Optimiertes Samplen der Kühlfunktion für Kosmologische Simulationen mithilfe eines amorphen Mesh

Stefan Lüders

München 2021

Optimized Sampling of the Cooling Function for Cosmological Simulations Using an Amorphous Mesh

Stefan Lüders

Munich 2021

Optimized Sampling of the Cooling Function for Cosmological Simulations Using an Amorphous Mesh

Master Thesis

at the

Ludwig–Maximilians–Universität (LMU) München

submitted by

Stefan Lüders

(Matr. Nr.: 11587082)

born on 1996-10-26 in Aschaffenburg

supervised by
Klaus Dolag

and
Aura Obreja

Munich, January 31, 2022

Evaluator: Klaus Dolag

Contents

1 Introduction and goal of the thesis

At the time of writing the standard model of cosmology is the Lambda Cold Dark Matter model (ΛCDM). In this model only 5% of the total mass-energy of the universe is contained in the visible "baryonic" matter. The vast majority constitutes dark matter (27%) and dark energy (68%) (Planck Collaboration et al. 2020). The true nature of dark energy, described by the eponymous cosmological constant Λ, is still unknown. Dark matter however, while elusive, is somewhat less mysterious. It consists of massive, electromagnetically non-interactive material. The leading theory states that dark matter is made of as of yet undiscovered Weakly Interactive Massive Particles (WIMPs).

Even without full knowledge about the composition of dark matter, its effects on the universe are readily observed in the form of gravitational lensing, the shape of galactic rotation curves, galaxy clustering, and many other phenomena. Based on knowledge gained from these observations, it is possible to conduct dark matter simulations. These simulations help understand the dark matter distribution and the formation of structures in the universe on a cosmological scale. They show us that dark matter tends to form "halos", which serve as a scaffold for baryonic matter to form stars, galaxies, and clusters. Mo et al. (2010) contains a good overview of these structure formation processes.

Jeans & Darwin (1902) were the first to suggest that gravitational instabilities in the early homogeneous universe would lead to the structures we observe today. Small perturbations in the uniform distribution of early matter could amplify and collapse if they exceeded certain length and mass scales. Nowadays, these are called the Jeans length and the Jeans mass. The origins of these perturbations are in the quantum uncertainty of the hot gas particles of the early universe. These tiny inhomogeneities can still be observed, imprinted in the Cosmic Microwave Background (CMB).

Galaxy clustering studies provide very strong evidence that baryonic matter alone could not have formed the structures that are observed in the night sky. The collapse of purely baryonic matter is too slow and dark matter is required to explain the difference: Unhindered by pressure due to its electromagnetic inertness it can collapse into halos early on, and provide gravitational wells for baryonic matter to "fall into", accelerating its collapse.

Eventually the gas becomes dense enough to start collapsing under its own gravity. This effect soon eclipses the gravity of the dark matter halo, accelerating the process. At this point it becomes crucial whether the gas can effectively cool, radiating away its energy to allow for further compression. If it can not lose sufficient energy, a thermodynamic equilibrium will form and the collapse will halt, resulting in a so-called hot halo.

If the cooling continues successfully, however, clusters and galaxies can form as the gas becomes dense enough for star formation. Depending on the angular momentum, the gas composition, and potential interaction with other systems, the result can be an elliptical or a spiral galaxy.

Whether the cooling is sufficient or not depends chiefly on certain feedback processes. For example, supernovae can release incredible amounts of radiation energy into the gas, heating it and even blowing it away with the shockwave. Mass accreting supermassive blackholes at the centers of galaxies, called Active Galactic Nuclei (AGN), can also send out vast amounts of radiation over long periods of time. Both of these processes can prevent the gas from cooling.

The gas cooling is thus a vital part of modern cosmological simulations that attempt to model all of these effects. It is a complicated process, dependent on factors such as the gas temperature, metallicity, and density, as well as the radiation background and the resulting ionization. Many different cooling processes contribute to the overall cooling, such as metal line cooling, recombination cooling, bremsstrahlung cooling and more.

These dependencies and effects are described by the so called cooling function Λ, which gives the loss of energy per unit of time and volume that a unit of gas experiences. Despite the complexity involved, it is possible to accurately simulate Λ for a given set of circumstances using modern radiative transfer simulation software, such as Cloudy (Ferland et al. 2017). However, this process is too slow to work during a cosmological simulation; Hence, large tables of precalculated cooling values are created, and interpolations based on these tables are a crucial part of these simulations.

Conventionally, these tables are based on regular grids, and interpolated using linear interpolation. Regular grids suffer from exponentially increasing space requirements as the number of cooling function parameters grows, and with it the number of dimensions in the parameter space. Further, the repeated linear interpolations employed in traditional implementations do not always yield accurate results.

This thesis attempts to find a better, smarter way of sampling the cooling function parameter space by leaving behind the confines of regular grids, and adopting an irregular, amorphous mesh instead. The goal is to develop a sampling method that increases the interpolation accuracy and consistency, while reducing the number of required samples. Further, a new interpolation technique is required that can deal with the new, irregularly spaced data.

The thesis is structured as follows: Chapter 2 gives an overview over the physics of the cooling function, the simplifications that are usually applied to it, and the conventional sampling and interpolation methods. Chapter 3 presents the newly developed sampling and interpolation techniques, describing both the algorithms and their implementations. In chapter 4 this new software is evaluated under various metrics, both by direct comparison to conventional methods and within the context of small cosmological simulations. Chapter 5 summarizes the results.

2 Background information

This section first gives a short overview of what the cooling function is and what physical processes shape it. It then describes how it is usually stored and interpolated in cosmological simulations.

2.1 The Cooling Function

The cooling function Λ describes the amount of energy a gas cloud loses per unit of volume and time. The term "cooling function" is often used interchangeably for both the pure and net cooling function, where the net cooling function takes into account the heating rate Γ:

$$\Lambda_{\text{net}} = \Gamma - \Lambda \tag{1}$$

To avoid confusion this thesis will refer to cooling and heating separately and always refer to the net cooling as Λ_{net}.

The physical processes responsible for cooling depend strongly on the gas properties; the most important factors are metallicity and ionization (Tielens 2010). These cooling processes include recombination cooling Λ_{rec}, cooling through collisional excitation and ionization Λ_{coll}, metal line cooling Λ_{Z}, bremsstrahlung cooling/free free emission Λ_{ff}, and Compton cooling Λ_{comp}:

$$\Lambda = \Lambda_{\text{rec}} + \Lambda_{\text{coll}} + \Lambda_{\text{Z}} + \Lambda_{\text{ff}} \pm \Lambda_{\text{comp}} \tag{2}$$

Compton scattering may also have a heating effect if the gas temperature is below the CMB temperature T_{CMB}.

If the metallicity of the gas is high, cooling by metal ions following collisional excitation is the dominant cooling mechanism at lower temperatures (Draine 2011). This so called metal line cooling occurs when heavy metal ions are collisionally excited by interactions with free electrons. Because of the low density of the interstellar medium the new state is not stable long enough for collisional deexcitation to occur, and the electrons eventually fall back into their lower levels spontaneously, emitting a photon. If the gas cloud is sufficiently transparent, which is usually a fair assumption due to the low densities, the photon can escape the cloud and carry away the energy that was formerly carried by the ions. Thus, kinetic energy has been turned into radiative energy and removed from the system, creating a cooling effect (Osterbrock & Ferland 2006).

Metals are particularly effective at this process due to their numerous possible electron states with low excitation energies, but one of the most important features of the cooling function is at the point where hydrogen can participate in this process. This happens at $T = 1.6 \times 10^4\,\text{K}$, which is the temperature required to ionize hydrogen, visible in the cooling function as a steep slope (see fig. 1), and is particularly important for the cooling function interpolation (see section 4).

Metal cooling is not very efficient at very high densities and temperatures. At high densities, collisional deexcitation occurs before the electrons can fall back to their lower levels spontaneously (Draine 2011). At high temperatures, electrons are set free from their ions instead of simply being excited due to the high energies involved. This is a heating process, called photoionization heating.

Freed electrons still take part in the cooling process, however. They are the most frequent partner for collisional excitation, and they can be recaptured by ions leading to recombination cooling (Dopita & Sutherland 2003; Osterbrock & Ferland 2006; Draine 2011). Again, the kinetic energy of the electrons is removed from the gas and radiated away, lowering the overall internal

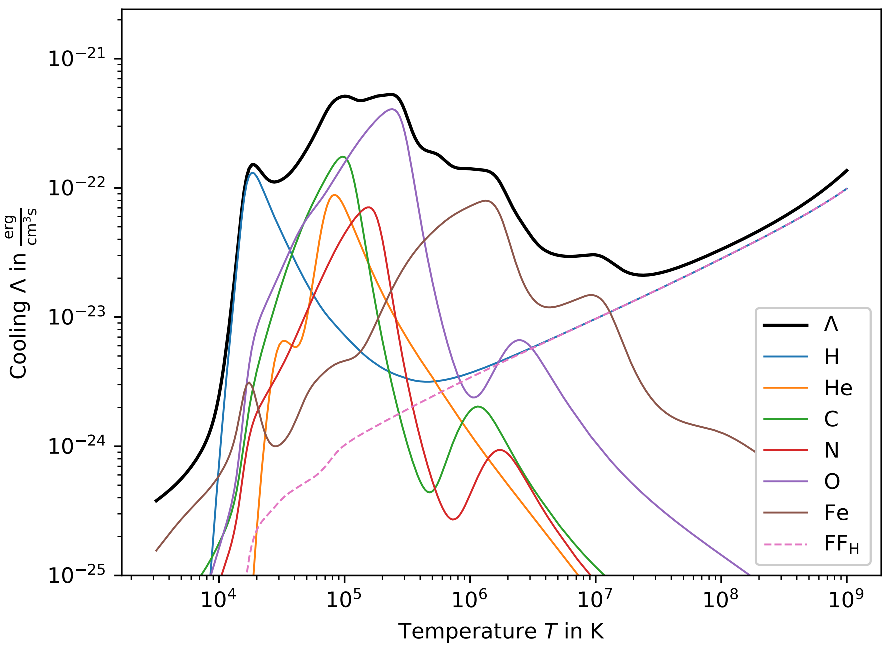


Figure 1: The cooling function at $z = 0$ for a gas with $Z = Z_\odot$ and $n_\mathrm{H} = 1\,\mathrm{cm}^{-3}$ (thick black curve), as well as the contributions of the most important metal species (colored curves). The ultraviolet radiation background is Cloudy's built-in HM12 (Haardt & Madau 2012). The hydrogen feature is clearly visible near 10^4 K. At high temperatures, free free emission due to hydrogen is the most important cooling process. Inverse Compton cooling is not effective at this combination of redshift and density.

energy and thus the temperature. Photoionization heating and recombination cooling usually stabilize into the so-called photoionization equilibrium.

At very high temperatures bremsstrahlung cooling, or free-free emission, dominates. It occurs when electrons closely pass by ions, where they are accelerated in their electric fields. Because of the acceleration they then radiate energy away in the form of bremsstrahlung (Draine 2011).

