Cache is King: Presenting `Corrfunc`– a Suite of Fast Correlation Function Codes

Manodeep Sinha[a]

[a]*6902 Stevenson Center, Department of Physics & Astronomy, Vanderbilt University, Nashville, TN 37235*

Abstract

In the era of 'Big Data', we need special computing techniques to both process the raw data as well generate and evaluate models to test our theories. In particular, large galaxy surveys like the Sloan Digitial Sky Survey requires computing a variety of n-point statistics, e.g., the 2-pt projected correlation function, $w_p(r_p)$. Since the clustering measure on the data is only done once, time taken to do such a measurement is not a bottleneck. However, modeling the observed galaxy distribution correctly *requires* a Monte-Carlo Markov Chain (MCMC) and repeated measurements of one or multiple clustering statistics. In order to efficiently compute *any* clustering statistic, we need to write code that is tuned for the underlying hardware. For modern CPUs, such tuning involves proper utilization of the cache hierarchy *and* vectorization with Single Instruction Multiple Data (SIMD) capable wide vector registers. In this paper, I present `Corrfunc`– a suite of OpenMP parallelized, clustering codes that exploit current CPU micro-architecture with hand-written Advanced Vector Extensions (`AVX`) and Streaming SIMD Extensions (`SSE`) intrinsics. Given a 3-D distribution of points in a cartesian system (i.e., generated from cosmological simulations), `Corrfunc` can compute spatial 3-D correlation functions $DD(r)$, $\xi(r)$, $\xi(r_p, \pi)$; 2-point projected correlation function $w_p(r_p)$; counts-in-spheres $pN(r)$. For a 3-D particle distribution on the sky (i.e., mock galaxy catalogs, observed galaxies), the codes can compute projected correlation function $DD(r_p, \pi)$, angular correlation function $DD(\theta)$ as well as $pN(r)$. By design, `Corrfunc` is highly optimized and can compute $w_p(r_p)$ for $\mathcal{O}(1$ million$)$ galaxies in ~ 6 seconds on a post-2011 CPU, which is a factor of few faster than existing public correlation

Email address: `manodeep.sinha@vanderbilt.edu` (Manodeep Sinha)

function routines. `Corrfunc` is publicly available at `https://github.com/manodeep/Corrfunc/` while benchmarks with existing codes are available at `https://gist.github.com/manodeep/cffd9a5d77510e43ccf0`.

Keywords: methods: numerical, (cosmology:) large-scale structure of universe,

1. INTRODUCTION

The large-scale structure of the Universe can be now measured using $\mathcal{O}$(million) galaxies using data from current surveys like the Sloan Digitial Sky Survey (SDSS). Upcoming surveys on the Large Synoptic Survey Telescope will probe even deeper and wider and aim to target on $\mathcal{O}$(10's of million) galaxies. With such a large galaxy data, we can measure the galaxy density field fairly accurately in the data. While the number of galaxies observed in these current and upcoming surveys is large, we have to compute the galaxy density fields *only once*. Thus, even a slow, correlation function code will be fine when measuring the data correlation function. However, predicting this galaxy density field data will require an even larger number of model galaxies – increasing the computational load of determining the density field even further. Typically, the modeling process also involves an MCMC – requiring $\sim$ millions of evaluations of the correlation function.[1]

2. BACKGROUND

2.1. HARDWARE ARCHITECTURE

Modern cpu's have a hierarchy of memory locations; the smallest (and fastest) are physically located close to the computing cores while the largest (and slowest) are the farthest. All cpu instructions need to be carried out from cpu registers - there are $\sim$ 100 registers typically available and the access times are essentially instantaneous. Next up is the *L1* cache divided into *L1D* for data and *L1I* for instructions cache. Since the cpu always necessarily executes instructions that are close together, we will ignore the instruction cache from now on. Typical L1 cache sizes range from 64KB to 128 KB (shared between instruction and data). Next level up is the L2 cache,

[1] I will assume that creating the models themselves is much faster compared to computing the correlation functions.

typically $\sim$ 256KB to 1 MB. The last level cache or the L3 cache is usually shared across all cores on the socket and can be 10 MB to 40 MB.

