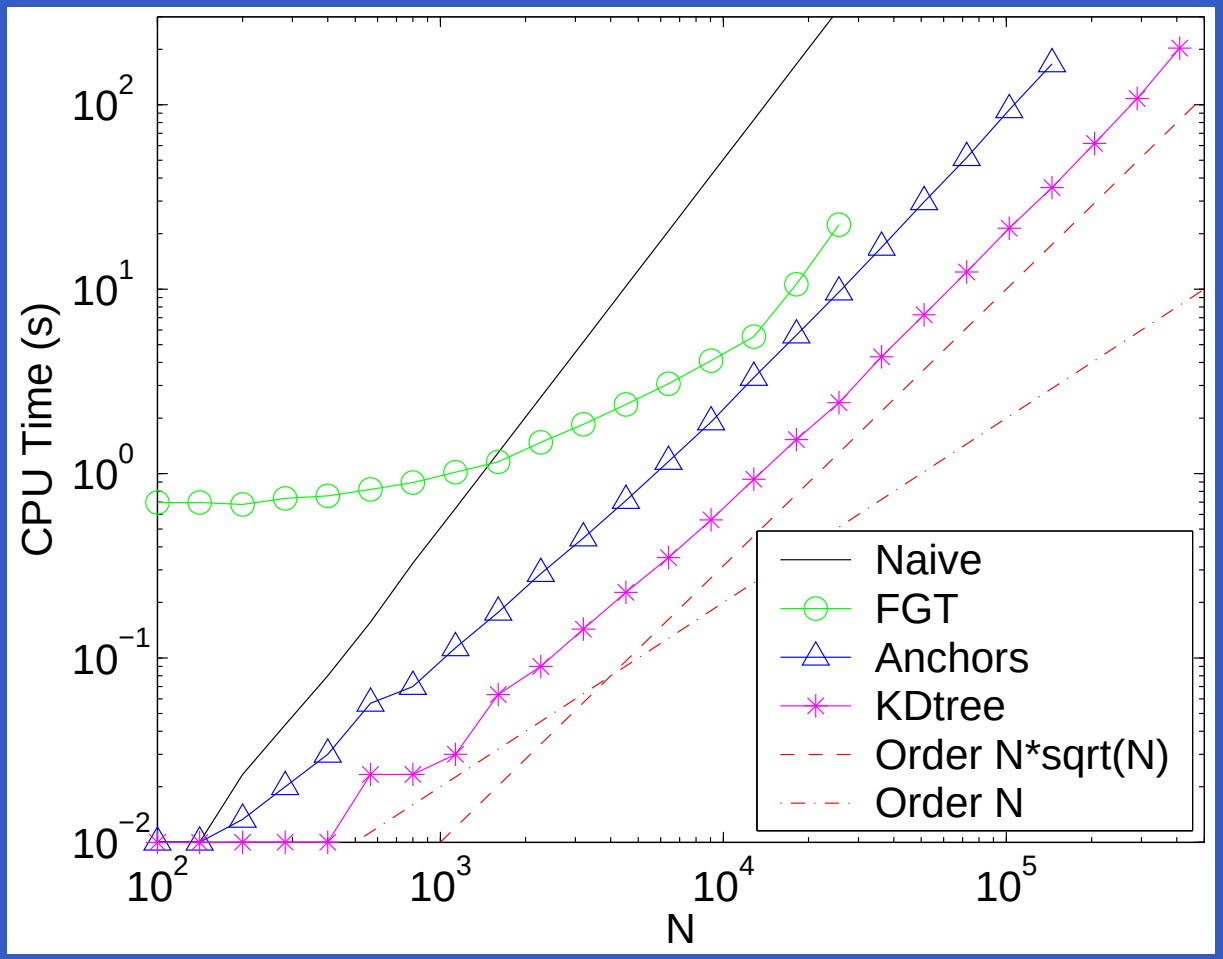
- Dual-Tree memory requirements are an issue.



Results (1): A Worst-Case Scenario

Uniformly distributed points, uniformly distributed weights, 3 dimensions, smaller bandwidth $h = 0.01$, $\epsilon = 10^{-6}$.

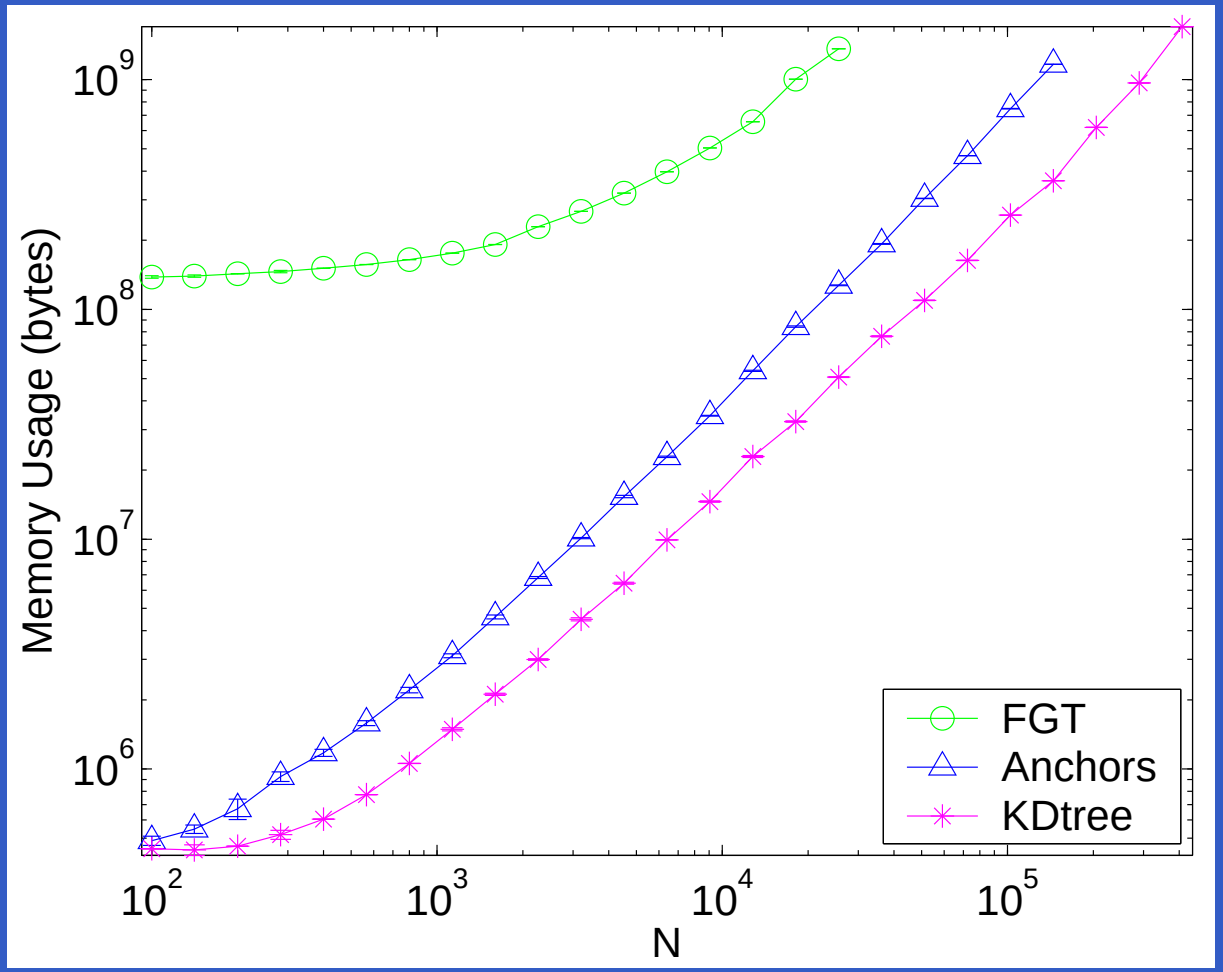
- IFGT cannot be run– more than 10^{10} expansion terms required for $N = 100$ points.
- Dual-Tree and FGT are fast, but not $O(N)$.



Results (2)

Uniformly distributed points, uniformly distributed weights, 3 dimensions, smaller bandwidth $h = 0.01$, $\epsilon = 10^{-6}$.

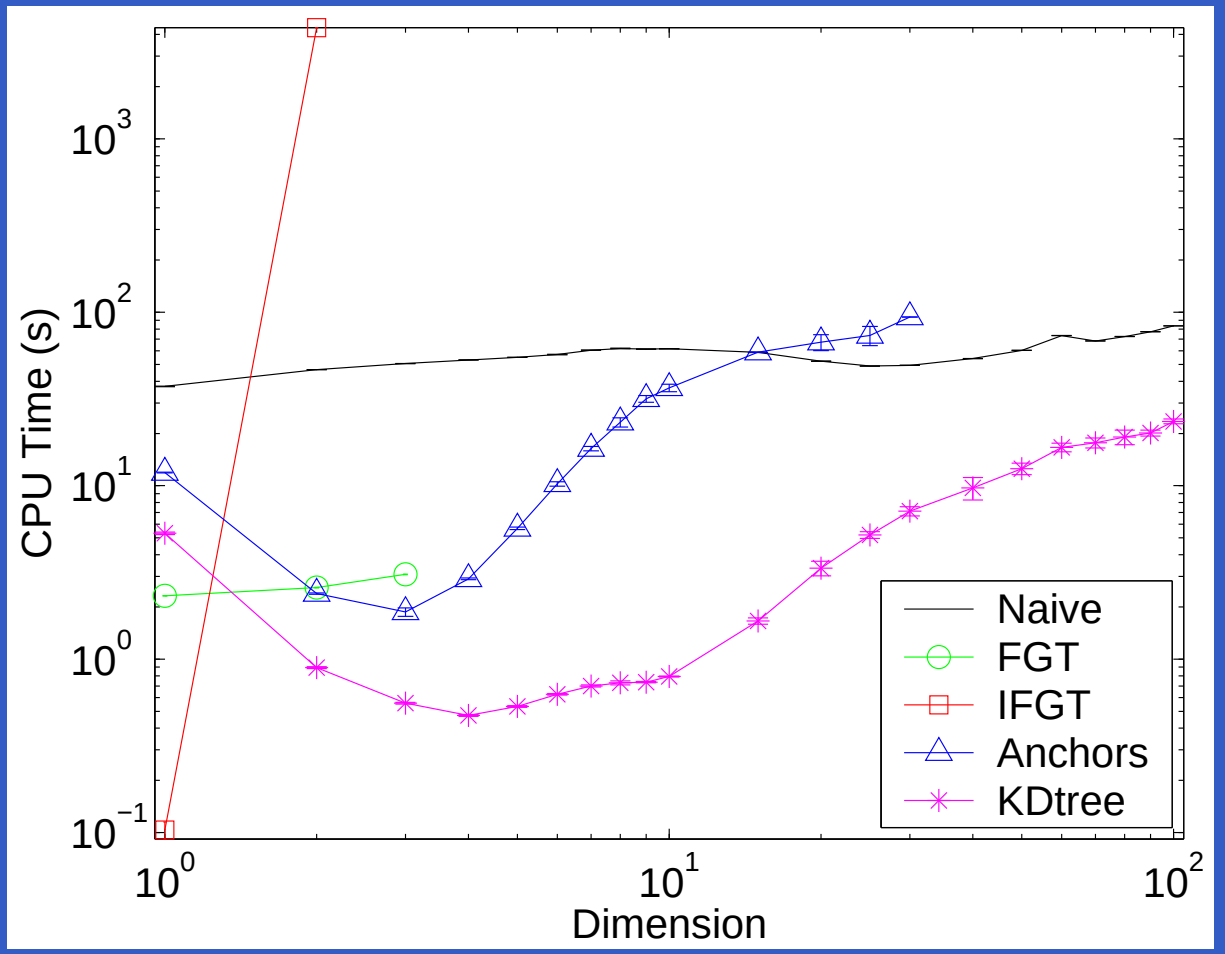
- Memory requirements are still an issue.