The Compton effect describes how photons scatter inelastically on free or loosely bound electrons. This transfers energy from the photons to the electrons, leading to a heating effect. The inverse Compton effect occurs when the scattering is superelastic instead. In this case the energy is transferred from the electrons to the photons, leading to Compton cooling (Dopita & Sutherland 2003).

2.2 Calculating the Cooling Function

Previously, cosmological simulations used Collisional Ionization Equilibrium (CIE) as basis for their cooling function calculations (e.g. Wiersma et al. 2009; Gnedin & Hollon 2012; Gnat & Ferland 2012). It describes an equilbrium between the collisional ionization of gas particles and the recombination of ionized particles with electrons (e.g. Dopita & Sutherland 2003). Under the further assumption that all ions are in the ground state, the CIE cooling function becomes independent of the hydrogen density n_H, making it easier to compute. However, this approach neglects photoionization effects. In the fully ionized limit, the CIE approximation for Λ can thus be off by up to two orders of magnitude (Gnedin & Hollon 2012).

Nowadays, blanket CIE approximation is not necessary anymore thanks to modern radiative transfer simulation software, and is only used for dense gas where it is appropriate. For less dense gas, a Photoionization Equilibrium (PIE) is assumed instead. It is even possible to compute the non-equilibrium cooling for large sets of reactions involving many species, including molecules (e.g. Richings et al. 2014a,b). This latter approach is very time consuming, and therefore not yet routinely employed in galaxy formation simulations.

All simulations in this thesis use the Cloudy simulation package to simulate the cooling function in PIE, specifically version 17.02, last described by Ferland et al. (2017). However, this approach is also too computationally expensive to use during cosmological simulations. A single Cloudy run takes on the order of ten seconds. Such a calculation would be required every time a particle is updated, and make large simulations infeasible.

Instead, cooling function values are tabulated beforehand and interpolated from these tables at runtime. However, in the general case the cooling function depends on a large set of parameters: Gas temperature T, baryon number density n_b, the fractional abundance X_{ij} for metal species i at level j, the distribution of the column density of species i at level j at different velocities compared to the systemic velocity $\frac{dN_{ij}(v)}{dv}$, the specific intensity of the radiation field J_ν, and the heating rate by cosmic rays ζ_{CR} (Gnedin & Hollon 2012):

$$\Lambda = \Lambda\left(T, n_b, X_{ij}, \frac{dN_{ij}(v)}{dv}, J_\nu, \zeta_{\mathrm{CR}}\right) \tag{3}$$

Thus, the number of parameters needs to be reduced to keep the size of the cooling tables viable. Equation 3 can be simplified by limiting it to the optically thin case (where $N_{ij} = 0$, excluding molecular, dust, and cosmic ray cooling), assuming ionization equilibrium and equilibrium of the level populations, and assuming that the relative distribution of heavy elements equals that of the solar system (Gnedin & Hollon 2012):

$$\Lambda = \Lambda\left(T, n_H, Z, J_\nu\right) \tag{4}$$

Here, hydrogen density n_H serves as a measure for baryonic density n_b, and metallicity Z is a multiplicative factor that gives the metallicity of the gas relative to n_H in terms of solar metallicity $Z_\odot$. The radiation background J_ν further depends on the redshift z via the CMB and the ultraviolet background (UVB), and may also be made up of several independent spectral energy distributions.

By simplifying the cooling function like this, the dimensionality of the parameter space spanned up by its arguments is significantly reduced. It can then be sampled to create large tables of Λ values, which can be used to interpolate approximate values during simulations. This is much faster than calculating the cooling function directly.

The lowest dimensional space typically used for Λ is $(n_{\mathrm{H}}, T, J_\nu(z))$, where $J_\nu(z)$ is the z-dependent UVB, and the contribution from metal line cooling is simply $\Lambda_{\mathrm{Z}} = Z/Z_\odot \Lambda_{\mathrm{Z}}(Z_\odot)$ (e.g. Shen et al. 2010). Relaxing this linear scaling for Λ_{Z} to account for individual metal species increases the dimensionality of the problem considerably (e.g. Wiersma et al. 2009). Additionally, relaxing the assumption that only the UVB acts as photoionization source increases the dimensionality even more (e.g. Gnedin & Hollon 2012; Kannan et al. 2016; Obreja et al. 2019; Ploeckinger & Schaye 2020).

2.3 Conventional Sampling and Interpolation Methods

Conventionally, the cooling function is sampled across its parameter space using a regular grid. There is a set number of sampling coordinates in each dimension, usually equidistant, and the set of points created by all possible combinations of these coordinates gives the sampling points of the grid. If such a grid has n sampling coordinates in each of its d dimensions, the final number of sampling points will be on the order of $O(n^d)$.

Points within the space covered by the grid can then be interpolated using multilinear interpolation (MLI). First, the hypercuboid containing the interpolation point, made up of the nearest two grid points in each dimension, must be found. There are 2^d of these points, and because of the regularity of the grid each can easily be found in constant time.

Then, repeated linear interpolation iteratively removes dimensions from this hypercuboid until it converges on the target point (e.g. Press et al. 2007). For each dimension, the hypercuboid is flattened such that its remaining volume intersects the interpolation point. This gives a new, less-dimensional cuboid, with known function values at all of its vertices. After enough iterations it reduces from a two dimensional plane to a line intersecting the target point, which can then be approximated.

Figure 2 visualizes this process for the three dimensional case. It starts out with a cube of grid points containing the target point. The function values are known at each vertex. Through $2^{(3-1)} = 4$ linear interpolations along the cube's edges, four new function values are approximated. They are located in a two dimensional plane that contains the target point. Again, $2^{(2-1)} = 2$ linear interpolations eliminate a dimension. Now, only two function values are left, defining a line through the target point. The target point can now be approximated with a single $(2^{(1-1)} = 1)$

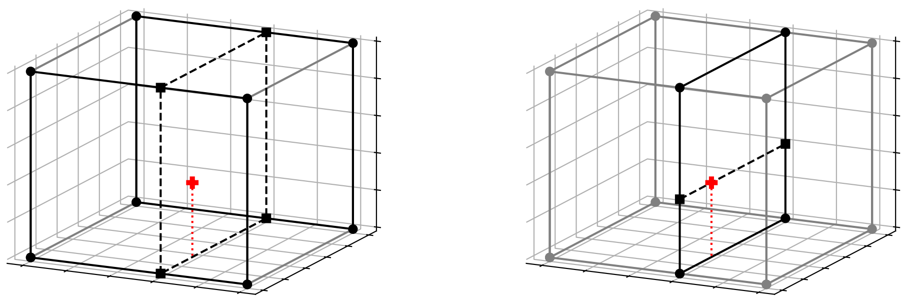

(a) 3D → 2D (b) 2D → 1D

Figure 2: In MLI dimensions are iteratively reduced within the grid using linear interpolation, until it is possible to interpolate the target point.

interpolation between these two points. Notably, despite the repeated linear interpolations the overall result is generally nonlinear (see fig. 3).

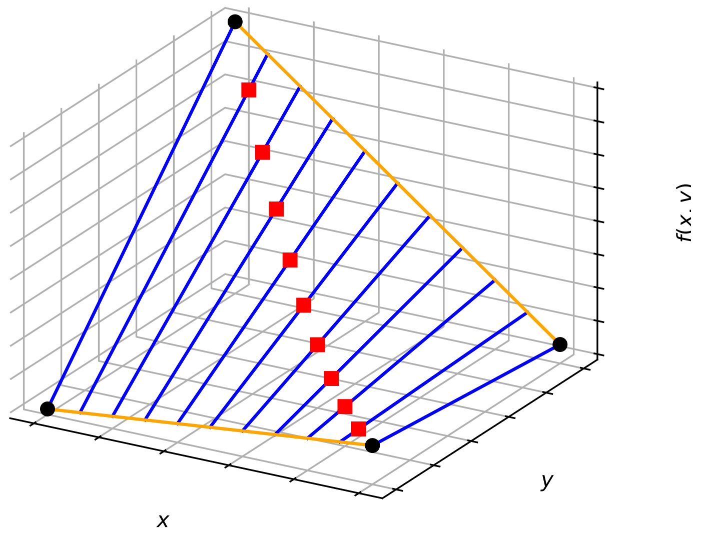


Figure 3: Repeated linear interpolation leads to nonlinear results: Four points that are part of a larger grid have function values corresponding to their height. Interpolating linearly between the corners (orange lines), then interpolating again between the interpolated values (blue lines) leads to a nonlinear distribution of results: The red squares are not on a line between the corner points. This has implications for the smoothness of the interpolation across larger grids. In particular across vertices of grid cuboids there may be discontinuities that could affect the interpolation quality.

It can easily be seen that each step requires $O(2^{\delta-1})$ linear interpolations, each of which can be executed in constant time. Here, δ is the number of remaining dimensions. The initial retrieval of the first hypercube's vertex points takes $O(2^d)$ time, so the overall time complexity of the process is:

$$O\left(\sum_{i=1}^{d} 2^i\right) = O(2^d) \tag{5}$$

Thus, both the space and time complexity of MLI grows exponentially with the number of dimensions.

More sophisticated cooling models may take into account additional features, such as the effects of individual metal species, or different photoionization sources. This quickly leads to high dimensional parameter spaces, and both the sampling and the interpolation on these spaces become inefficient. In order to keep the amount of samples at a reasonable size, the sampling resolution must be reduced in all but the most important dimensions. Accuracy suffers as a result.

Smart irregular sampling meshes could help reduce the overall number of required samples while keeping the accuracy reasonable. Dense sampling could be limited to parts of the parameter space that require it for high accuracy, while other parts would be sampled less densely. However, leaving the grid based approach behind requires new algorithms, both for the sampling of the mesh and for interpolating on that mesh.

3 Sampling and Interpolating Using an Amorphous Mesh

This section describes the new Delaunay triangulation based sampling and interpolation methods. It first gives some mathematical background information on Delaunay triangulations, ball trees, and barycentric coordinates in section 3.1. Section 3.2 describes the algorithms, as well as aspects of their implementation in the CHIPS python package and the DIP C/C++ library.

3.1 Mathematical Background

The algorithms outlined in section 3.2 use two key mathematical concepts in order to implement adaptive sampling and interpolation on the resulting irregular mesh. These two concepts are the so called Delaunay triangulation and barycentric coordinates. Further, ball trees are used to find the initial conditions for the "simplex flipping" algorithm. All three are explained in this section.

3.1.1 Delaunay Triangulation

A subdivision of a two dimensional space into triangles is called a triangulation. Algorithms to build such triangulations may either place the vertices of these triangles in the space themselves, or use an already existing set of points (e.g. Ch. 3 and 9 in Berg et al. 2008). The methods described in this work make use of the latter kind of algorithm. The main challenge for such an algorithm is to connect the points into a mesh in some optimal way.

Triangulations and many triangulation algorithms generalize to an arbitrary number of dimensions (e.g. Barber et al. 2013). In this case the triangulation does not consist of triangles but of simplices, which are generalizations of triangles. An n-dimensional simplex is called an n-simplex. A 3-simplex is also called a tetrahedron.