2.2. VECTORIZATION

2.3. OPTIMIZATION STRATEGIES

3. PACKAGE SUMMARY

`Corrfunc` is written primarily in `C` and comes with convenient `python` wrappers for typical clustering statistics. The fundamental design principles for `Corrfunc` are the following:

1. Correct – `Corrfunc` has a base set of correct outputs for every statistic generated either through slow, brute-force methods or independent, external codes. Within `Corrfunc`, every clustering statistic is covered with at least one test case that *requires* reproducing this 'known-correct' result *exactly*. The 'PASS/FAIL' status for each test is the direct output of `diff -q test_output known_correct_output`.
2. High Performance – Performance is the overarching goal for `Corrfunc`. The code-base has been repeatedly analyzed with Intel VTune (`https://software.intel.com/en-us/intel-vtune-amplifier-xe`) on a variety of platforms to optimize the innermost loops.
3. Portable – `Corrfunc` is written in `ISO/IEC 9899:1999` compliant `C`. All instruction set specific codes are protected via compile-time constant definitions.

Every clustering statistic in `Corrfunc` can either be accessed through the API call (e.g., via `python`), linking with the static libraries) or as a command-line executable.

4. Methods

4.1. Complexity of the Code

We need to compute pairwise () distances to get the correlation function. A naive implementation of a correlation function would compute *all possible* pairwise separations with a complexity $\mathcal{O}(N^2)$. However, for almost all correlation functions, we are only interested in separations less than a certain $\mathcal{R}_{max}$, where $\mathcal{R}_{max}$ is much smaller than the domain of the point distribution itself. We can then immediately see a way to prune pairs that can not *possibly* be within $\mathcal{R}_{max}$. If we impose a 3-d grid, with cell-size $\mathcal{R}_{max}$, then two points separated by more than one cell size ($\mathcal{R}_{max}$) in any one dimension can not be within $\mathcal{R}_{max}$ of each other. Thus, given one point which is the target galaxy and a grid with cell-size $\mathcal{R}_{max}$, immediately allows us to prune *all* of the points that are not within 1 cell offset along each dimension. However, even with this pruning, the actual implementation of the algorithm matters. For instance, a linked-list in each cell performs $\sim 60\times$ worse than the algorithm described here.[2] We can now compute the theoretical complexity of the proposed algorithm. Let $\mathcal{L}$ be the side-length of the cube over which $\mathcal{N}$ points are distributed, then each cell contains $r^3_{\mathrm{max}} \times \mathcal{N}/\mathcal{V}$ particles. To compute the correlation function, we have to loop over each point, and then *all* of the points in the neighboring cells. This results in a complexity of $\mathcal{O}(\mathcal{N}\mathcal{M})$, where $\mathcal{M} = \mathcal{N} \times (\mathcal{R}_{max}/\mathcal{L})^3$. Typical $\mathcal{R}_{max}$ is $\sim 0.10 - 0.2 \times \mathcal{L}$, therefore ordering the particles in cells of size $\mathcal{R}_{max}$ should result in a speedup of $(\mathcal{R}_{max}/\mathcal{L})^3 \sim 125 - 1000$ compared to the brute-force algorithm. Since every possible pair within $\mathcal{R}_{max}$ has to be examined, the algorithm deteriorates to $\mathcal{O}(\mathcal{N}^2)$ for $\mathcal{R}_{max}$ comparable to $\mathcal{L}$. Space partitioning tree based approaches are far more suitable for cases where $\mathcal{R}_{max}$ is $\gtrsim 0.5 \times \mathcal{L}$ (e.g., Curtin et al., 2013; Feng and Modi, 2016).

4.2. Partitioning the Particles based on $\mathcal{R}_{max}$

Fig. 1 shows a schematic 2-D grid for the partitioning scheme. The red circles represents the reference Poisson distributed points while the blue filled circle shows the query point. The grid is constructed in the exact same way for both datasets such that the 2-D (the index is in 3-D for the actual code) index for the query point is identical to that of the reference dataset. Once the central cell is determined, *any* reference point that satisfies the distance inequality $\mathcal{R}_{sep} < \mathcal{R}_{max}$, *must* lie within the shaded 9 cells (corresponds to 27 in 3-D).

4.3. How to Maintain Cache Locality within the Grid

For all pairs around a given target galaxy, we need to compute distances to all points within all neighbouring 3-d cells. We ensure that the particle locations are contiguous by moving them into the following `C struct` in the order in which they arrive.

[2]The usual linked-list is very cache-unfriendly. Each dereference requires a read from a new region of memory and an almost guaranteed cache miss.