Results (2)

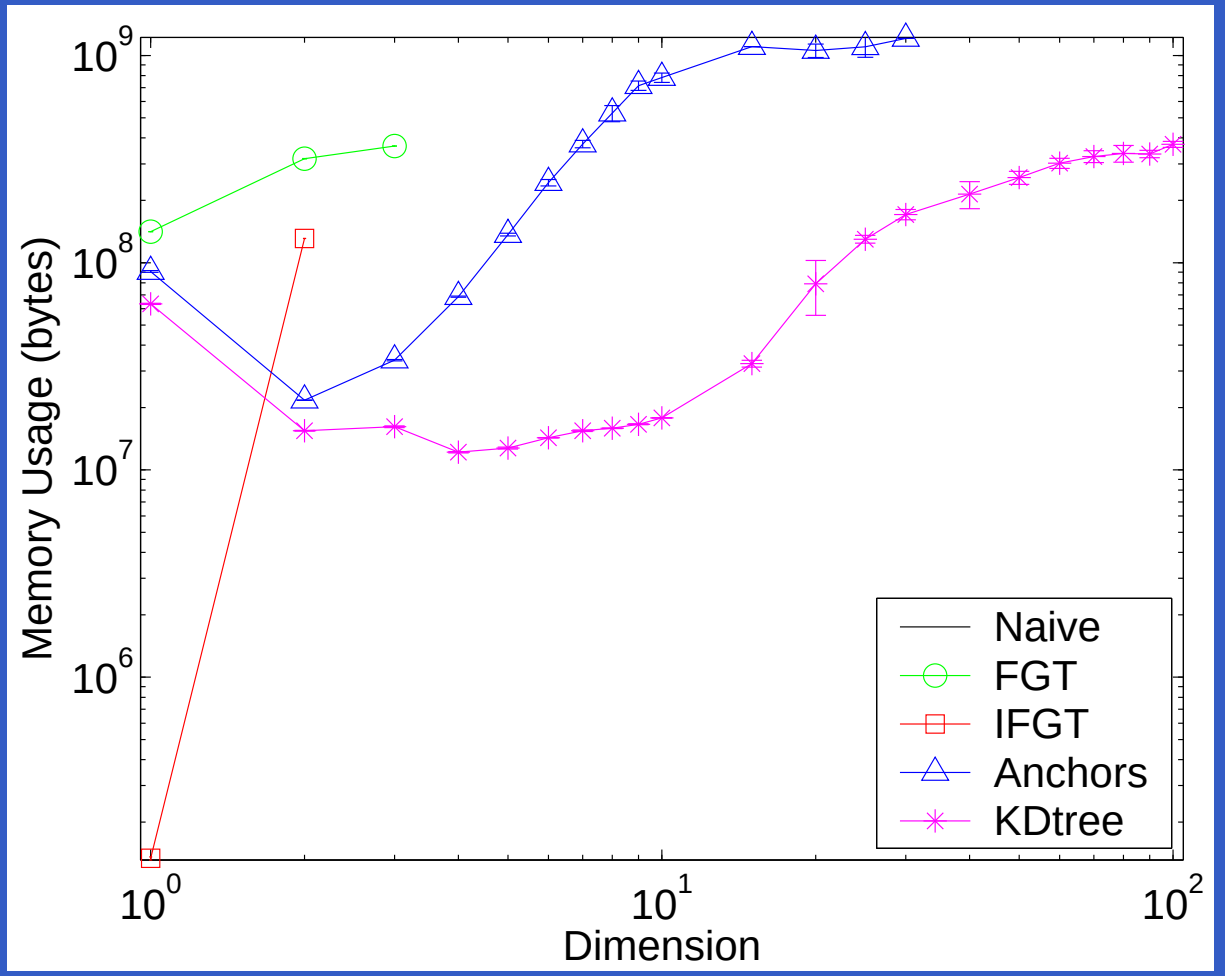
Uniform data and weights, $N = 10{,}000$, $\epsilon = 10^{-3}$, $h = 0.01$, varying dimension: CPU time.

- IFGT very fast for 1D, infeasible beyond 2D.
- KDtree, Anchors show (unexpected?) optimal behaviour around 3 or 4 dimensions.



Results (3)

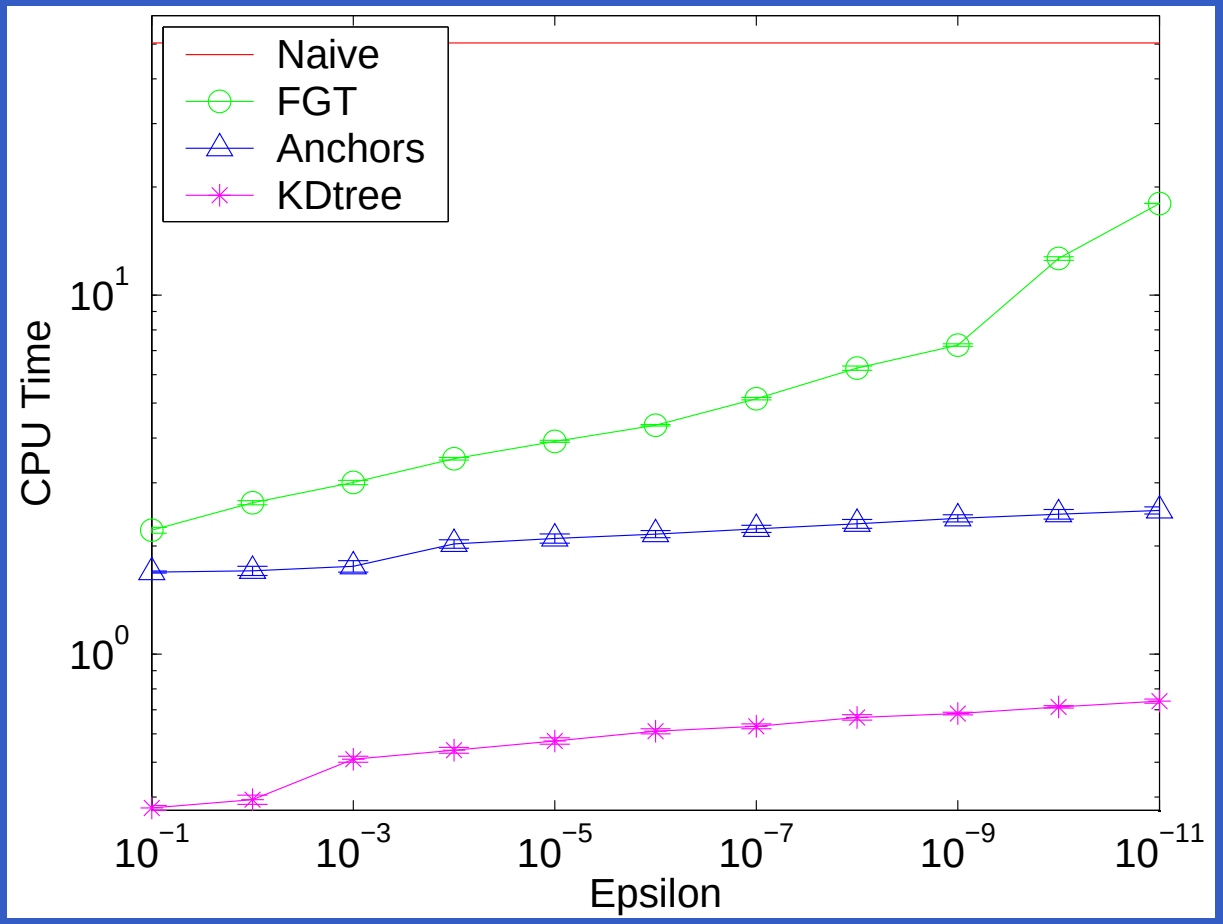
Uniform data and weights, $N = 10{,}000$, $\epsilon = 10^{-3}$, $h = 0.01$, varying dimension: Memory usage.



Results (3)

Uniform sources, uniform targets, $N = 10{,}000$, $h = 0.01$, $D = 3$, $\epsilon = 10^{-6}$: CPU time.

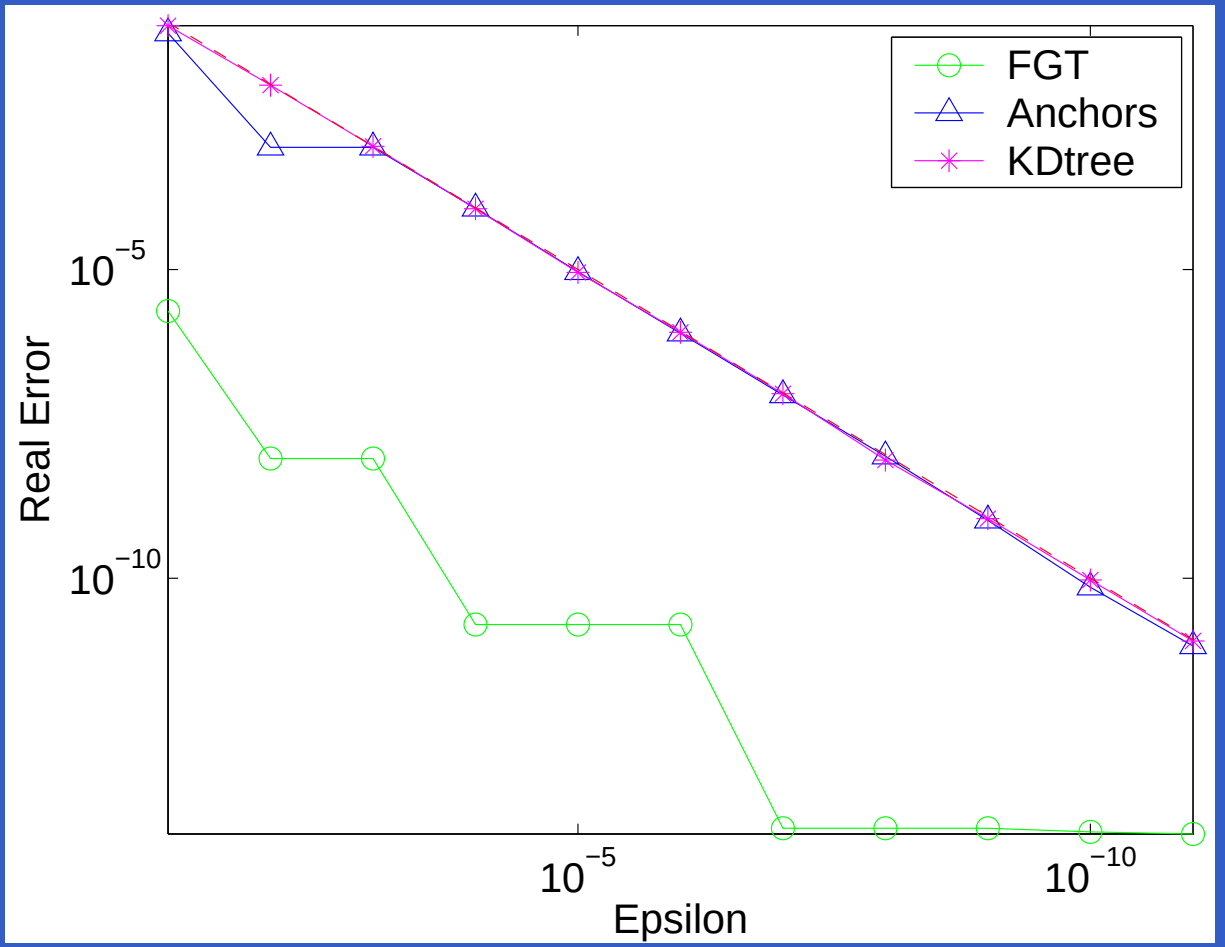
- Cost of Dual-Tree methods increases slowly with accuracy.
- FGT cost rises more quickly.