One triangulation scheme with desirable properties is the Delaunay triangulation, for which a variety of algorithms exist. It was named after the Russian mathematician Boris Nikolaevich Delone, Delaunay being the French transliteration (Berg et al. 2008). It is closely related to the Voronoi tesselation. The special property of the Delaunay triangulation can be expressed in several equivalent ways (Press et al. 2007):

- Most intuitively the Delaunay triangulation **maximizes the internal angles of all simplices**.
- None of the circumcircles of the simplices contain any points - they only have one simplex's points on their surface. In arbitrary dimensions these are hyperspheres.
- In the $\mathbb{R}^2$ case, two points in the triangulation are connected by an edge if and only if there is some circle connecting them which contains no further points. This property is used in the "Randomized Incremental Algorithm" (see below).

As a direct result the Delaunay triangulation minimizes the amount of long, thin simplices, which is advantageous for interpolation purposes. However, while the Delaunay triangulation maximizes the overall internal angles of all simplices, it does not minimize the maximum angle or the lengths of edges (Press et al. 2007). This means that it can not guarantee that long, thin simplices do not occur. In fact, as the number of dimensions rises they become unavoidable unless the points are distributed very evenly.

One possible way to prevent this would be to use a mesh refinement method such as Delaunay refinement (Shewchuk 2002), which adds more points to the space in order to break up ill-behaved simplices. This is done by placing a point inside the circumsphere of each problematic simplex turning those simplices invalid. However, such techniques are not applied in this work since they undermine the advantages of the sampling algorithm, namely the low amount of required samples and the possibility to have intentionally undersampled areas of the parameter space (see section 3.2.1). These refinement techniques are also associated with a large computational cost, as each simplex needs to be checked for some kind of refinement criterion and the number of simplices in a typical triangulation is large.

Specifically, given a set of n points in d dimensions, the Delaunay triangulation on this set contains $O(n^{\lceil d/2 \rceil})$ simplices (Seidel 1995). Such a triangulation is unique, provided no $d+1$ or more points are on a straight line, and no $d+2$ or more points are on a d-dimensional hypersphere (Press et al. 2007).

A variety of algorithms to generate Delaunay triangulations exist (e.g. Welzl et al. 1997). One very intuitive way to calculate a Delaunay triangulation in two dimensions is the “Randomized Incremental Algorithm” described in Guibas et al. (1992).

The algorithm works by individually adding points to the triangulation from a given set of points. The order in which the points are added is arbitrary and does not change the result. This process gives the algorithm its name.

After each point the triangulation is updated to ensure it remains a valid Delaunay triangulation: The new point is connected to each of the points making up the triangle containing it, creating 3 new triangles. Then, the new triangles are checked for validity. A triangle is valid if its circumcircle does not contain any points. If a triangle is invalid it has at least one illegal edge. An edge is illegal if all circles that connect its two points contain at least one point.

If a triangle has an illegal edge it is removed in a so-called “edge-flip”, removing the edge and bisecting the newly created quadrilateral with the other possible edge. This process repeats recursively for all newly created triangles until all triangles are valid and a new point can be added. Figure 4 illustrates this process. In implementations a tree is used to keep track of the recursion.

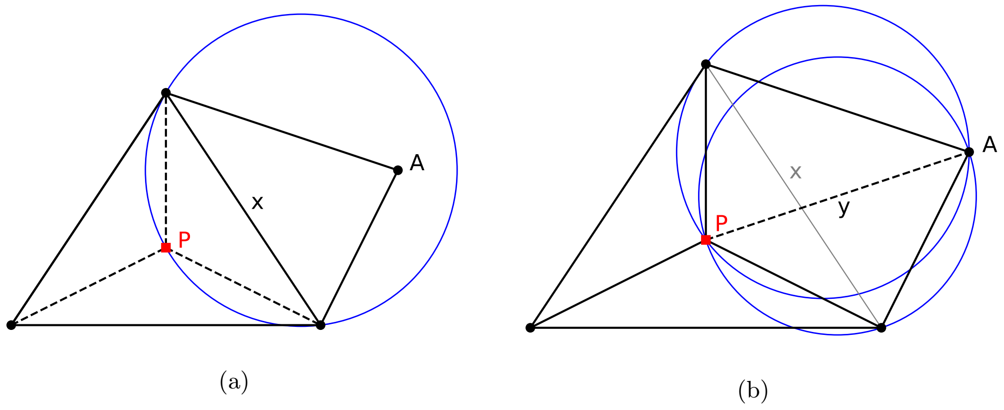


Figure 4: Reproduced from fig. 21.6.5 in Press et al. (2007). This figure visualizes the Randomized Incremental Algorithm. In (a), a valid Delaunay triangulation of four points is expanded with a new point P. The point is connected with all the vertices of the triangle that contains it, creating three new triangles. However, one of the newly created triangles is invalid and contains the illegal edge x. In (b), edge x has been removed, and the quadrilateral created by this removal has been bisected by a new edge y. Both of the triangles created by this step are valid and a new Delaunay triangulation has been found.

The advantages of the Randomized Incremental Algorithm are its ease of understanding and the tree structure used in its construction, which can also be used to find the simplex that contains any arbitrary given point. The main disadvantage of this method is that it does not generalize to higher dimensions.

This is why in this work the Delaunay implementation of the SCIPY python package[1] is used instead. This method is less intuitive but does generalize. It uses the QHull[2] library which implements the Quickhull algorithm (Barber et al. 2013). QHull calculates the Delaunay triangulation in an $\mathbb{R}^d$ space by “lifting” the triangulation points into $\mathbb{R}^{d+1}$ onto a paraboloid and computes the

[1]https://docs.scipy.org/doc/scipy/reference/generated/scipy.spatial.Delaunay.html
[2]http://www.qhull.org

convex hull of this paraboloid using Quickhull. The projection of the ridges of the lower convex hull back into d-dimensional space then gives the Delaunay triangulation of the original set of points (Brown 1979; Barber et al. 2013).

To identify which simplex contains a given point, a ball tree (see section 3.1.2) and a custom "simplex flipping" algorithm (see section 3.2.2) are used instead.

3.1.2 Balltrees

Balltrees are space-partitioning binary trees similar in construction to kd-trees. However, in addition to the elements of the tree structure each node in the tree has an associated $\mathbb{R}^d$ hypersphere or "ball". These balls are chosen with a minimum radius such that the ball of any given node contains all balls of that node's subtree. The radius of balls associated with leaf nodes is zero (Omohundro 1989).

Balltrees are very efficient structures for finding nearest neighbors of a given point. The balls allow ruling out large parts of the tree as potential solutions: If there is a closest point P to the target point, and the ball of a given node N is farther from the target point than P is, then none of the nodes in the subtree of N can be closer to the target point than P.

DIP (see 3.2.2) uses the k-d construction and nearest neighbor finding algorithms described in Omohundro (1989) to build a ball tree over the centroids of all simplices in a Delaunay triangulation and identify the simplex most likely to contain the target point. It is then used as starting point to find the simplex actually containing the target point.

3.1.3 Barycentric coordinates

Barycentric coordinates (Möbius 1827) are homogeneous coordinates relative to the vertices of a simplex. There is one coordinate associated with each vertex. Since a d-dimensional simplex has $d+1$ vertices, this means there are $d+1$ barycentric coordinates, one of which is linearly dependent.

The name derives from the term barycenter, referring to the center of mass of several bodies. Barycentric coordinates refer to that point in space which would be the barycenter of the associated simplex, if each vertex of that simplex had a mass equal to the associated barycentric coordinate (Coxeter 1969).

For example, in the 2D case the barycentric coordinates $(0, 0, 1)$ would refer to the third vertex of the triangle. The coordinates $(1, 1, 1)$ or $(\frac{1}{3}, \frac{1}{3}, \frac{1}{3})$ would refer to the centroid of the triangle. Barycentric coordinates may also become negative. In this case they refer to areas outside of the simplex (see fig. 5). The coordinates $(0, 0, 0)$ are invalid and do not represent any point.

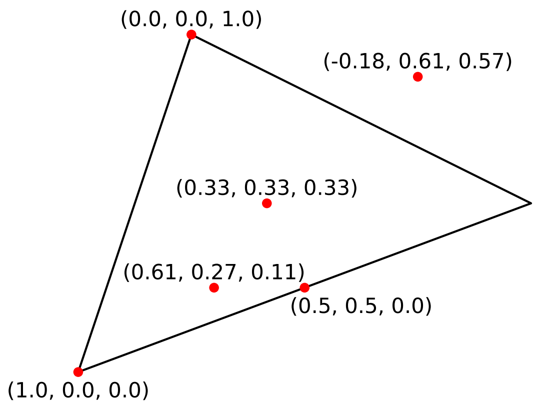


Figure 5: Examples for barycentric coordinates in the 2D case. Note how the coordinates become zero as they hit the edge opposite of the associated vertex, and negative after.

Since it is the ratio between the coordinates that matters and not their magnitude, barycentric coordinates are not unique. A set of coordinates multiplied by a scalar still represents the same point in space. By convention the coordinates are normalized such that their sum equals unity:

$$1 = \sum_{i=1}^{d+1} \lambda_i \tag{6}$$

In this form they are also called areal coordinates, because in the 2D case each barycentric coordinate denotes the fraction of the triangle opposite the coordinate vertex relative to the total triangle area (Coxeter 1969). With this normalization coordinates can be easily calculated from the following linear transformation:

$$\mathbf{T} \cdot \boldsymbol{\lambda} = \mathbf{r} - \mathbf{r}_{d+1}, \tag{7}$$

where $\boldsymbol{\lambda}$ are the barycentric coordinates, $\mathbf{r}_{d+1}$ is the vector of the $d+1$ vertex, $\mathbf{r}$ is the vector of the coordinates being transformed, and $\mathbf{T}$ is the following transformation matrix:

$$\mathbf{T} = \begin{pmatrix} r_{11} - r_{d+1,1} & r_{21} - r_{d+1,1} & \dots & r_{d1} - r_{d+1,1} \\ r_{12} - r_{d+1,2} & r_{22} - r_{d+1,2} & \dots & r_{d2} - r_{d+1,2} \\ \vdots & & \ddots & \vdots \\ r_{1d} - r_{d+1,d} & r_{2d} - r_{d+1,d} & \dots & r_{dd} - r_{d+1,d} \end{pmatrix} \tag{8}$$

Note that if the simplex is not degenerate, $\mathbf{T}$ is invertible and can be easily stored to convert from cartesian to barycentric coordinates, by solving the equation with e.g. Gaussian elimination. This is generally a $O(d^3)$ operation (e.g. Farebrother 1988).

Most importantly, normalized barycentric coordinates can be used for interpolation between the simplex vertices (e.g. Hormann 2014). One can assign a function value to each vertex, then calculate the weighted average of these values, using the coordinates as weights. The result is the interpolated function value at the point described by the coordinates. If the coordinates are already normalized the normalization drops out of the weighted average:

$$f(\boldsymbol{r}) \approx \sum_{i=1}^{d+1} \lambda_i f(\boldsymbol{r}_i) \tag{9}$$

Here, the function values $f(\mathbf{r}_i)$ at the position $\mathbf{r}_i$ of each vertex i are used to interpolate at point $\mathbf{r}$, which is represented by barycentric coordinates $\boldsymbol{\lambda}$.