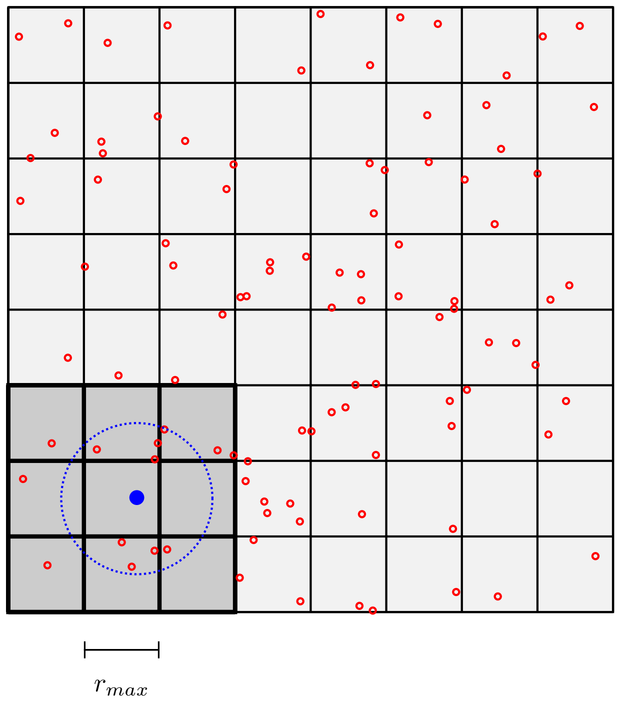

Figure 1: A 2-D grid showing the bin-lattice partitioning scheme. The bigger square show the entire domain, the red circles show a random distribution of 100 particles. Let's say we want to compute all pairs for the target blue point, then we would only have to consider red points that are within one cell (the dark shaded region). A circle with radius $\mathcal{R}_{max}$ is also drawn to shown the actual pairs that will eventually count in the correlation function.

```
typedef struct{
  DOUBLE *x;
  DOUBLE *y;
  DOUBLE *z;
  int64_t nelements;
} cellarray;
```

Since the typical particle data is small ($\sim$ 20-30 MB), duplicating the entire particle distribution does not impose a strong constraint on the memory requirements. However, this duplication allows us to store the particles contiguously and produces fewer cache misses while looping over particles in a cell. The entire 3-D particle distribution is then deposited on the uniform grid. Now, for each particle we need to visit 27 total cells to compute all possible pairs within $\mathcal{R}_{max}$.

5. The pair-counting algorithm

5.1. $DD(r)$

5.2. $\xi(r_p, \pi)$

5.3. $w_p(r_p)$

5.4.

5.5. Hand-written Vectorization Support

Advanced Vector Extensions (AVX) has been available in cpu's more recent than 2011. AVX allows the processing of 8 floats or 4 double simultaneously – thus, potentially increasing the throughput by a factor of 8x/4x. However, automatic vectorization is not always possible by the compiler and in those cases we can write AVX vector intrinsics to directly manipulate 8 floats/4 doubles[3].

6. Benchmarks & Scaling

In this section we present the runtimes and scalings for different number of particles, $\mathcal{R}_{max}$ and OpenMP threads for the codes. For all of the scaling tests, only an auto-correlation calculation was used and the fiducial catalog contains $\sim$ 1.2 million galaxies on a periodic cube of side 420 $h^{-1}Mpc$.

[3]Another option would be to use the vectorclass written by Agner Fog here: `http://agner.org/optimize/vectorclass/`

6.1. Scaling with Number of Particles

In Fig. 2, we show the scaling for the three codes with the number of particles. For this scaling, we subsampled the fiducial mock to attain 10 logarithmic steps in particle number ranging from 1.2×10^4 to 1.2×10^6.