Results (4)

Uniform sources, uniform targets, $N = 10{,}000$, $h = 0.01$, $D = 3$, $\epsilon = 10^{-6}$: CPU time relative to Uniform.

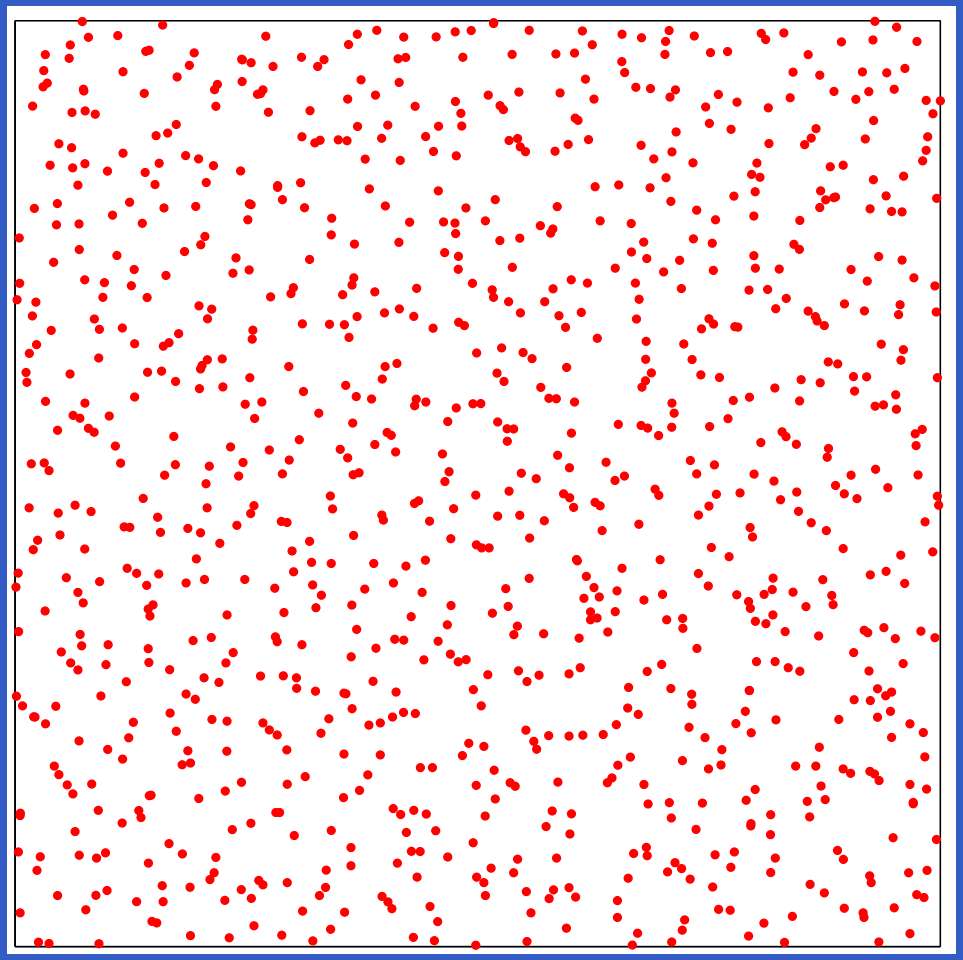
- Error of Dual-Tree methods almost exactly as large as allowed (ϵ).
- FGT (and presumably IFGT) overestimate the error–thus do more work than required.



Results (4)

Uniform data is a worst-case scenario for these methods.

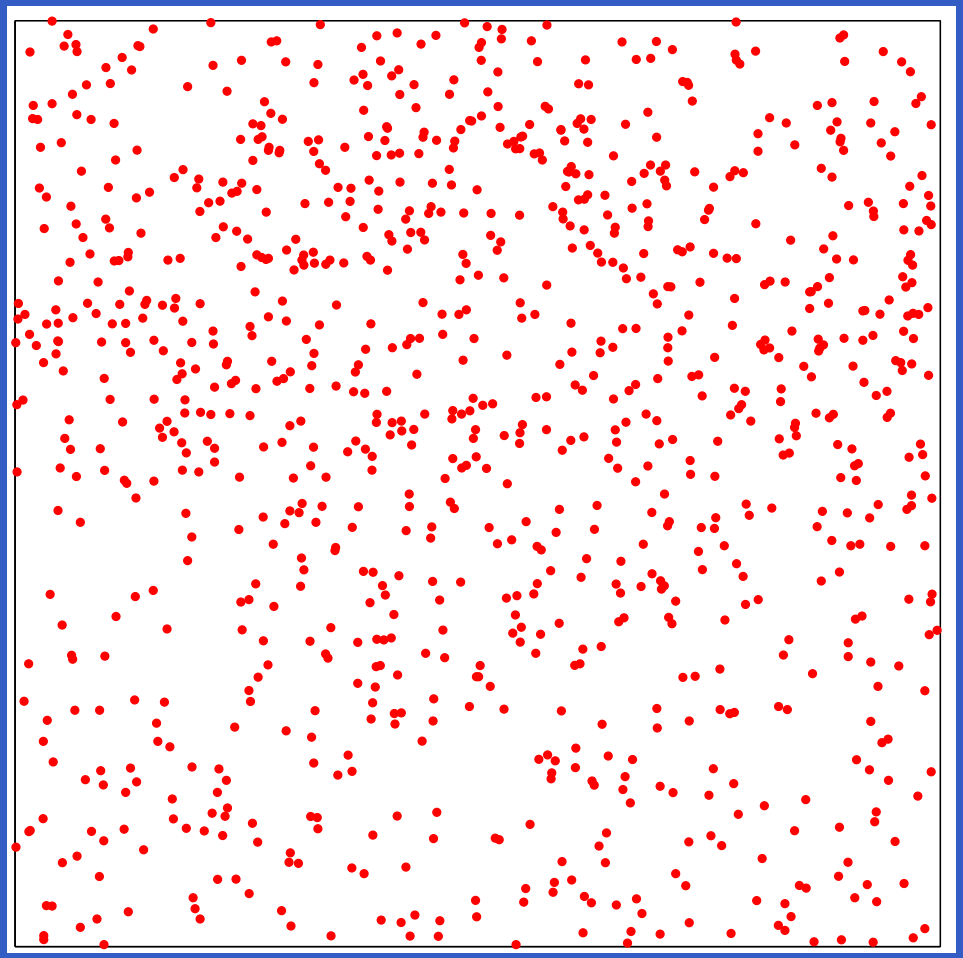
Next: clumpy data! Clumpiness = 1.0



Clumpy Data

Uniform data is a worst-case scenario for these methods.

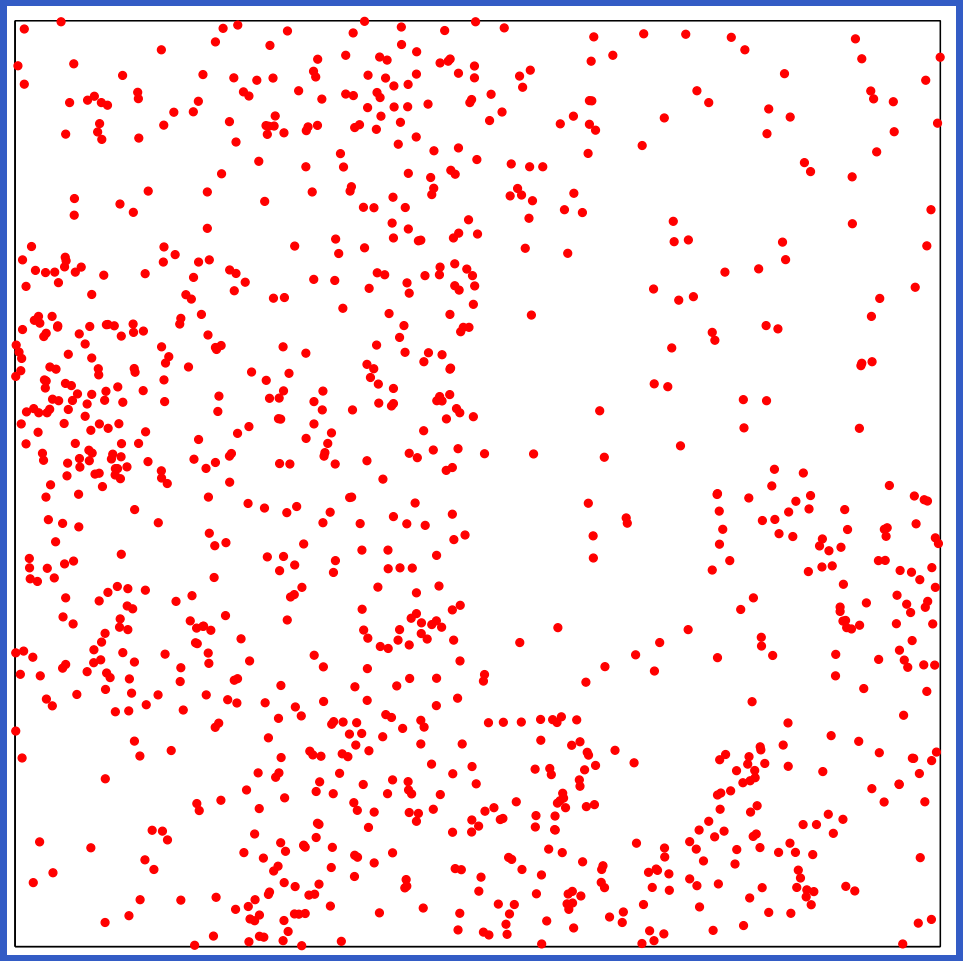
Next: clumpy data! Clumpiness = 1.1



Clumpy Data

Uniform data is a worst-case scenario for these methods.