Interpolation using this method is akin to moving on the d-dimensional hyperplane defined by the function values on the vertices. This makes it a “true” multdimensional linear interpolation (compare fig. 6, contrast fig. 3). It linearly extends past the simplex as one would expect, although extrapolated values quickly become unusable for practical purposes.

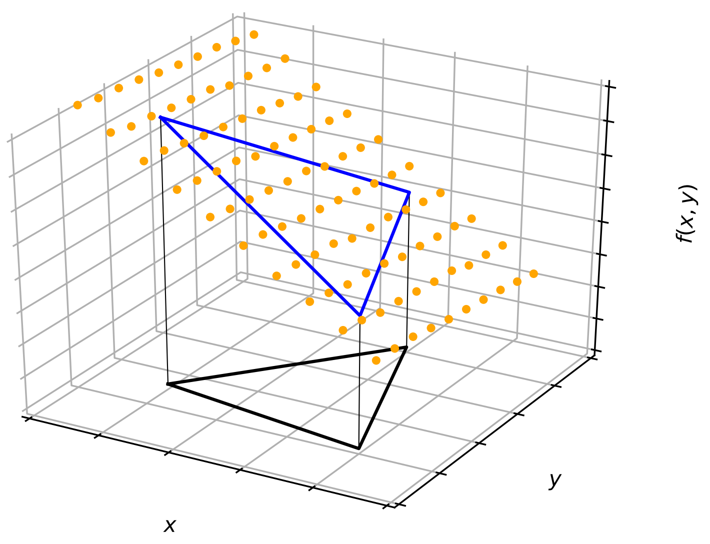


Figure 6: The black 2D simplex has associated function values, lifting the simplex into 3D space (blue triangle). The orange points show function values interpolated using they barycentric coordinates of the simplex. They lie on a plane defined by the simplex and its associated function values.

3.2 Implementation

This section describes the two software packages developed for this thesis and their algorithms. The **C**loudy based **H**euristic and **I**terative **P**arameter space **S**ampler (CHIPS) is a python package that implements the iterative sampling algorithm that produces the amorphous mesh of cooling data. This data is then exported for use in the **D**elaunay **I**nterpolation **P**rogram (DIP), a C/C++ library implementing efficient barycentric coordinate interpolation (see fig. 7).

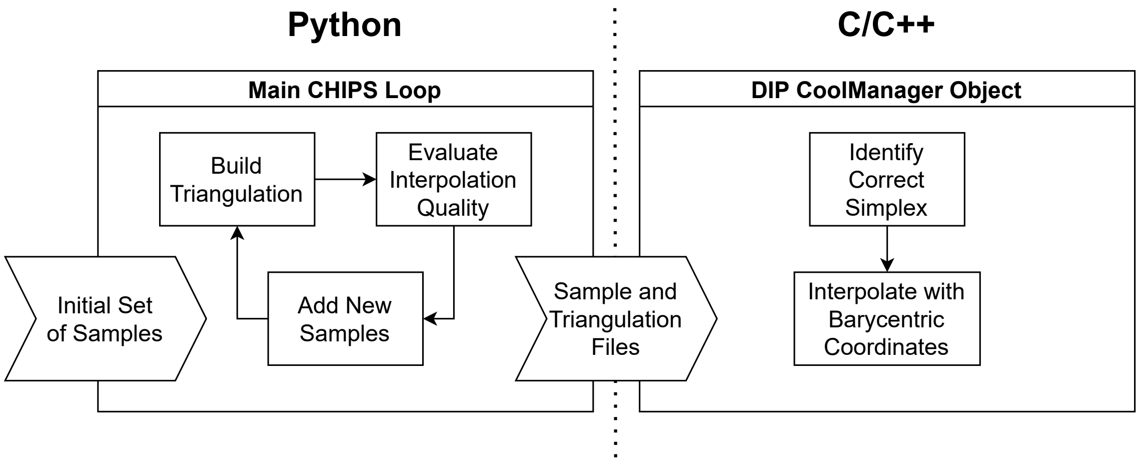


Figure 7: Chart showing the workflow of CHIPS and DIP. The CHIPS algorithm starts with an initial set of samples and iterates until it has built a satisfactory cooling mesh. In each iteration it builds a triangulation on the current set, evaluates the interpolation quality from that triangulation, and adds more samples if necessary. If exit conditions are met, samples and triangulation are exported to files that can be used by DIP. DIP's main challenge is finding the right simplex for a given interpolation point, which can then be interpolated using the barycentric coordinates relative to that simplex.

3.2.1 CHIPS

The goal of CHIPS is to sample the parameter space in such a way that interpolation results based on the sampling have consistent accuracy. In the parts of the parameter space where the gradient of the cooling function is large, denser sampling is required than in areas where it is smaller. This makes a regular-grid-based approach infeasible; an irregular mesh is required instead.

To build such a mesh one needs to evaluate the interpolation quality provided by the current mesh, identify areas of insufficient quality, and add new samples in those areas. This is challenging because drawing new samples is computationally expensive, making it infeasible to have an "evaluation" data set to compare against. If one could prepare a sufficiently dense data set to accurately judge the interpolation quality across the entire parameter space, "smart" sampling would not be necessary - one could simply use multilinear interpolation on the test data set.

The solution is to evaluate the quality of the interpolation using samples that are already part of the mesh. By removing a sample from the mesh and interpolating at its position it is possible to judge the interpolation quality at that point. If the error is larger than some given threshold, a new sample is drawn in the vicinity.

In practice, removing each sample individually and checking the interpolation on it is not feasible, since it requires building a new Delaunay triangulation for each point. Instead, the samples are divided into equally sized partitions, then all samples inside one partition are checked simultaneously. This leads to slightly more samples being drawn than needed, since samples within a partition aren't available for interpolation of other samples within the same partition, which leads to slightly lower interpolation quality.

The number of partitions is chosen by the user. By default, CHIPS uses 10 partitions. Using more partitions does not seem to lead to additional improvements.

Once all samples over the accuracy threshold have been identified, new samples need to be drawn. Several different methods of drawing the samples were tried. At first, kernel density estimation (KDE) was used to randomly draw samples in areas with low accuracy. However, this proved impractical: The random nature of KDE led to a lack of control that had to be compensated with oversampling, and there was no clear optimal choice of hyperparameters.

Further, biased sampling was tried. The goal was to draw the sample in such a way that it took outside information, such as the value or differential of the cooling function at the input points, into account in order to maximize the information gain. However, this proved to be too biased and led to distinct and unwanted inhomogenities in the sampling distribution.

Random uniform sampling within the simplices that contained the interpolation points proved successful, but required too many points for a satisfactory mesh.

In the final version of the algorithm, new samples are chosen as the centroids of the simplices containing at least one point not fulfilling the accuracy requirements. This leads to an unbiased, optimal refinement of the triangulation mesh, while at the same time minimizing the number of new samples per iteration: If multiple samples within a simplex do not fulfill the requirements, only one new sample is drawn.

Algorithm 1 The CHIPS algorithm in detail.

```
Require: Set of existing samples $S$, empty set $C$, error threshold $t$
 1: repeat
 2:     Partition $S$ into $N \in \mathbb{N}$ partitions $S_i$
 3:     for $i = 1$ to $N$ do
 4:         Build Delaunay Triangulation on $\tilde{S} = S \setminus S_i$
 5:         Interpolate points in $S_i$ using triangulation on $\tilde{S}$
 6:         Calculate differences between interpolation and correct values
 7:         For all samples with a difference greater than $t$, add the centroid of the containing
            simplex to $C$
 8:     end for
 9:     Evaluate $\Lambda$ for all points in $C$, add results to $S$
10:     $C \leftarrow \emptyset$
11: until exit conditions are fulfilled
```

Algorithm 1 describes the full, final sampling algorithm. This is the version of the algorithm implemented in the CHIPS python package. This package and its documentation are available here[3].

CHIPS uses the radiative transfer code Cloudy (Ferland et al. 2017) to calculate the cooling function. Thus, the most important input is a Cloudy input template file, which is filled with parameters using python format strings. This means that any part of a Cloudy input file that can be varied can be a dimension of the parameter space that is being sampled. Cloudy and its documentation can be found here[4].

Care must be taken when using commands that interpret inputs differently depending on their magnitude! For example, the `metals` command interprets values smaller than or equal to zero logarithmically and values greater than zero linearly. This leads to a discontinuity in the parameter space, which will result in oversampling and prevent CHIPS from converging on a solution. This can be prevented by enforcing linear or logarithmic interpretation with the `lin` and `log` keywords. In general, it is highly recommended to use logarithmic scaling for all parameters in order to avoid numerical issues.

In addition to these "regular" parameters, the CHIPS package also supports adding custom radiation components. In this case the parameter is a scaling factor applied to a particular radiation field, also known as spectral energy distribution (SED). These custom SEDs are automatically overlaid and appended to the input file, where they will be used on top of any other radiation fields loaded by Cloudy, such as the CMB or UVB. This feature was added to provide compatibility to the already existing custom SEDs implemented in Gasoline2 (Obreja et al. 2019).

CHIPS provides an initial sampling for the given parameter space using Poisson disc sampling. It uses a variation of the Poisson disc sampling algorithm described in Bridson (2007). It generates samples such that there is both a minimum and maximum distance to each of a sample's neighbors. This leads to an even sample distribution throughout the domain, unlike regular uniform sampling which shows over- and underdensities unless the number of samples is huge. The advantage of

[3]`https://github.com/Vetinar1/CHIPS`
[4]`https://nublado.org`

distributing the initial samples this way instead of simply using a sparse grid is that, unlike a grid, it introduces no structure and hence no bias to the initial mesh.

The number of samples generated in this way is limited by $\frac{1}{r^d}$, r being the minimum distance between the points and d being the number of dimensions. A good choice of r is thus dependent on d.

There is, however, an option to add uniformly distributed samples in each iteration. Their purpose is to provide an independent influx of points to areas of the parameter space where the algorithm may have converged to a local optimum. How this parameter should be chosen depends on d and on the particular parameters making up the parameter space.

During runtime, the algorithm actually samples in a wider space than the user defined at the start. The difference between the parameter space selected by the user and the parameter space sampled by the program are the "margins".

These margins are necessary in order to provide accurate interpolation at the edges of the parameter space. Information from beyond these edges must be taken into account to properly sample and interpolate around them. While sampling does also occur within the margins, these samples are not actually interpolated and evaluated by the algorithm. This leads to a gradual decline in sampling density from the inner edges of the margins to their outer edges.

The opposite occurs if the margin size is reduce to zero: The algorithm will fail to correctly interpolate near the edges, and attempt to increase the sampling density. This leads to prohibitive oversampling, and in the worst case scenario the algorithm does not converge.

Testing with DIP has revealed that points from the margins work poorly for interpolation purposes. For this reason they are dropped after CHIPS is complete. Because interpolation near the edges of the parameter space can still be a problem for DIP, it is thus recommended to leave an additional buffer zone and make the parameter space slightly larger than required (see fig. 8).