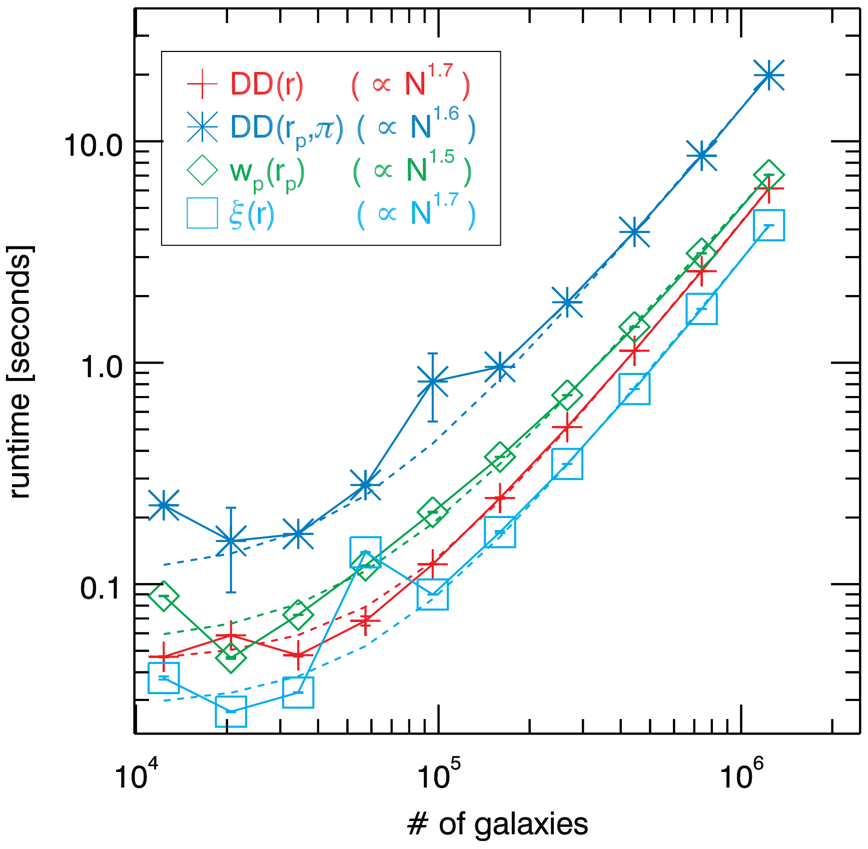

Figure 2: Scaling with particle number for $DD(r)$, $\xi(r_p, \pi)$, $w_p(r_p)$ and $\xi(r)$

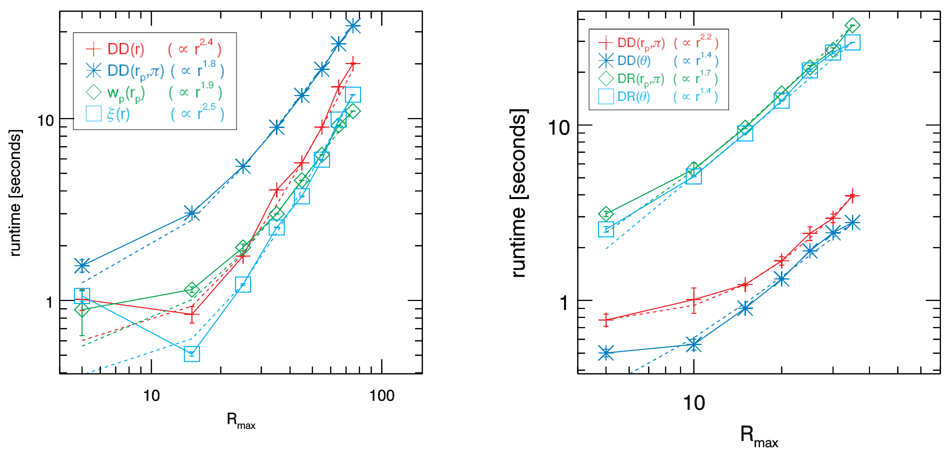

Figure 3: Scaling with $\mathcal{R}_{max}$ for $DD(r)$, $\xi(r_p, \pi)$, $w_p(r_p)$ and $\xi(r)$

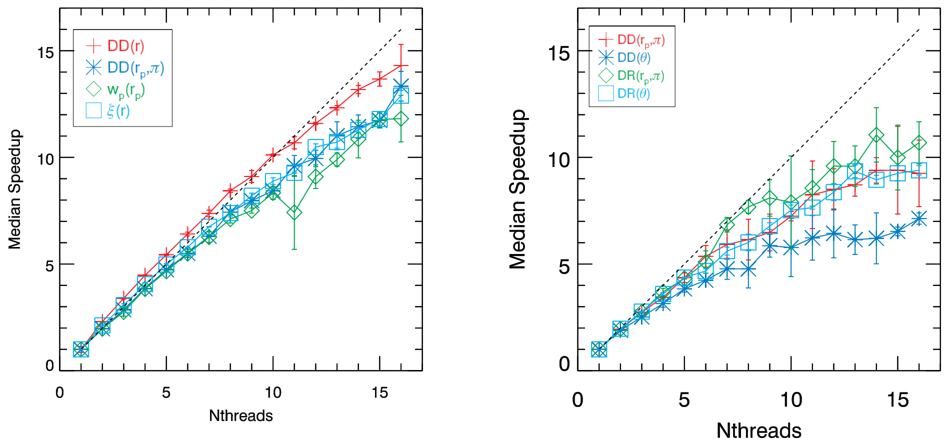

Figure 4: OpenMP scaling for $DD(r)$, $\xi(r_p, \pi)$, $w_p(r_p)$ and $\xi(r)$

Table 1: OpenMP scaling for the three codes. Efficiencies are defined as `Speedup/Nthreads`. Note, that $DD(r)$ has super-linear scaling with `nthreads` up to ~ 10 threads.