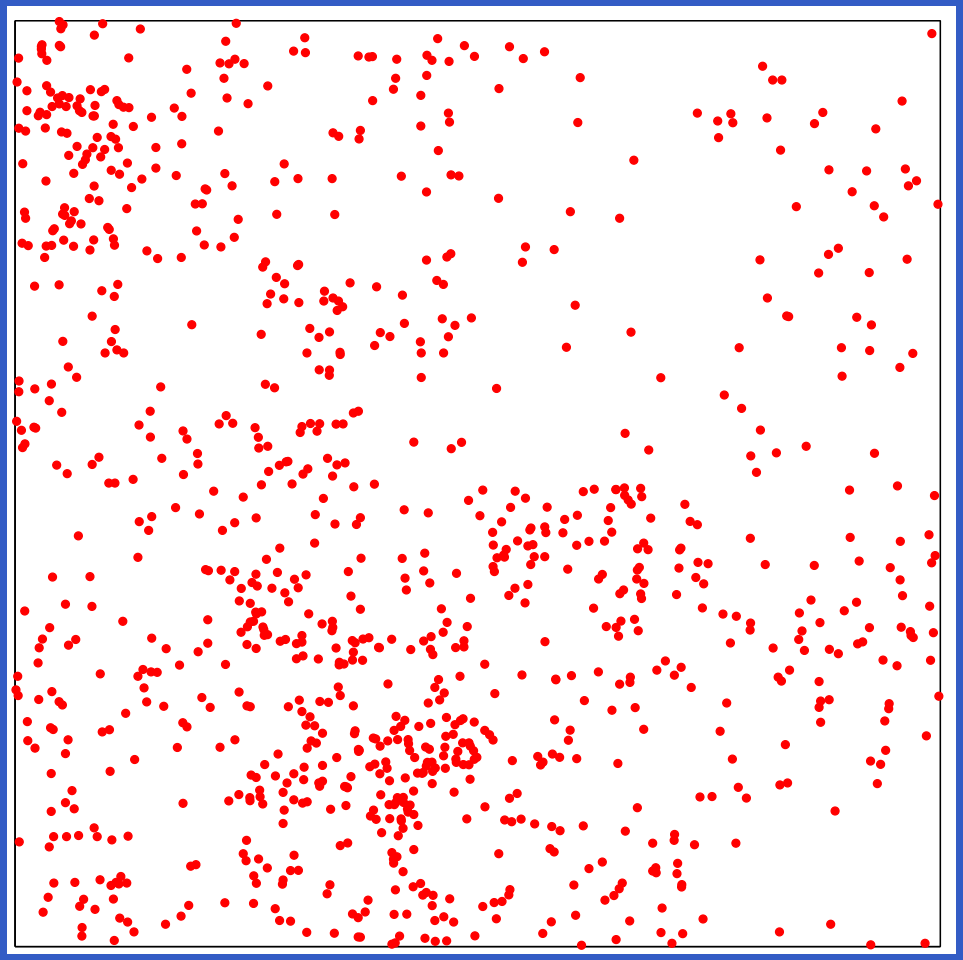
Next: clumpy data! Clumpiness = 1.2



Clumpy Data

Uniform data is a worst-case scenario for these methods.

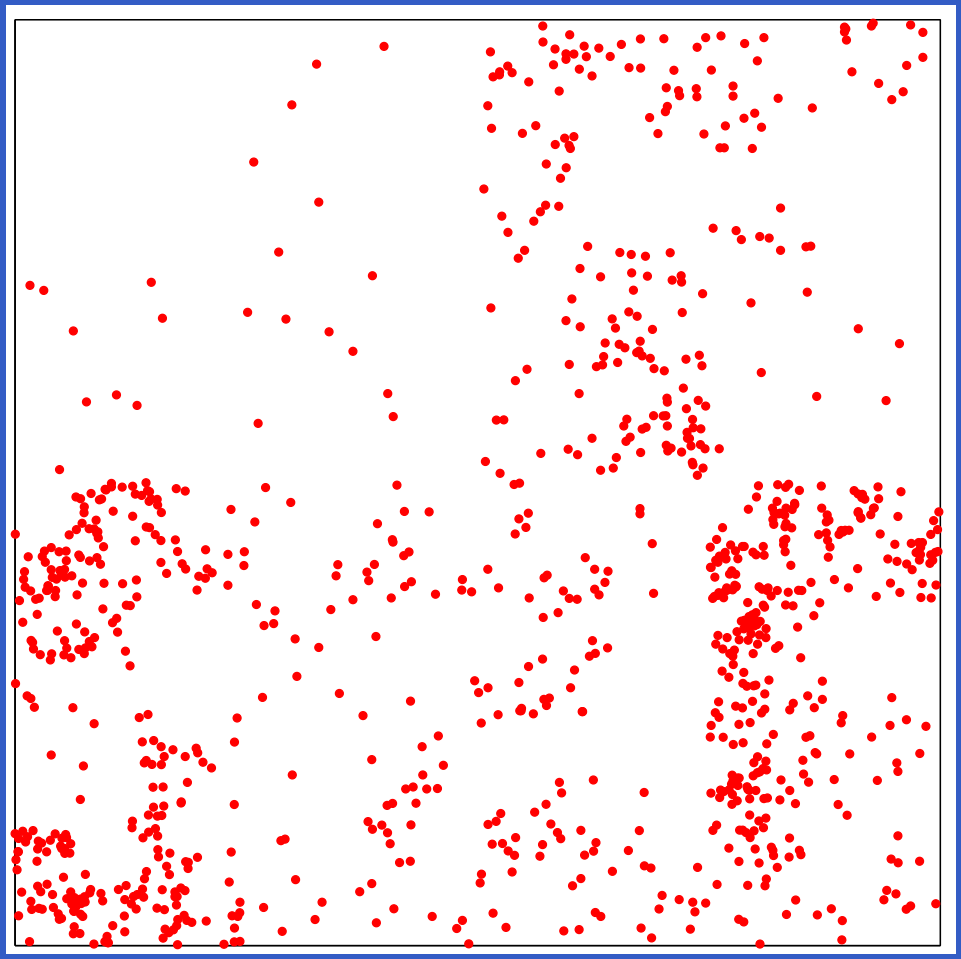
Next: clumpy data! Clumpiness = 1.3



Clumpy Data

Uniform data is a worst-case scenario for these methods.

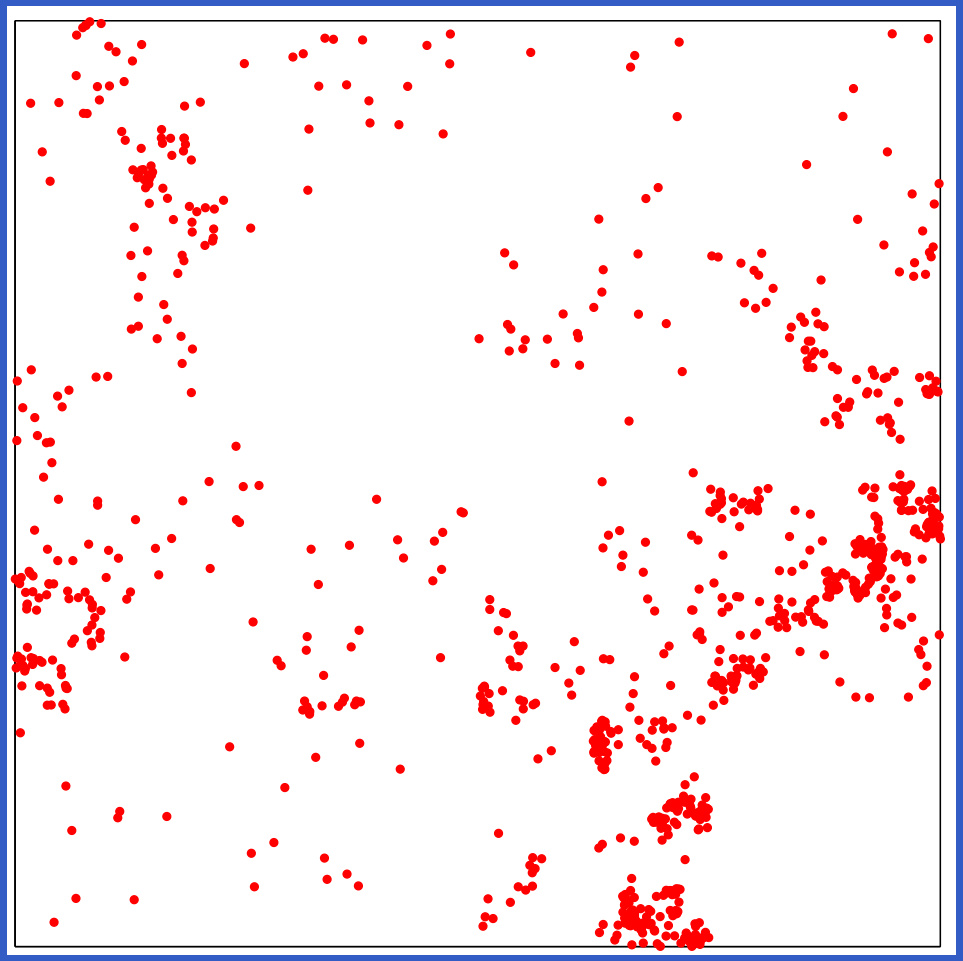
Next: clumpy data! Clumpiness = 1.5



Clumpy Data

Uniform data is a worst-case scenario for these methods.

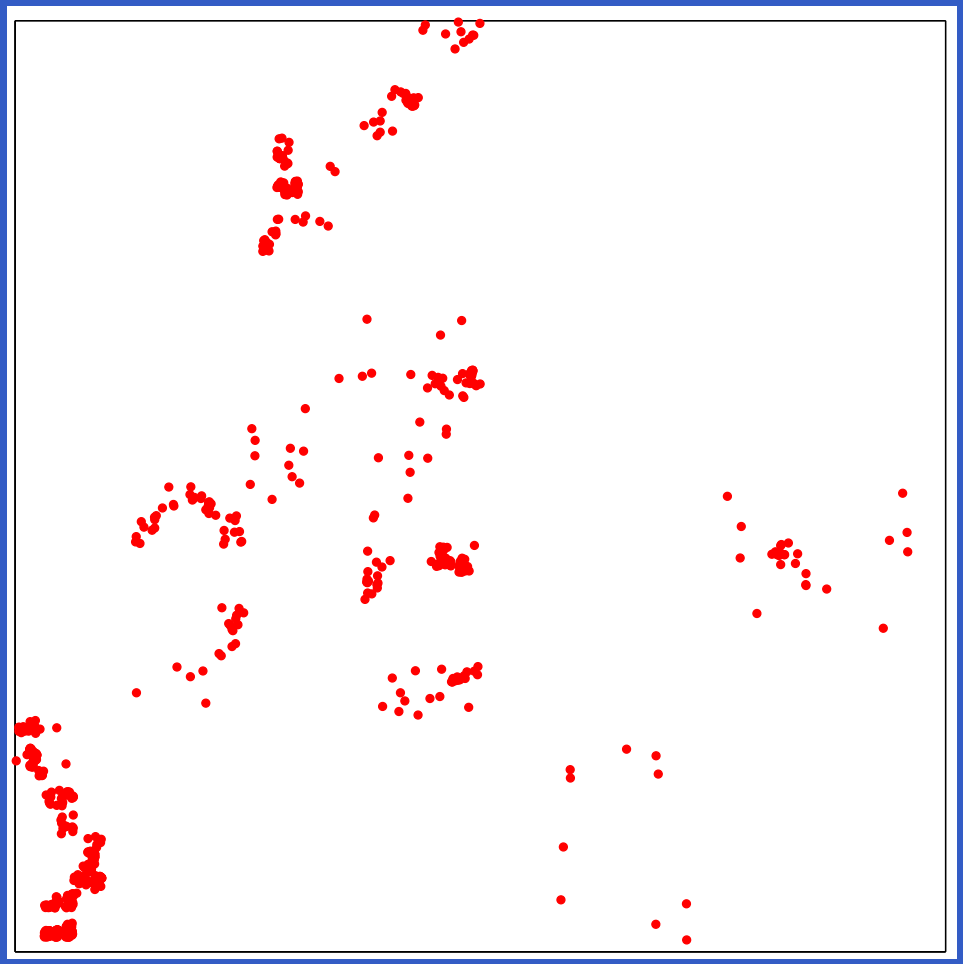
Next: clumpy data! Clumpiness = 2.0



Clumpy Data

Uniform data is a worst-case scenario for these methods.