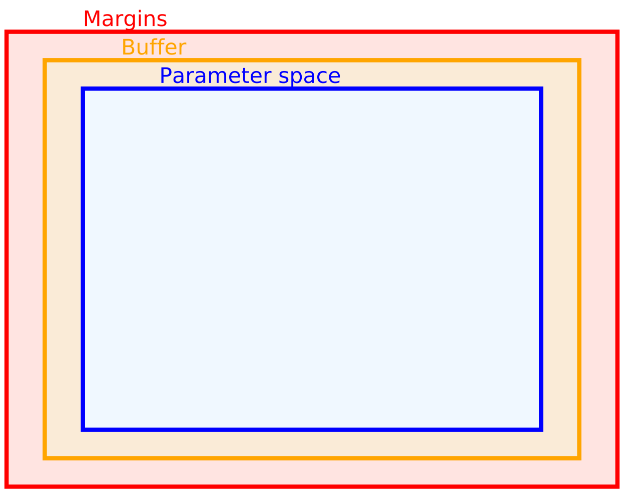


Figure 8: Visualisation of the relationship between the parameter space and margins. The user wishes to use the space marked in blue with DIP. Because DIP encounters problems near the edges of parameter space, they instead sample further and leave a buffer zone, marked in orange. CHIPS itself adds the margins, marked in red, in order to optimize the sampling process. Samples from the margins are removed once the algorithm is complete.

The following parameters determine whether a run exits after an iteration or not:

- Error threshold and error fraction: The algorithm stops if the fraction of samples with an error larger than the threshold is no larger than the error fraction.
- Maximum error: The algorithm does not stop as long as any sample's error is larger than the maximum error. Care must be taken when using this, as parts of the parameter space sometimes do not converge properly, e.g. in corners. It should therefore always be combined with some other hard exit condition.

- Maximum number of iterations: The algorithm always stops if the maximum iteration is reached.
- Maximum number of samples: The algorithm stops after the maximum has been reached. It will take into account the number of new samples in the previous iteration to not overshoot the limit.
- Maximum Runtime: The algorithm will stop if the maximum time is reached, measured in wall clock time. It will attempt to take into account the duration of the previous iteration to not overshoot the runtime limit.
- Maximum storage space: The algorithm will stop if the resulting data files exceed a maximum size limit. It will attempt to take into account the storage requirements of the previous iteration to not overshoot this limit. By default, all CLOUDY output files are saved. This way the data can be reconstructed in case the program quits unexpectedly. Storage saving options that automatically delete certain or all CLOUDY files are available.

After the final iteration, CHIPS saves the n points, the $O(n^{\lceil d/2 \rceil})$ simplices with $d+1$ points each and the neighborhood relations between the simplices, where each simplex has $d+1$ neighbors. Therefore, the output of CHIPS has an overall space complexity of

$$O(n + (d+1) \cdot n^{\lceil d/2 \rceil} + (d+1) \cdot n^{\lceil d/2 \rceil}) = O(dn^{\lceil d/2 \rceil}). \tag{10}$$

3.2.2 DIP

DIP is a C/C++ module that provides fast, efficient interpolation on data generated by CHIPS. While the module itself is written in C++ it also provides a C interface. DIP and its documentation can be found here[5].

Given a point in the parameter space, DIP returns the interpolated value of the cooling function at that point. The main challenge faced by DIP is finding the correct simplex containing this point, in order to interpolate it using barycentric coordinates. This is done in a two step process: First a ball tree is used to identify a simplex close to the interpolation point, then the correct nearby simplex is found using an algorithm dubbed “simplex flipping”.

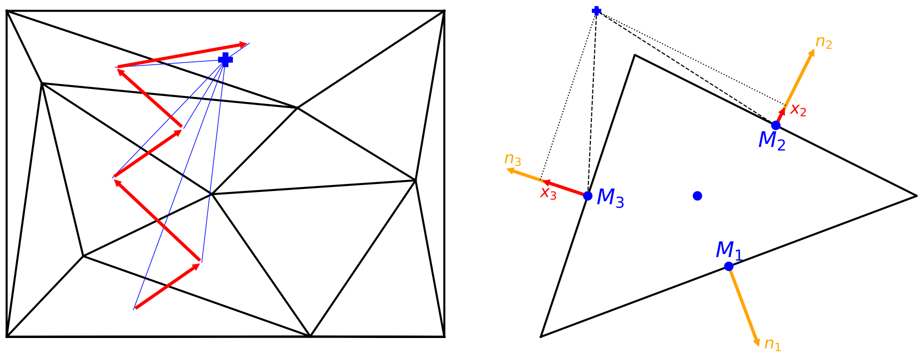
The ball tree is built over the centroids of all simplices in the triangulation when the triangulation is loaded, and is used to find the centroid closest to the interpolation point. The associated simplex provides the starting point for the simplex flipping algorithm.

Algorithm 2 The DIP algorithm.

```
Require: Interpolation point $P$, number of flips $n_f = 0$, maximum number of flips $F_{\max}$
 1: Use Ball Tree to find the centroid closest to $P$
 2: The associated simplex is the first candidate simplex $S_C$
 3: while $P \notin S_C$ do
 4:     for all faces $F_i$ of $S_C$ with midpoint $M_i$ and outwards pointing normal $\vec{n_i}$ do
 5:         $\vec{d_i} = M_i - P$
 6:         $x_i = \vec{d_i} \cdot \vec{n_i}$
 7:     end for
 8:     Determine $j$ such that $\forall i : x_j >= x_i$
 9:     Let $S_N$ be the neighboring simplex sharing $F_j$ with $S_C$
10:     $S_C \leftarrow S_N$, $n_f \leftarrow n_f + 1$
11:     if $n_f = F_{\max}$ then
12:         break
13:     end if
14: end while
15: Use barycentric coordinates of $S_C$ to interpolate $P$
16: return $P$
```

The simplex flipping algorithm works by iteratively moving through the triangulation from simplex to simplex. Since a simplex shares all of its points except one with a given neighbor simplex, moving to that neighbor is akin to “flipping” the simplex across the shared face.

[5] `https://github.com/Vetinar1/DIP`



(a) (b)

Figure 9: (a) The path the simplex flipping algorithm takes from a starting simplex in the bottom left until it finds the interpolation point marked by the blue plus. (b) Diagram illustrating how the next simplex is chosen. The target point P is at the position of the blue plus. The difference vectors between P and the face midpoints M_i are projected on the normal vectors $\vec{n}_i$, giving the projection vectors $\vec{x}_i$, the largest of which determines the direction of the next simplex. For clarity the projection on $\vec{n}_1$ is not shown: Since $\vec{x}_1$ is negative, it must lead away from the target point.

Algorithm 2 describes in detail how to determine the next simplex to move to. It starts with a candidate simplex S_C, initially determined using the ball tree, and the target point P, which gives the coordinates at which one wants to interpolate.

First, it checks if the interpolation point is contained in the candidate simplex. This can be done by calculating the barycentric coordinates of P relative to S_C. If all coordinates are greater than zero the simplex contains the point. Points on the surface of one of the simplex faces are also considered to be inside.

If the simplex does not contain the point, a new candidate simplex must be found out of the neighboring simplices. Here, the difference vectors $\vec{d_i}$ between the target point and the midpoints M_i of the faces of the simplex are calculated, as well as the scalar product between the outwards facing normals $\vec{n}_i$ and the difference vectors $\vec{d_i}$. This results in a value x_i for each face.

If $\vec{d_i}$ and $\vec{n}_i$ are parallel $x_i = 1$, and going to the neighboring simplex in the $\vec{n}_i$ direction would clearly move towards the target point. If they are antiparallel $x_i = -1$, and the associated neighbor simplex is clearly further away from the target point (see figure 9).

Thus, the heuristically best simplex to flip to to reach the target point is the one which maximizes x_i. This simplex becomes the new candidate simplex S_C and the process can repeat. Once the simplex containing the target point has been found, the interpolation can be done easily using the already calculated barycentric coordinates.

In order to safeguard against unforeseen infinite loops, there is also a parameter to limit the number of total flips. If this limit is reached, the last found simplex is used for extrapolation. While the results of this are not as good as if the correct simplex had been identified, they should still give a correct order of magnitude estimation for the cooling function.

Overall, the flipping algorithm scales as $O(F_{\max}d^3)$, although in practice $F_{\max}$ is only rarely reached. However, getting a "typical" number of flips is difficult, since it depends strongly on the dimensionality, the part of the parameter space, and the specifics of the triangulation. The d^3 term stems from the Gaussian elimination used to calculate the barycentric coordinates at each step, which takes $O(d^3)$ time in d dimensions. Section 4.2.4 gives more information on the performance of this algorithm.

The DIP package contains an object class `Cool` that implements this algorithm. The user simply needs to instantiate the object, pass the files containing the data, call the tree building function, and set the clamping of the parameter space to avoid issues at the edges (see also section 3.2.1). Since `Cool` objects hold a lot of data they should always be allocated on the heap to avoid a stackoverflow. The C interface does this automatically.

In practice, it is prudent to treat redshift separately from the other dimensions, partly because it is difficult to get good sampling at $z = 0$ if the redshift is treated as a continuous dimension. The solution is to sample z in "slices", much like in the traditional regular grid - although these slices do not have to be equidistant from each other. DIP provides a object class `CoolManager` that automatically handles two `Cool` objects and interpolate linearly between them as needed. After passing a file containing a map of valid redshifts and associated files, as well as a starting redshift, the object is ready for interpolation. It will automatically update its `Cool` objects so that the current redshift interval fits the given inputs.

Lastly, both `Cool` and `CoolManager` objects feature customizable error handling.

4 Evaluation

This section examines the performance of CHIPS and DIP. It compares the Delaunay based interpolation (Delaunay interpolation, DI) with multilinear interpolation (MLI) in terms of accuracy, consistency, speed and storage requirements. It further includes the results of two simple cosmological simulations made with DIP in OpenGadget3 (Springel 2005) and Gasoline2 (Wadsley et al. 2017).

The CHIPS algorithm has many adjustable hyperparameters (see section 3.2.1) that make it impossible to draw a complete comparison between DI and MLI. For this reason a representative data set based on Gasoline2 cooling tables was generated and compared to the equivalent grid. The Cloudy template file used in these calculations is attached in appendix A.

4.1 The Cooling Data Sets

The comparisons in section 4.2 use data sets based on Gasoline2 cooling tables (Kannan et al. 2016; Obreja et al. 2019). These cooling tables use radiation fields from old and young stars as additional parameters, containing four free parameters in total. This makes them a good example to study the performance of CHIPS and DIP as function of the dimensionality. These four parameters are:

- Temperature T (units K)
- Hydrogen density n_H (units cm^{-3})
- Φ_{old}, the multiplier for the SED produced by old stars (units $\mathrm{M}_\odot\mathrm{kpc}^{-2}$)
- Φ_{SFR}, the multiplier for the SED produced by young stars[6] (units $\mathrm{M}_\odot\mathrm{s}^{-1}\mathrm{kpc}^{-2}$)

The redshift was not used as free parameter, since the redshift is interpolated differently in DIP (see section 3.2.2). It was set to zero instead. Metallicity was set to solar metallicity.

This parameter space was chosen because it includes custom SEDs in addition to "regular" cooling parameters. At four dimensions it is also reasonably complex, but can still be analyzed easily.