Nthreads	**Efficiency[%]**							
	Periodic Boxes				**Spherical Geometry**			
	$DD(r)$	$\xi(r_p,\pi)$	$w_p(r_p)$	$\xi(r)$	$DD(r_p,\pi)$	$DD(\theta)$	$DR(r_p,\pi)$	$DR(\theta)$
1	100	100	100	100	100	100	100	100
2	115	98	98	106	96	94	97	98
3	113	95	91	102	91	83	91	92
4	111	96	95	101	85	78	92	89
5	108	94	93	99	87	76	86	87
6	106	91	91	96	89	70	84	77
7	105	90	89	96	84	68	97	80
8	105	92	88	92	76	59	95	75
9	101	88	83	91	72	65	89	75
10	101	84	83	88	72	57	78	75
11	97	87	67	84	74	56	77	69
12	96	82	75	87	70	53	79	69
13	94	84	76	82	67	47	73	71
14	94	81	77	80	67	44	78	63
15	91	78	78	78	62	43	66	61
16	89	83	73	80	57	44	66	58

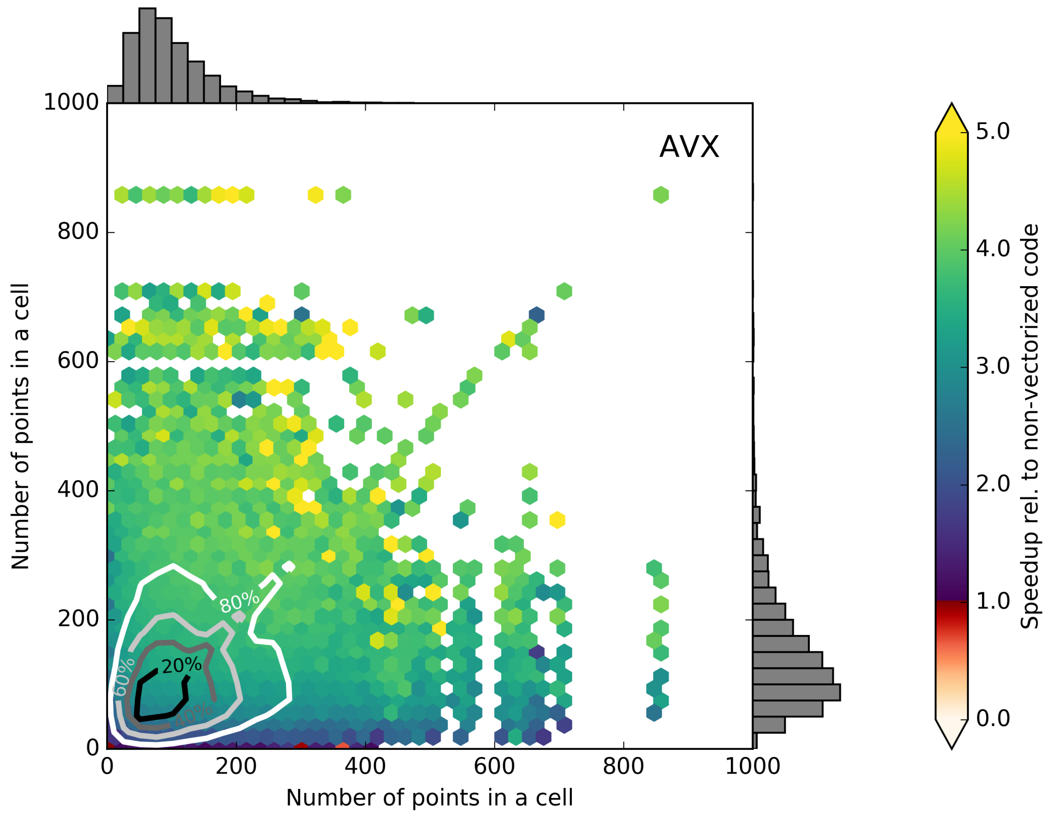

Figure 5: Speedup from AVX

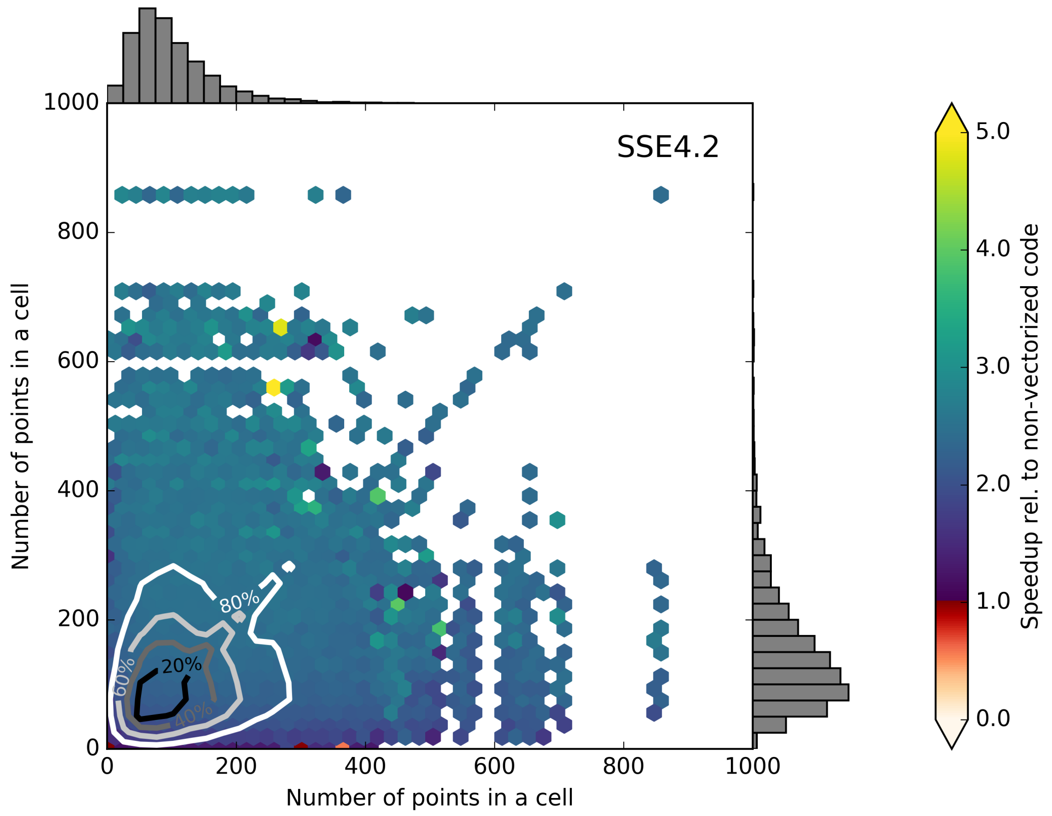

Figure 6: Speedup from AVX

6.2. Scaling with $\mathcal{R}_{max}$

6.3. Scaling with OpenMP threads

6.4. Speedup from SIMD code

7. Conclusions

I have presented a suite of three fast correlation function codes that take advantage of the underlying hardware. The reasons why the codes are much faster for typical workloads are:

- A 'bin-lattice' scheme is used to first partition the computational domain into 3-D cells that can then be used to prune majority of the volume.
- To obtain better cache-locality, the particle list is duplicated into a contiguous array for each dimension. Thus, all particles that fall into the same 3-D cell are stored in a contiguous array.
- With the AVX instruction set, modern CPU's can process 8 floats/4 doubles simultaneously. The codes contain hand-written AVX intrinsics that offer a factor of few speedup compared to the compiler generated scalar code.

Foreman-Mackey et al. (2013)

Acknowledgements

Curtin, R. R., Cline, J. R., Slagle, N. P., March, W. B., Ram, P., Mehta, N. A., Gray, A. G., 2013. mlpack: A scalable C++ machine learning library. Journal of Machine Learning Research 14, 801–805.

Feng, Y., Modi, C., Jul. 2016. A fast algorithm for identifying Friends-of-Friends halos. ArXiv e-prints.

Foreman-Mackey, D., Hogg, D. W., Lang, D., Goodman, J., Mar. 2013. emcee: The MCMC Hammer. PASP125, 306–312.