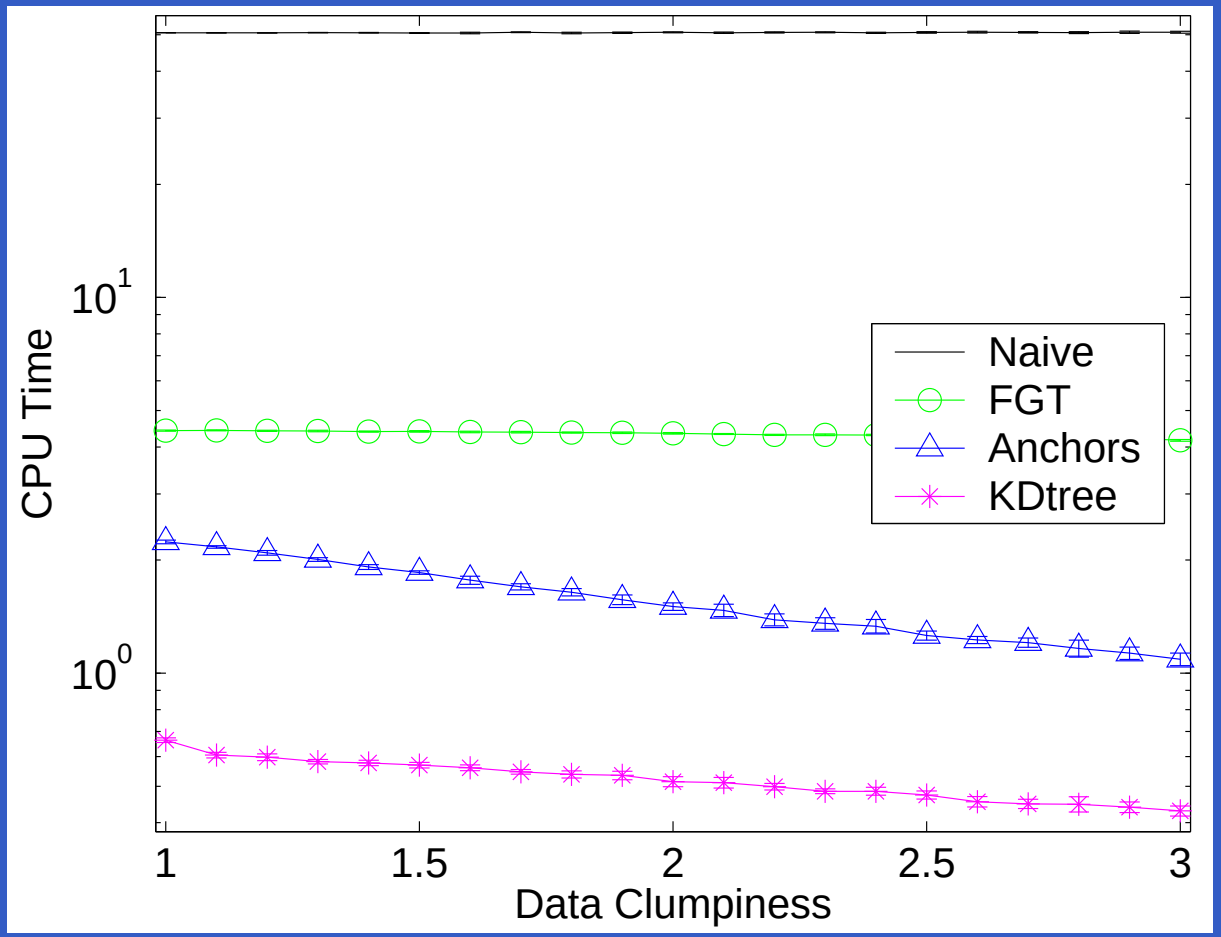
Next: clumpy data! Clumpiness = 3.0



Clumpy Data

Clumpy sources, uniform targets, $N = 10{,}000$, $h = 0.01$, $D = 3$, $\epsilon = 10^{-6}$, varying clumpiness: CPU time.

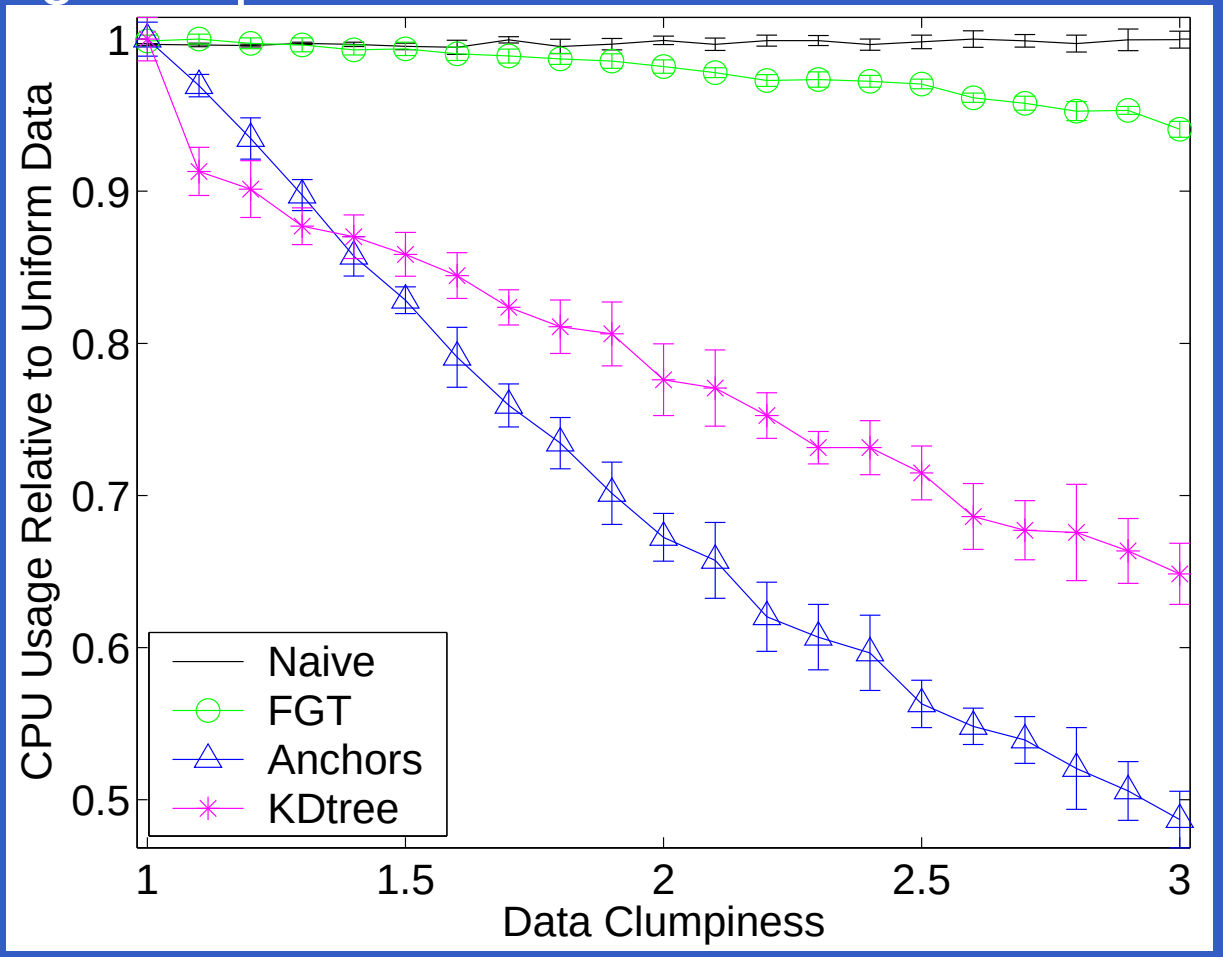
As clumpiness increases, Dual-Tree methods get faster.



Results (5): clumpy sources

Clumpy sources, uniform targets, $N = 10{,}000$, $h = 0.01$, $D = 3$, $\epsilon = 10^{-6}$, varying clumpiness: CPU time relative to Uniform.

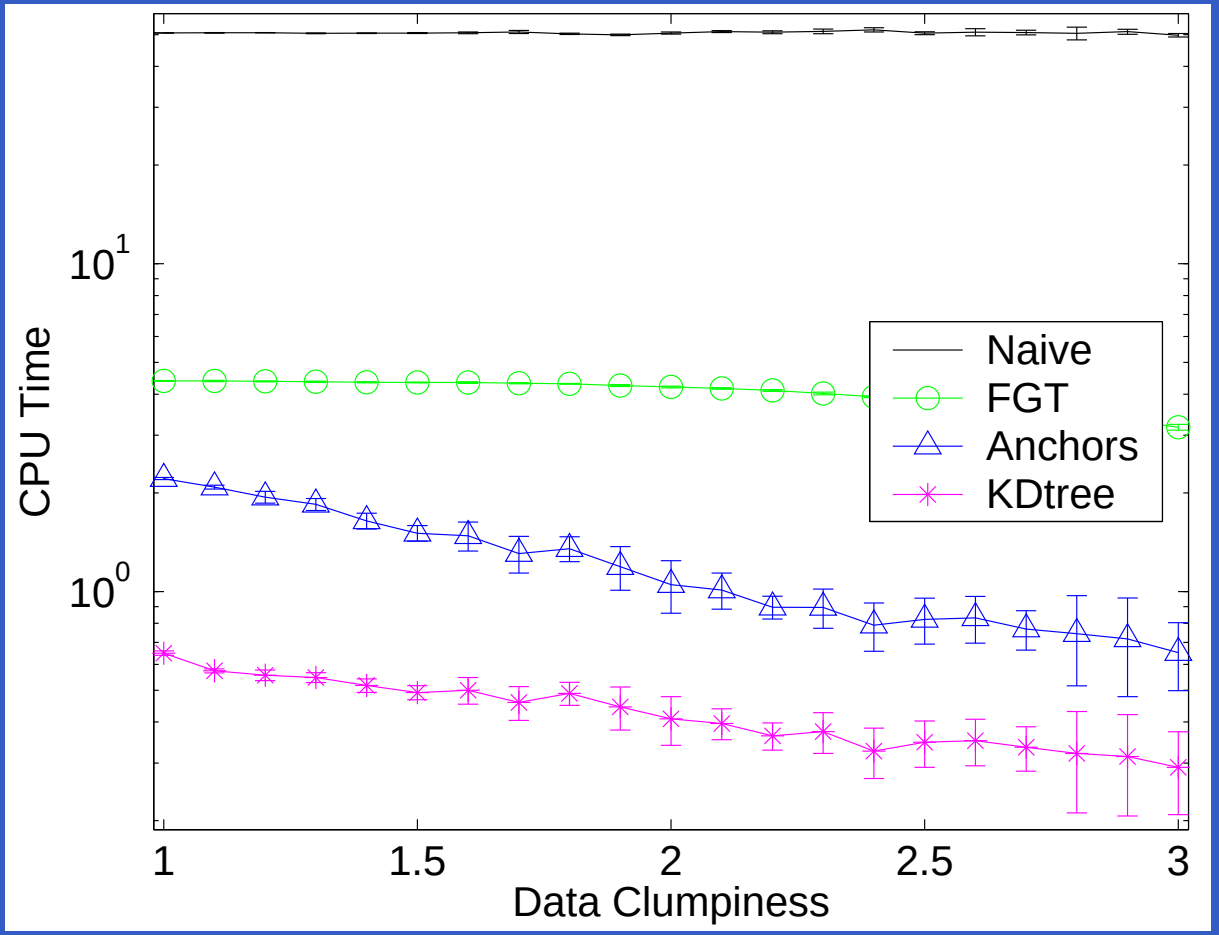
Especially Anchors.