Fig. 10 shows the different SEDs used in the cooling function calculations. All runs contained the cosmological microwave background (CMB) and an ultraviolet background (UVB). The chosen UVB was Haardt & Madau (2012) (a.k.a. HM12), which has the advantage of being directly available from within Cloudy. Both the CMB and the UVB depend on the redshift, which was constant at zero. Fig. 10 also shows the shape of the SEDs created by old and young stars. They are multiplied with the free parameters Φ_{old} and Φ_{SFR}, which gives the parameter space two of its four dimensions.

The spacings of the regular grid used for MLI can be seen in table 1. Note that Cloudy did not produce valid outputs for some of these parameter combinations, particularly in the low density regime ($10^{-9}\,\mathrm{cm}^{-3} \leq n_H \lesssim 10^{-5}\,\mathrm{cm}^{-3}$). 863 points did not yield valid results and were excluded, leaving a total of 61759. It took 5h 57m for Cloudy to evaluate these points on 60 cores. As plain text tables they require 2.3 MB of hard drive space.

CHIPS was run using the edges of this parameter space as base. The threshold for interpolation errors was set to 0.1 dex; this threshold had to be met for at least 90% of samples. The maximum error across all samples was 1 dex, and in each iteration 200 uniformly distributed random points were added to the mesh.

After 18 iterations the accuracy condition was fulfilled. The mesh contained about 26000 samples at this point. However, the maximum error condition was not satisfied yet, and the

[6]SFR stands for Star Formation Rate

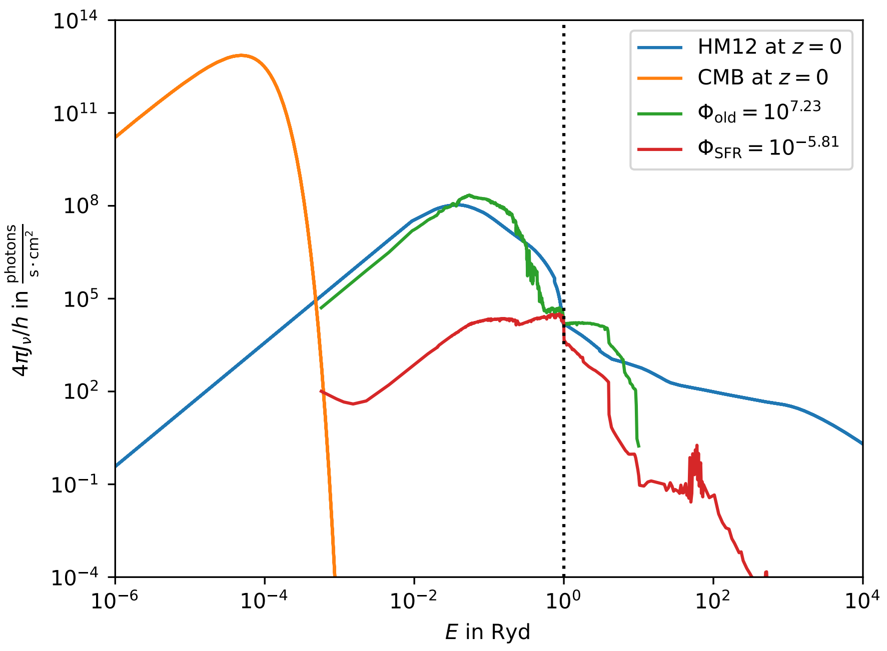


Figure 10: The different spectral energy distributions used in the cooling function simulations. The CMB and UVB are constant since $z = 0$. Here, the factors Φ_{old} and Φ_{SFR} are chosen such that their corresponding SEDs have the same photon flux as the UVB at 1 Ryd.

Parameter	Minimum	Maximum	Spacing	Number of Steps
T [K]	2	9	0.1	71
n_H [cm^{-3}]	-9	4	1	14
Φ_{old} [M$_\odot$kpc^{-2}]	6	12	1	7
Φ_{SFR} [M$_\odot$s^{-1}kpc^{-2}]	-5	3	1	9

Table 1: Spacings of the regular grid used for MLI. All values are given in dex. In total this 4-dimensional grid contains $71 \times 14 \times 7 \times 9 = 62622$ points.

program continued until it reached the maximum number of iterations. This occurred at iteration 20, when the mesh contained 31111 points. At this stage the largest error was 1.16 dex and the simulation had been running for 2h 59m on 60 cores.

The final triangulation contained over 927000 simplices and required 59.3 MB of hard drive space. The majority of this space is taken up by the triangulation, as the actual samples are only 1.4 MB in size. Unless otherwise specified, this is the data set used in section 4.2.

For testing purposes another run was conducted, using the data of the first simulation as starting point. It was allowed to run until the maximum error condition was fulfilled. This was reached within 7 iterations, after the sample count had grown to 57076. It took 2h 16m to create this extended data set on 60 cores.

The sample distribution of the CHIPS data set can be seen in fig. 11. It shows distinct overdensities and underdensities in different parts of the parameter space. The most notable of these is in the temperature dimension, near $T = 10^4$ K. The sampling here is particularly dense for high hydrogen densities ($n_H > 1\,\text{cm}^3$) and weaker radiation radiation fields ($\Phi_{\text{SFR}} < 10^0\,\text{M}_\odot\text{s}^{-1}\text{kpc}^{-2}$).

The hydrogen density n_H also shows interesting structures. Matching the temperature feature, its highest density sampling corresponds very strongly to the $T \approx 10^4$ K overdensity and weakly to $\Phi_{\text{SFR}} < 10^0\,\text{M}_\odot\text{s}^{-1}\text{kpc}^{-2}$, both at $n_H > 1\,\text{cm}^{-3}$.

The sampling is less dense for high temperatures ($T > 10^{7.5}$ K), as well as low hydrogen densities ($n_H < 10^{-5}\,\text{cm}^{-3}$). These underdensities are visible across the entire parameter space.

In conclusion, there seems to be a part of the parameter space near $T = 10^4$ K that is particularly difficult to interpolate at high densities. This matches the hydrogen feature of the cooling function (cf. section 2 and fig. 1). CHIPS samples this region thoroughly in order to achieve consistent results. MLI is expected to perform poorly here.

By contrast, CHIPS only sparsely samples the low density space. Presumably, MLI will deliver comparatively good results here due to the low amount of samples required.

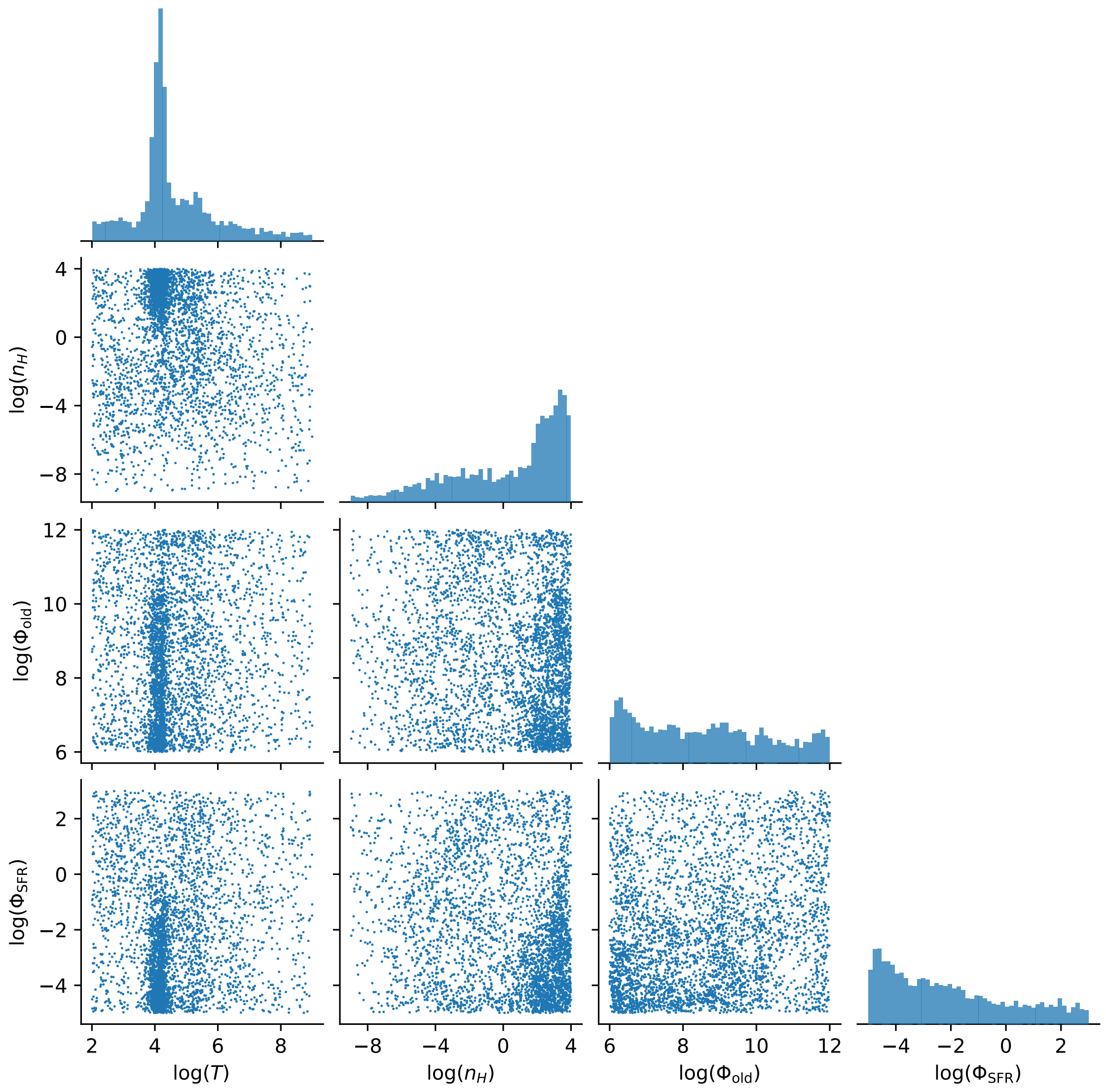


Figure 11: The distribution of samples generated by CHIPS. Only 4000 randomly chosen samples are shown for legibility. Clear structures related to the cooling function are visible, most notably at $T \sim 10^4$K.

4.2 Delaunay Interpolation vs Multilinear Interpolation

This section compares DI and MLI in terms of memory use, interpolation accuracy, interpolation consistency, and speed.

4.2.1 Memory Use

The most obvious difference between the regular grid and the irregular mesh is in the number of samples and the space they take up. While the irregular mesh only contains about half the number of points, it takes up almost 26 times the space due to the metadata that needs to be saved in the form of simplices and simplex relations.

During a simulation this data needs to be held in memory, taking up valuable resources that are not available to other parts of the program. This presents a hard limit on how large meshes can get.

In principle it is possible to avoid the issue by not storing the mesh. Instead, one could use e.g. a ball tree to find the n nearest neighbors of the interpolation point and build a small triangulation on these points. Then, this small scale triangulation could be used for interpolation. However, there is a significant computational cost associated with this method. Not only does it need to calculate these small meshes, but it also needs to ensure that the interpolation point is actually contained within those meshes. Doing so efficiently is a nontrivial task. The increased runtime caused by these problems was deemed unacceptable for this implementation.

Another possible solution might be parallelization. In the current implementations, each OpenGadget3 and Gasoline2 thread contains its own, fully independent DIP instance. It would be possible to save a significant amount of memory by having one or more threads entirely dedicated to cooling function interpolation. However, this introduces synchronicity problems, and because cooling function interpolation takes a non-negligible amount of time it might become a problematic bottleneck.

For now, DIP implements the simplest solution that outsources and caches as many calculations as possible in order to minimize the impact on the runtime of cosmological simulations that use it. If memory is a limiting factor for their execution, MLI is currently the better choice.

4.2.2 Interpolation Accuracy

In order to gauge the accuracy of both interpolation methods, 45000 uniformly distributed random samples were drawn in the 4-dimensional parameter space. Their values were interpolated with DI and MLI, and the interpolation was compared to the values given by Cloudy.

Fig. 12 shows the cumulative error distribution of both approaches. In the ideal case, where almost all points are interpolated correctly, most errors are small and the curve would quickly rise and approach unity. This is the case for both interpolation methods to a limited extent.

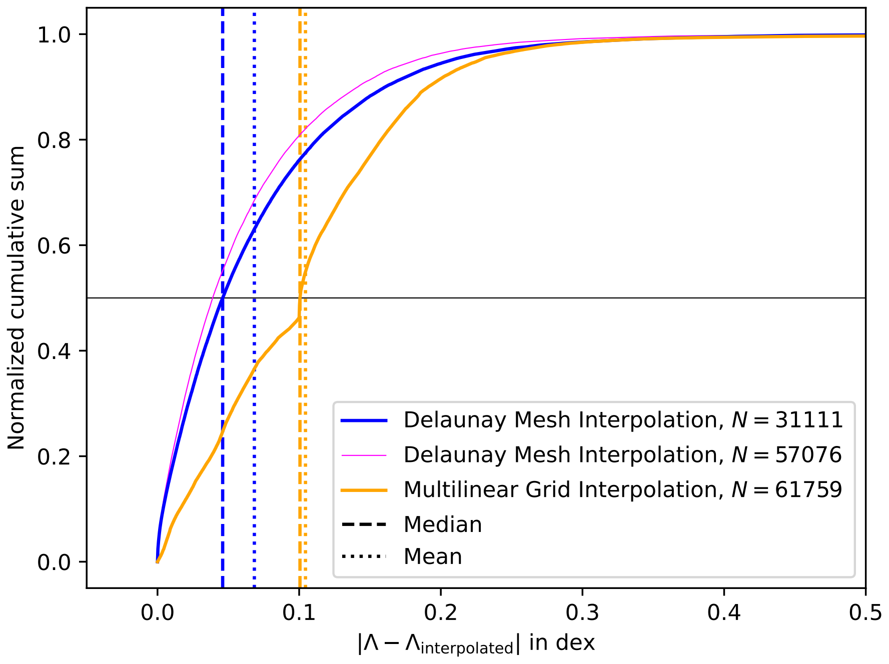


Figure 12: The cumulative sums of the differences between interpolated and actual cooling values for both approaches. MLI shows a characteristic kink near 0.1 that stems from MLI's problems at low densities (see section 4.2.3). DI performs much better, but mostly improves on the lower end of the spectrum. The extended CHIPS data set is also shown for reference. It improves the error distribution further, but compared to the regular data set the improvement is small.

In terms of accuracy, DI clearly outperforms MLI. About 75% of Delaunay interpolations have an error of 0.1 dex or smaller, as opposed to only 45% of multilinear interpolations. The error distribution of DI is superior to MLI mostly in the 0 dex to 0.2 dex interval. This indicates that DI mostly improves on those parts of the parameter space that were already interpolated somewhat accurately before. At larger errors the distributions approach each other, and they are virtually indistinguishable beyond 0.3 dex. The two interpolation methods seem to perform similarly well in this area.

This has implications on how to choose hyperparameters for CHIPS: If the main advantage of Delaunay interpolation lies in improving the lower end of the error distribution, unnecessary iterations might not have to be spent on trying to improve the overall maximum error. By achieving results of similar quality with a fraction of the samples, computation time and hard drive space can be freed up to e.g. add more dimensions.

The magenta curve further underlines this point: It shows the error distribution produced by using the extended Delaunay mesh (see section 4.1), containing almost as many data points as the regular grid and almost twice as many as the regular Delaunay mesh. Despite this it shows only limited improvement over the smaller mesh.

In fact, the number of samples can be reduced even further with acceptable losses in accuracy. Testing revealed that subsampling the CHIPS data set with 20000 points still outperforms the regular grid (see also appendix B). This reduced the size of the associated triangulation to 11.4MB.

Fig. 12 also shows the medians (dashed lines) and averages (dotted lines) of the two error distributions. Both the median and the mean error of the multilinear interpolation are at about 0.10 dex, while the Delaunay interpolation's median is at 0.046 dex with an average of 0.068 dex. This constitutes an error reduction of 54% and 35%, respectively.

Lastly, it should be noted that the x-axis cuts off at 0.5 for plotting reasons. The largest error produced by the MLI interpolation was 1.70 dex, while the largest error of the Delaunay interpolation was 27% smaller at 1.25 dex.

Overall, DI vastly improves over MLI in terms of accuracy. Both the average and maximum errors are significantly reduced while only using a fraction of the samples compared to the grid based approach. In principle it should be possible to fulfill arbitrary accuracy requirements by increasing the number of samples. However, adding more samples quickly leads to diminishing returns. Since DI gets a lot more performance out of less samples it would be smarter to keep the sampling density low and instead add more parameters in order to model the cooling function more accurately.

4.2.3 Interpolation Consistency

In order to judge the interpolation consistency, the spatial error distributions of both algorithms were plotted. The results are shown in fig. 13. The upper row shows the MLI error distribution for selected parameter pairings. The lower row shows the matching DI error distribution. The color scale is cut off at 0.5 dex, matching figure 12.

In all pairings there are clearly visible structures. They are mostly correlated to the temperature, which is why the temperature axis was chosen as the y-axis for all plots. Clear streaks can be seen, corresponding to major temperature dependent cooling function features, the largest being near 10^4 K. This is where the temperature becomes high enough to ionize hydrogen, which can then contribute to the cooling (cf. section 2.1 and fig. 1). Because the gradient of Λ is large here, small differences in parameters lead to large differences in cooling. This in turn leads to larger errors.

These errors are most pronounced for very high hydrogen densities, where cooling is strongest. Smaller temperature features are also visible at roughly $10^{5.5}$ K and between 10^6 K and $10^{6.5}$ K. At solar metallicity, these match intervals where the cooling function has large gradients.

At $10^{5.5}$ K the peak of the cooling function is followed by a downward slope as oxygen rapidly loses its cooling power. Between 10^6 K and $10^{6.5}$ K is another prolonged slope, where iron stops cooling efficiently but before bremsstrahlung cooling can kick in.

These peaks of the spatial error distribution match the regions of high density sampling in the CHIPS data set (see section 4.1 and fig. 11). Both are most pronounced for high hydrogen densities and at $T \approx 10^4$ K.

The benefits of this high density sampling can be seen in the lower row of fig. 13. While most of the features apparent for MLI are still visible, for DI they are much weaker.

One surprising benefit of the Delaunay interpolation is in the low density regime ($n_H < 10^{-4}\,\mathrm{cm}^{-3}$). The CHIPS data set is sampled only sparsely here, indicating that interpolation in this region is comparatively easy. Despite this, MLI struggles in this part of the parameter space, while DI produces some of the most accurate results here overall. This is an unexpected result. Testing revealed that MLI consistently produces Λ values that are too large by 0.1 dex to

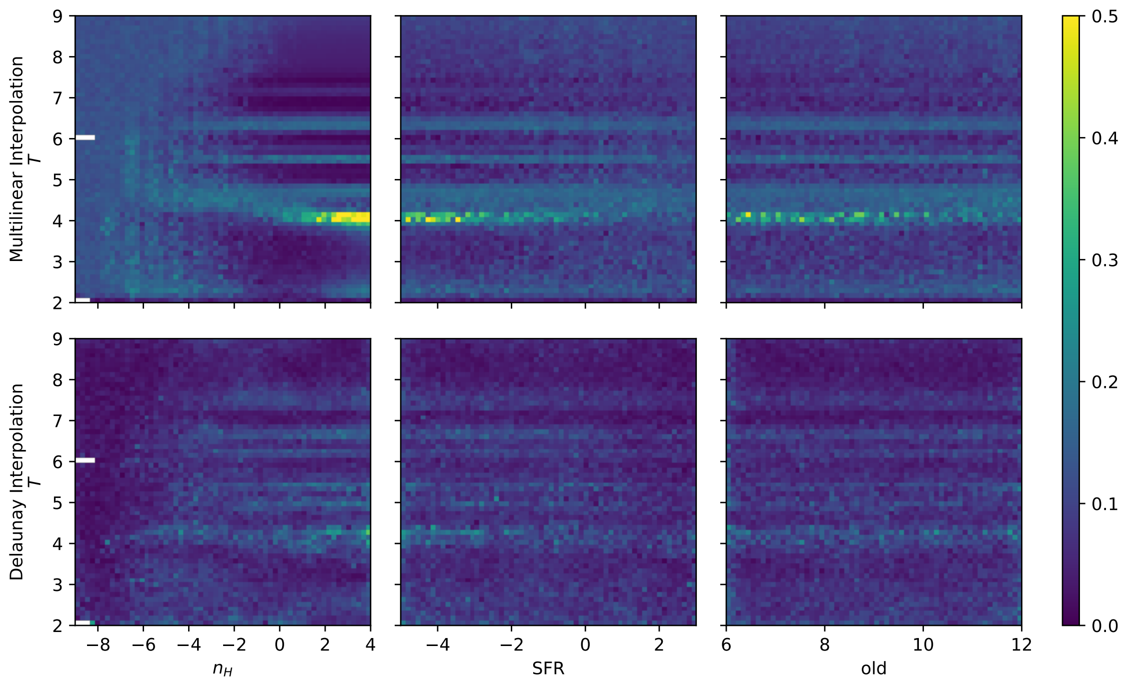


Figure 13: Heatmap of the spatial error distributions for certain parameter pairings. The upper row shows multilinear interpolation while the lower row shows Delaunay interpolation. Structures matching the cooling function are clearly visible in both, but are much more pronounced for MLI. For clarity, six parameter pairings with less pronounced structures are omitted. The error scale is cut off at 0.5 dex, matching figures 12 and 14.