Results (5): clumpy sources

Clumpy sources, clumpy targets, $N = 10{,}000$, $h = 0.01$, $D = 3$, $\epsilon = 10^{-6}$, varying clumpiness: CPU time.

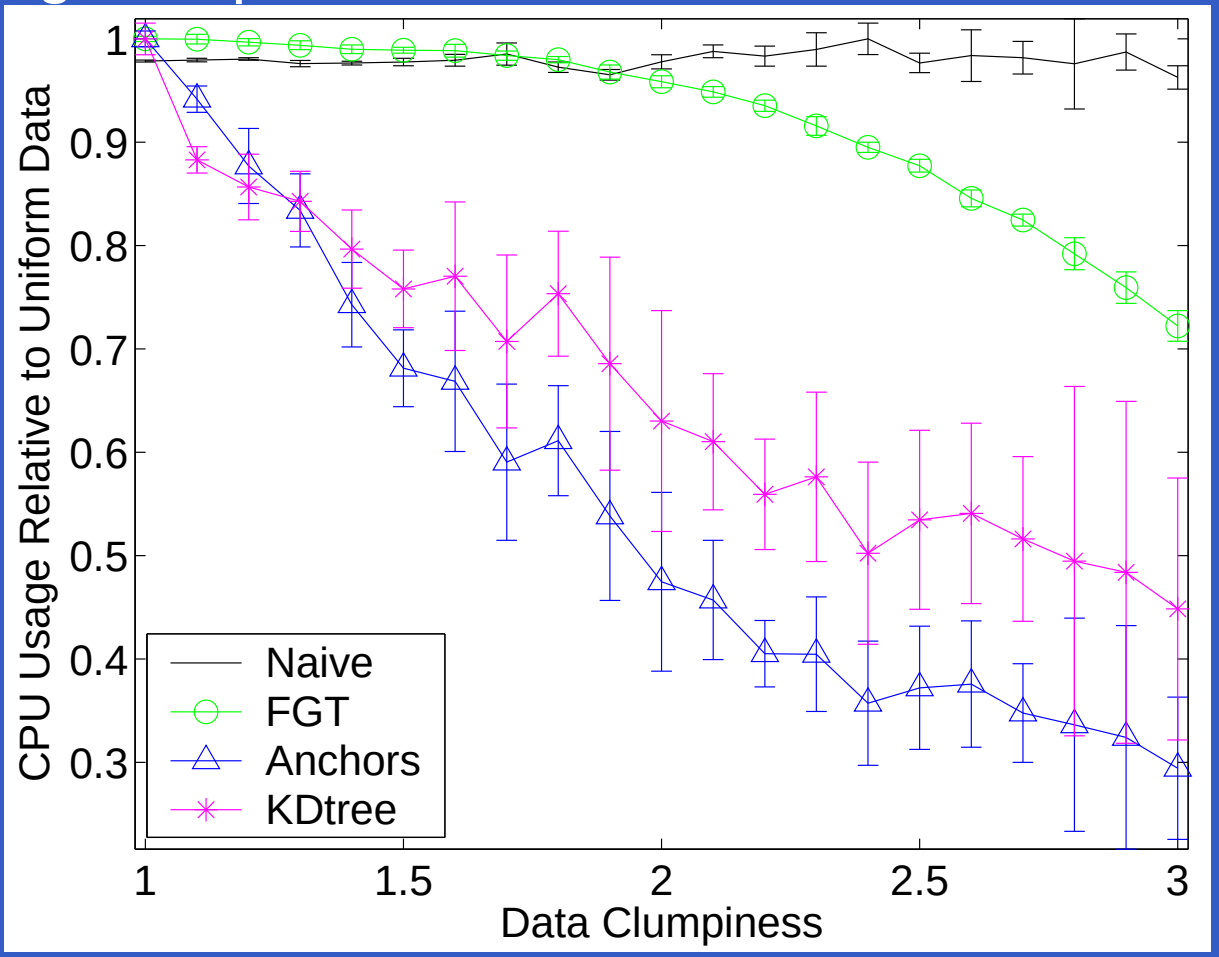
Even bigger improvements!



Results (6): clumpy sources and targets

Clumpy sources, clumpy targets, $N = 10{,}000$, $h = 0.01$, $D = 3$, $\epsilon = 10^{-6}$, varying clumpiness: CPU time relative to Uniform.

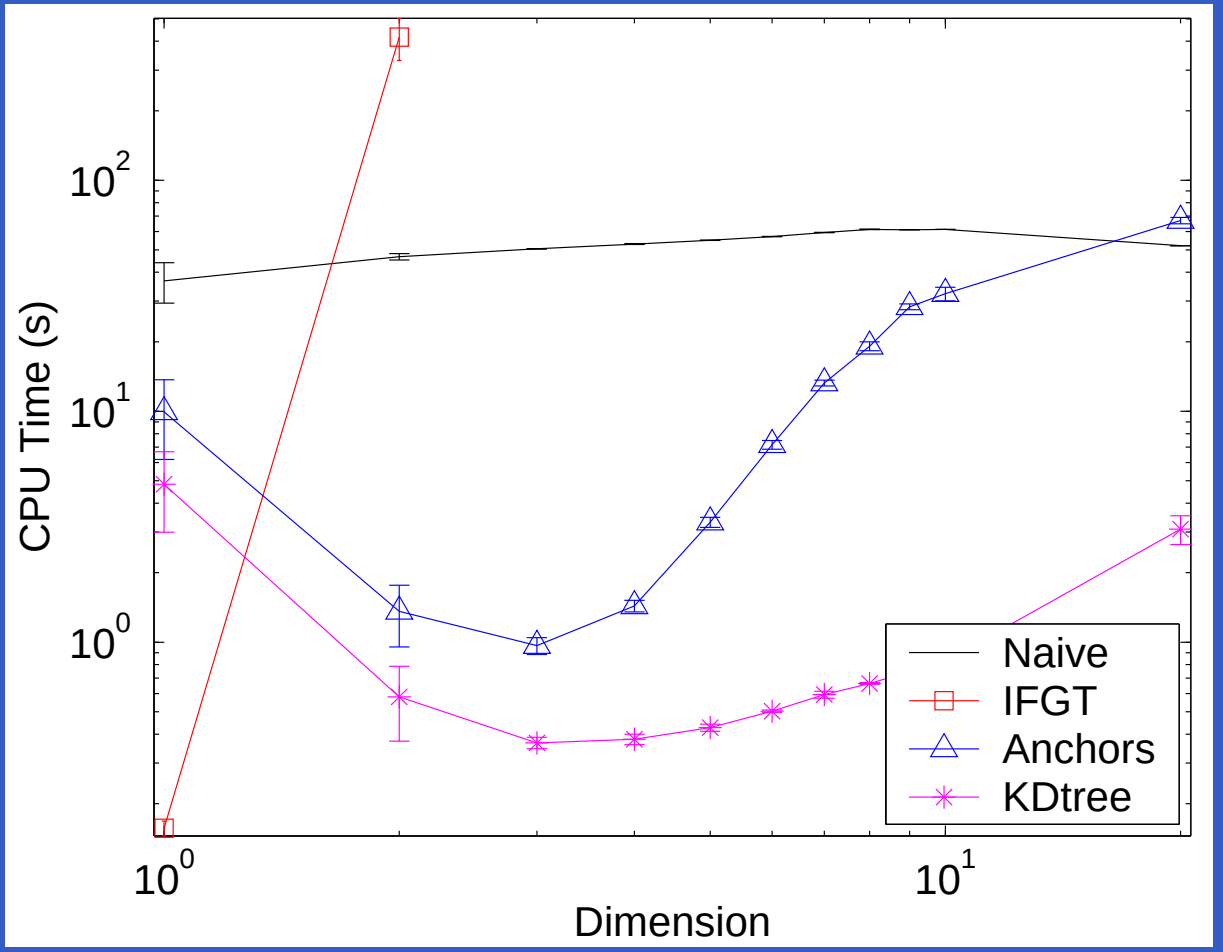
Large variance- details of particular clumpy data sets?



Results (6): clumpy sources and targets

Clumpy sources and targets ($C = 2$), $N = 10{,}000$, $h = 0.01$, $\epsilon = 10^{-3}$, varying dimension: CPU time.

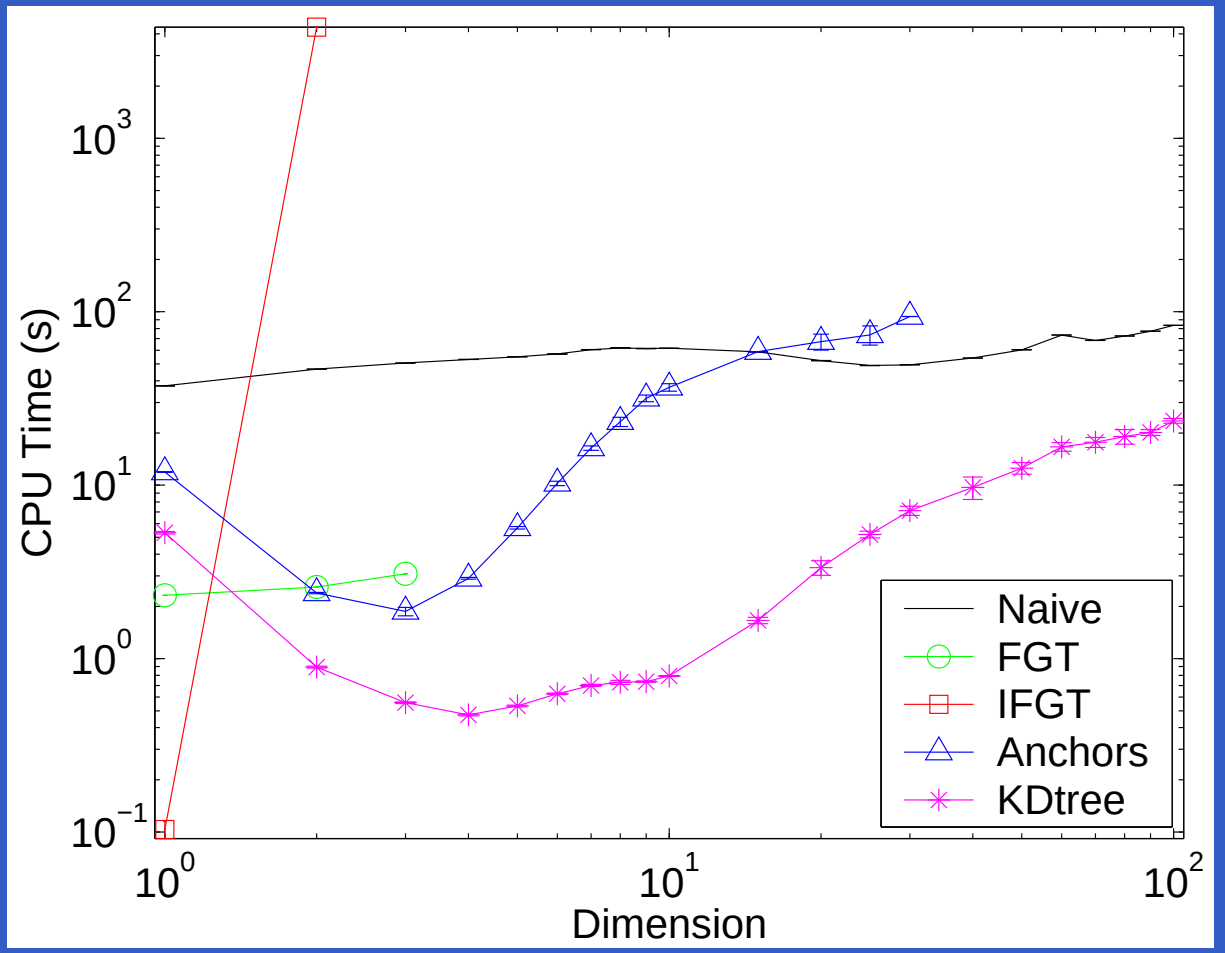
Not qualitatively different from uniform data!



Results (7): clumpy, dimensionality

Clumpy sources and targets ($C = 2$), $N = 10{,}000$, $h = 0.01$, $\epsilon = 10^{-3}$, varying dimension: CPU time.

For reference: the non-clumpy results.



Results (7): clumpy, dimensionality

- Synthetic-data tests; each algorithm is required to guarantee results within a given error tolerance.
- IFGT:
  - We devised a method of choosing parameters– a different method might work better.
  - The error bounds seem to be very loose, so it does *much* more work than necessary.

Summary (1)

Dual-Tree:

- Work well when either the kernel is highly local (small bandwidth) or when the data has strong structure.
- Work well across a wide range of error tolerances– give errors that are close to the estimate.
- Memory requirements are an issue (some heuristics could be used).
- In these tests, Anchors Hierarchy doesn't outperform KDtree, though it improves significantly with clumpiness.

Summary (2)

And Now For Something Slightly Different:

Max-Kernel

- Given:
  - N target points (y_j)
  - M source points (x_i) with weights w_i
- Compute, for each y_j:

$$f_j^{MAX} = \max_i w_i K(x_i, y_j)$$

- Cost: $O(MN)$

The Problem

- Given:
  - N target points (y_j)
  - M source points (x_i) with weights w_i
- Compute, for each y_j:

$$f_j^{MAX} = \max_i w_i K(x_i, y_j)$$

- Cost: $O(MN)$
- Applications:
  - maximum *a-posteriori* belief propagation
  - Viterbi algorithm for chains
  - (MAP) particle smoothing

The Problem

1. **Distance Transform**

The Methods

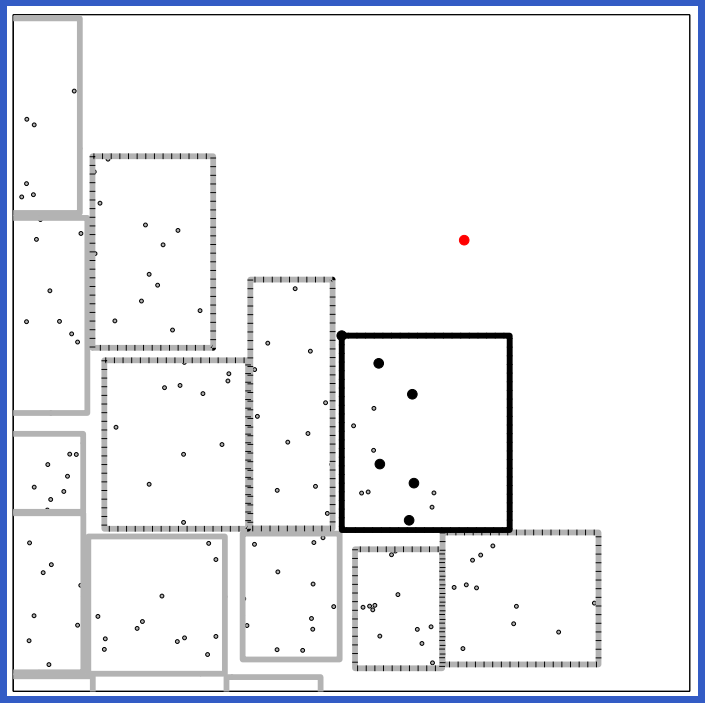
1. **Distance Transform**
   - as previously presented
   - can be extended to handle Monte Carlo grids in 1D
     - increases cost to $O(M \log M + N \log N)$

The Methods

1. **Distance Transform**
   - as previously presented
   - can be extended to handle Monte Carlo grids in 1D
     - increases cost to $O(M \log M + N \log N)$
2. **Dual-tree algorithm**

The Methods

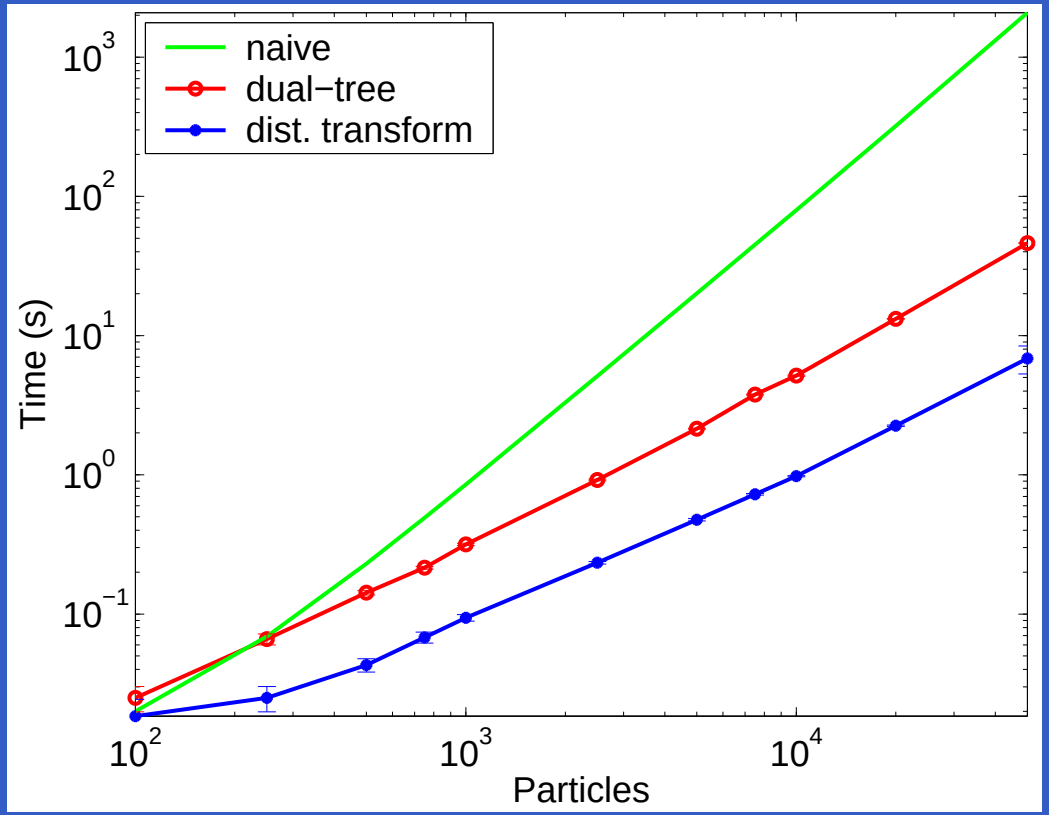
1. **Distance Transform**
   - as previously presented
   - can be extended to handle Monte Carlo grids in 1D
     - increases cost to $O(M \log M + N \log N)$
2. **Dual-tree algorithm**
   - "bound and prune" recursion
   - details: Klaas, Lang, de Freitas. "Fast maximum *a-posteriori* inference in Monte Carlo state spaces". AISTATS 2005 (to appear).



The Methods

- MAP particle smoothing
- Non-linear, multi-modal time series
- Note **log-log** scale

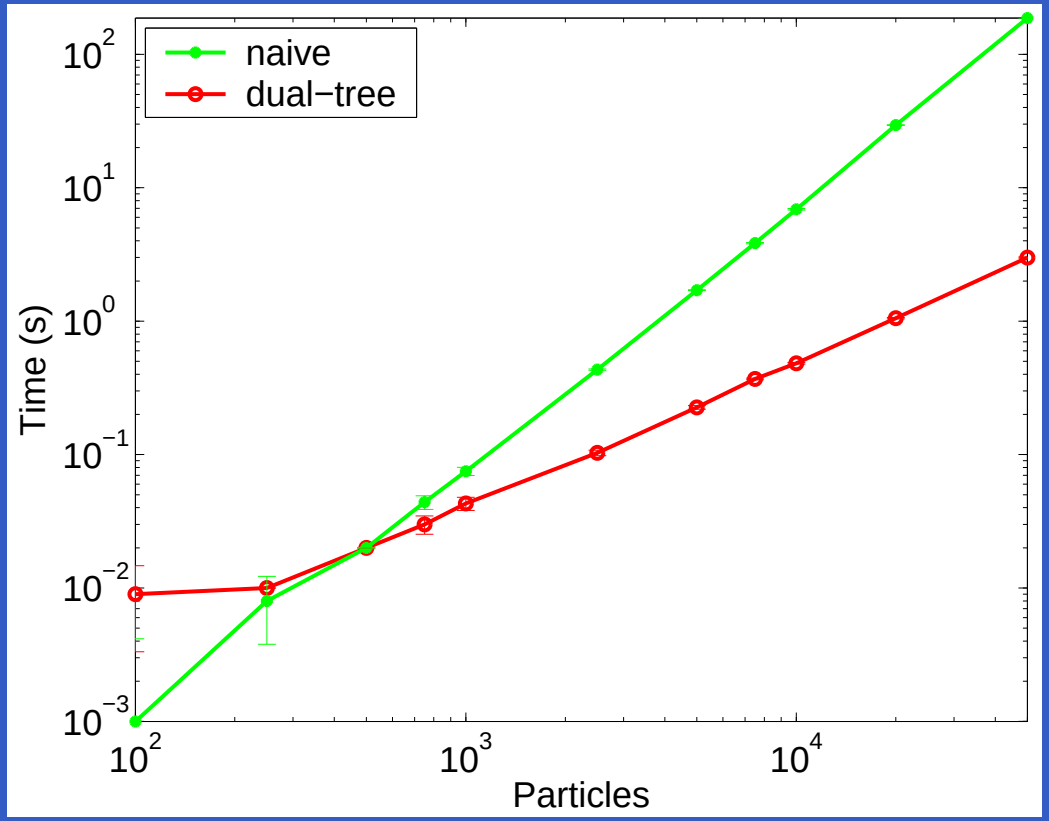
- Both beat naïve by orders of magnitude
- Dist. trans. 2-3$\times$ faster than dual-tree
- Similar asymptotic growth
- Clearly, dist. trans. should be used when possible!



1D time series

- Particle-filter based beat tracker
- MAP smoothing on a 3D Monte-Carlo state space
  - distance transform cannot be used

- Dual-tree is faster after $10ms$ compute time
- Dual-tree exhibits asymptotic $O(N \log N)$ growth
- Takes seconds rather than days to process a song.



Applied example: beat-tracking

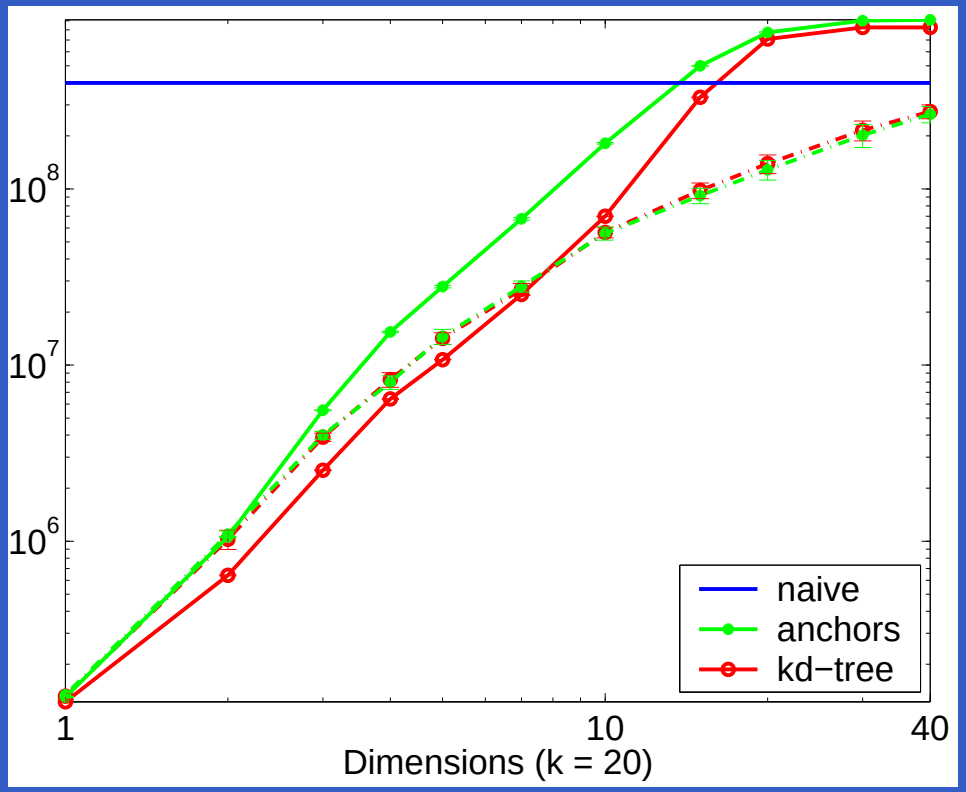
- The behaviour of dual-tree algorithms as N grows is well-understood
- What about other factors?

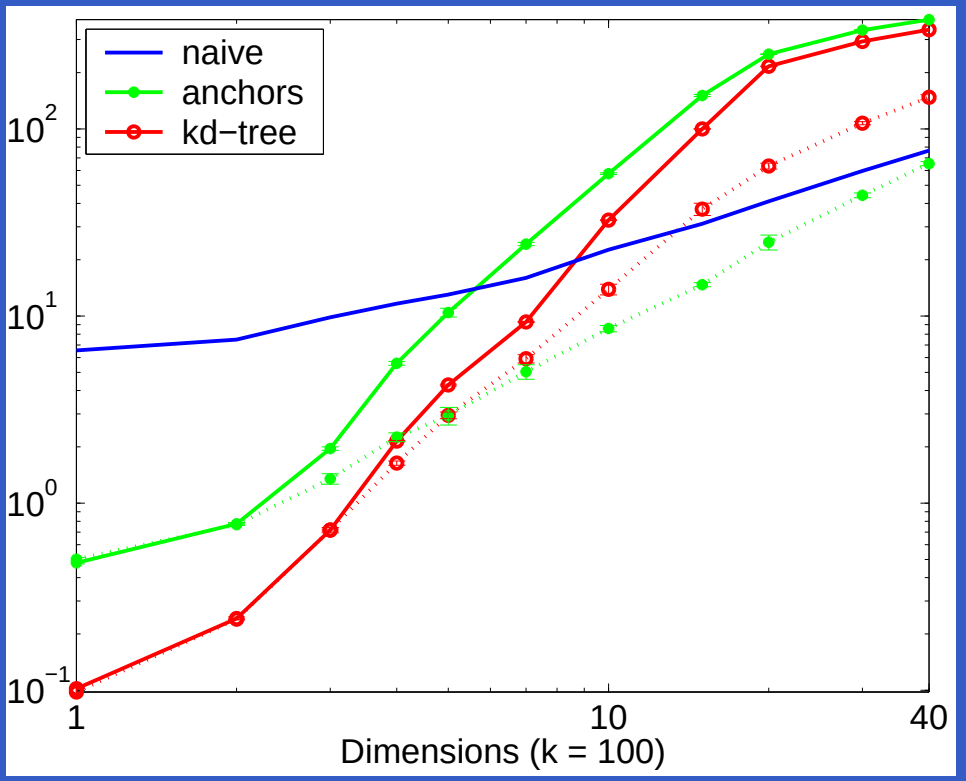
Other factors: dimensionality

- The behaviour of dual-tree algorithms as N grows is well-understood
- What about other factors?
- Synthetic test:
  - 20,000 data points (fixed)
  - Gaussian kernel with fixed bandwidth
  - distribution: uniform, clustered
  - clustered data formed by drawing from k Gaussians
  - $k = 4$ (dash), 20 (dash-dot), 100 (dotted) uniform (solid)
  - kd-trees (red) vs. metric trees (green)

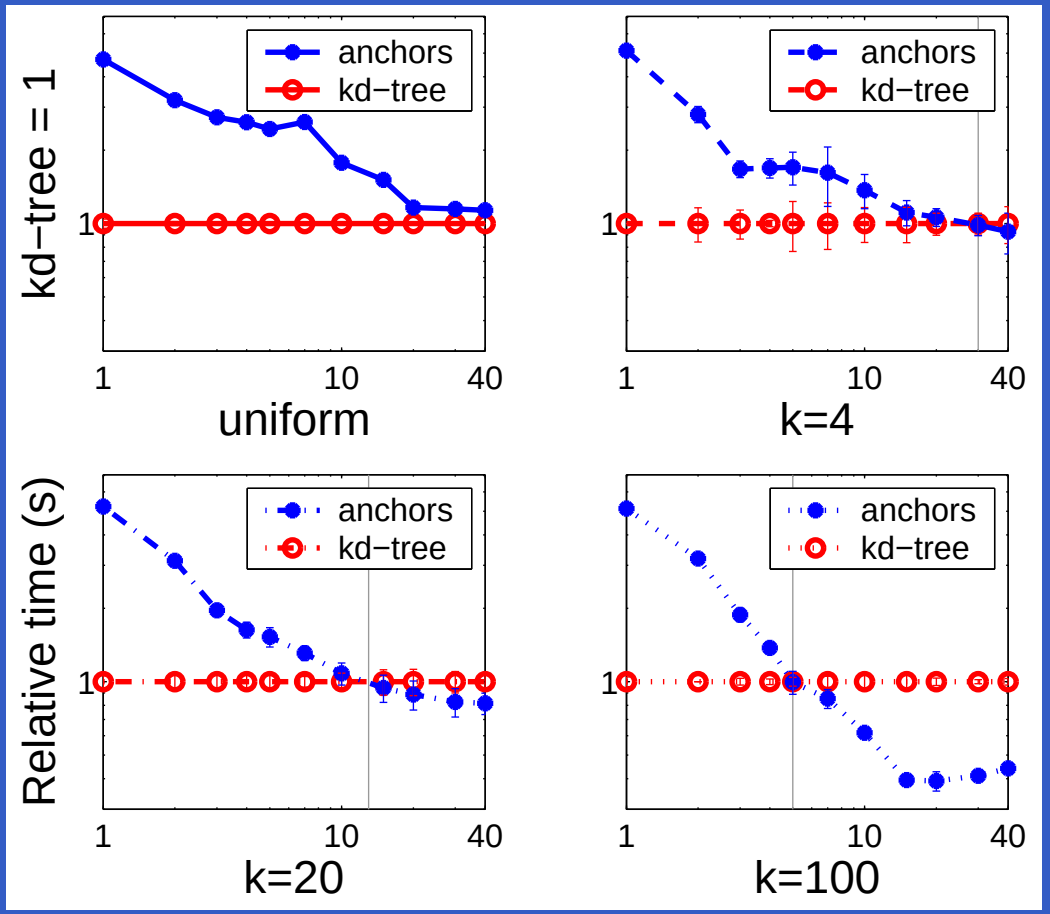
Other factors: dimensionality

- Two examples: distance computations (L); time (R)
- Dual-tree methods can be slower than naïve, and this is due to inherent complexity, not just high constants.
  - ie., it uses $O(N^2)$ distance computations.





Dimensionality (cont.)



- Clustering is necessary for metric trees to be effective.

Dimensionality (relative)

- Distance transform and dual-tree methods are fast

Summary

- Distance transform and dual-tree methods are fast
  - ...but dual-tree has more overhead.

Summary

- Distance transform and dual-tree methods are fast
  - ...but dual-tree has more overhead.
  - Use the distance transform when:
    - kernel is $e^{-\|x-y\|^2}$ or $e^{-\|x-y\|}$ (or others?)
    - data is one dimensional, or lies on a regular grid.

Summary

- Distance transform and dual-tree methods are fast
  - ...but dual-tree has more overhead.
  - Use the distance transform when:
    - kernel is $e^{-\|x-y\|^2}$ or $e^{-\|x-y\|}$ (or others?)
    - data is one dimensional, or lies on a regular grid.
- Although we focus on performance as N grows, it is the “constants” that really matter
  - these are determined by the data distribution, the kernel, and the spatial index.
  - **huge** potential for future investigation.

Summary

Thanks!

Time for Questions!

- Clumpy Data generation
- Choosing IFGT params

Q&A

We generate clumpy data with clumpiness C by recursively distributing points into sub-boxes such that the occupancies satisfy:

$$\sum_{i=1}^{n} N_i = N$$

$$\text{var}\left(\{N_i\}\right) = (C-1)\,\text{mean}\left(N_i\right)^2$$

This describes the width of the distribution of ‘mass’ among boxes.
Recurse until $N \leq 10$.

Clumpy Data (back)

K : number of source clusters

r_y : influence radius of clusters

p : number of expansion terms

We choose a maximum number of clusters K^*. The complexity is NK, so to be $O(N)$, K^* must be a constant.

In these tests, we instead set $K^* = \sqrt{N}$, since we tested across orders of magnitude.

Choosing IFGT Parameters (back)

Four constraints:

$\boldsymbol{C_1}$: outside-of-influence-radius error $E_C \leq \epsilon$

$\boldsymbol{C_2}$: truncation error $E_T \leq \epsilon$

$\boldsymbol{C_3}$: $K \leq K^*$

$\boldsymbol{C_4}$: $\left(\frac{r_x r_y}{h^2}\right) \leq 1$

the first three are hard, the fourth is soft (helps convergence).

(Each source point contributes to error through either E_C or E_T)

Choosing IFGT Parameters (2)

for $k = 1$ to K^*:

run k-centers algorithm.

find largest cluster radius r_x.

using $r_y = r_y(\text{ideal})$, compute C_1, C_4.

if C_1 AND C_4 satisfied:

break

if $k < K^*$: *// C_4 can be satisfied.*

set $r_y = \min(r_y)$ such that C_1 AND C_4.

else: *// C_4 cannot be satisfied.*

set $r_y = \min(r_y)$ such that C_1.

set $p = \min(p)$ such that C_2.

Choosing IFGT Parameters (3)