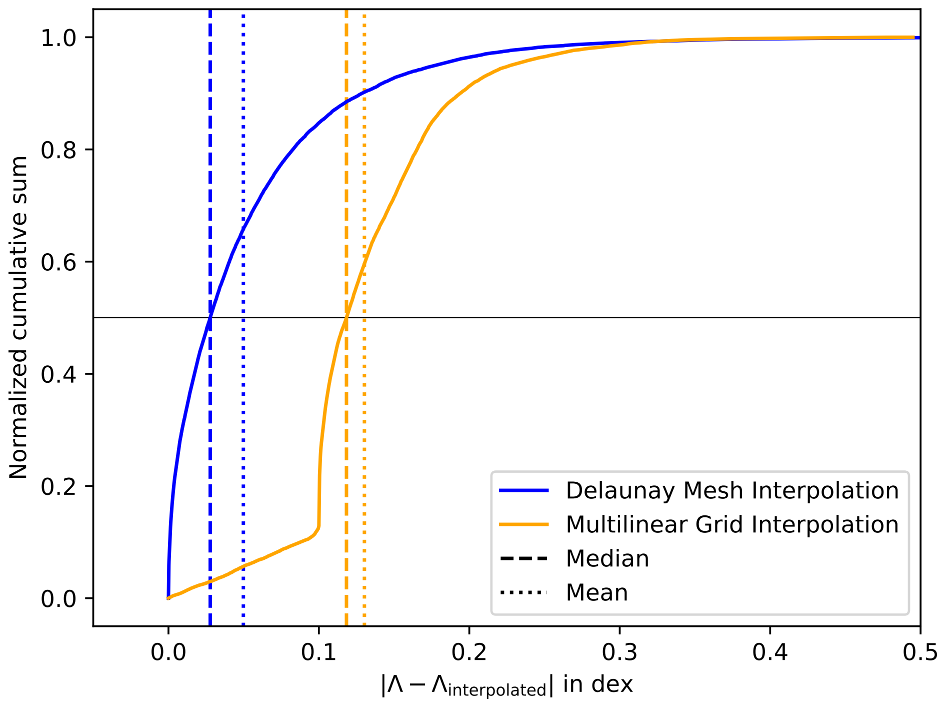


Figure 14: The figure shows the cumulative error distribution for hydrogen density $n_H < 10^{-4}\,\mathrm{cm}^{-3}$. MLI performs very poorly here and is rarely more accurate than 0.1 dex. Meanwhile, DI actually performs better than on the rest of the parameter space. At median errors of 0.028 for DI and 0.12 for MLI this is an improvement of 77%.

0.2 dex in this region. The DI errors are much more evenly distributed, leading to values that are both too large and too small.

Fig. 14 reproduces the cumulative error distribution from figure 12, taking into account only points with hydrogen densities $n_H < 10^{-4}\,\mathrm{cm}^{-3}$. The MLI curve shows a very distinctive kink at 0.1 dex. It appears that for some reason, multilinear interpolation is consistently wrong by about 0.1 dex at low hydrogen densities. This explains the related feature in figure 12, but the reasons for this are unknown.

Meanwhile, the cumulative error distribution of DI is almost unchanged compared to figure 12. In fact, the DI curve proved to be remarkably consistent when plotted for different regions of the parameter space, whereas the MLI curve would change by a lot. In this particular case, the median error of DI is 77% lower than the median error of MLI.

MLI does have the upper hand in the T vs. n_H pairing outside of the error peaks. It produces very accurate results in this region, while DI is noticeably more noisy.

Overall, Delaunay based interpolation does not only deliver more accurate values than multilinear interpolation, but does so consistently across the entire paramater space. Regions that produce extreme errors for MLI are much less pronounced, and regions of low hydrogen density that showed medium sized errors before are now much more accurate. Further, this was all achieved with only half the amount of samples, and could potentially require far less (see section 4.2.2).

4.2.4 Speed

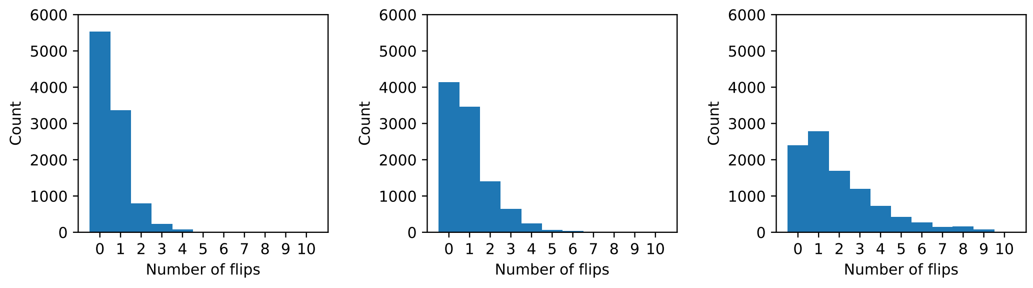
This section evaluates the performance of the Delaunay interpolation. It examines both the implementation independent performance of the simplex flipping algorithm and the runtime of the specific DIP implementation (see section 3.2.2).

The regular grid used for MLI was missing some data points in regions of low hydrogen density (cf. section 4.1). Since the MLI reference implementation used in these benchmarks was not equipped to deal with missing data points, the test was limited to the denser half of the parameter space. This should not affect the results.

In theory, DI provides an exponential speedup over MLI. While MLI runtime scales exponentially with the number of dimensions at $O(2^d)$ (see section 2.3), the specific interpolation algorithm implemented in DIP only scales polynomially at $O(Fd^3)$. Here, F is the number of steps executed in the simplex flipping algorithm (see section 3.2.2). This speedup should be noticeable at $d \gtrsim 10$.

In practice such a high number of dimensions is never used, since the execution time is going to be prohibitively slow, independent of the interpolation algorithm. At more typical dimensionalities of $d \lesssim 6$ MLI is the faster interpolation method.

It is difficult to estimate the impact of F on the runtime theoretically since it depends on a variety of factors, such as the density of the triangulation, the dimensionality, and the shape of the sample distribution. For this reason empirical tests were run instead. For several d, 10000 points were interpolated using DIP, and the number of flips per interpolation recorded. The results are shown in fig. 15. The two and three dimensional data sets used for this test were created like the main CHIPS data set. The 3D data set omitted one of the custom SEDs, while the 2D data set omitted both.



(a) 2D (b) 3D (c) 4D

Figure 15: Distribution of the number of flips in the simplex flipping algorithm for different dimensions.

The histograms show that the ball tree usually finds a good starting point for the simplex search. In 2D and 3D the first candidate simplex already contains the target point most of the time. In 4D there is usually at least one flip required.

The variance in the number of flips does increase strongly with the number of dimensions. In two dimensions there is a sharp peak at zero flips, and only a minority of interpolations require more than two flips. In four dimensions there is a significantly larger spread, with interpolations frequently requiring anywhere between zero and five flips, sometimes up to nine.

As the number of dimensions increase, the performance of the simplex flipping algorithm not only worsens because the matrix solving techniques employed at each step scale as $O(d^3)$, but also because the average number of flips required rises with d. Further, the variance in runtime increases with the variance in F at higher dimensions.

It is possible to limit the maximum runtime for a single interpolation by setting a maximum number of flips, $F_{\max}$. This parameter was originally intended to prevent infinite loops in earlier versions of the program and must be used with care. If the simplex search is cut short, DIP will use whichever simplex was found last to extrapolate a cooling value. This is much less accurate than the interpolation that is performed when the correct simplex is identified. The errors introduced by artificially limiting the number of flips might outweigh the performance gain.

Ideally, $F_{\max}$ is set to a value that is only reached in exceptional cases. A good choice for $d \leq 4$ is $F_{\max} = 10$.

To measure the actual runtimes, benchmarks were run for several d. For each dimension 10000 points were interpolated and execution time measured. This process was repeated ten times under identical conditions. The results were very consistent and can be seen in fig. 16.

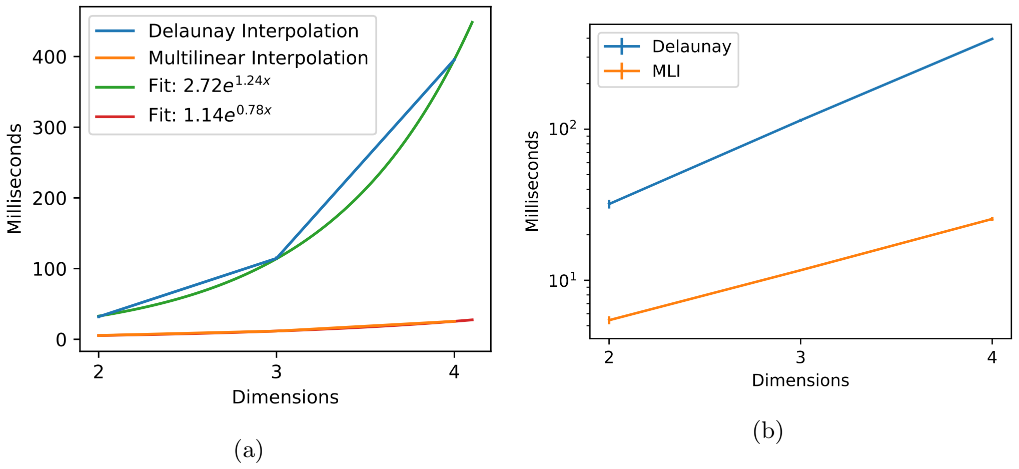


Figure 16: (a) DI and MLI execution times at different dimensions in milliseconds. Exponential functions were fitted to the data. Error bars are plotted, but are small due to the consistent execution times. (b) The same plot in log space, without the fits.

In 2D, MLI takes about 5.5 ms to interpolate these points, while DI is almost 6 times slower at 31 ms. In 4D, this has increased to a factor of 16, as DI requires almost 400 ms compared to 25 ms for MLI.

It is clear that DI struggles to keep up with MLI for "large" d as they are typically seen in simulations. This is unfortunate, but may still present a worthy tradeoff if the additional accuracy delivered by DIP is desired. The overall impact of the cooling interpolation on cosmological simulations, while not negligible, is small compared to other major tasks, such as the calculation of the particle trees. A large relative increase in calculation time here can be manageable.

4.3 Implementation into OpenGadget3 and Gasoline2

To test the impact of DIP on actual cosmological simulations it was integrated into Gasoline2 and OpenGadget3.

4.3.1 Gasoline2

The cooling implementations currently available in Gasoline2 consider the primordial species separately. Their non-equilibrium cooling is calculated at runtime, and only the metal contribution to Λ is interpolated from pre-computed tables. Due to an oversight this was not considered in the first integration of DIP.

The Λ tables generated by CHIPS also contain the effect of the primordial species. This means that their cooling contribution was counted twice, leading to erroneous results. This issue will be corrected in the future, but for the results shown below this important caveat has to be taken into account.

This first version of Gasoline2 integrated with DIP was used to run a low resolution zoom-in galaxy simulation from $z \approx 6$ to $z \approx 2$, which took about ten days to complete. The results of this simulation are compared to a reference run, which used identical initial conditions and simulation setup, but differed in the cooling implementation. The reference run used the cooling described in Obreja et al. (2019), which includes the effects of photoionization due to young stars and old stars with SEDs like in fig. 10. It further includes three bins of temperature for the thermal emission from hot halo gas, which only kicks in at $z < 3$. The particular galaxy in these simulations contains no gas at the temperatures relevant to thermal emission down to $z \approx 2$, making these latter parameters effectively nonexistent.

The other important difference between the cooling implementation is the UVB. Obreja et al. (2019) uses the Faucher-Giguère et al. (2009) UVB, while DIP uses Haardt & Madau (2012). However, this should not be an issue as the choice of UVB only affects the cooling very little (Lüders 2018).

Fig 17 shows a phase diagram of the simulation particles after the simulation was complete, along with the same phase diagram for a reference simulation that used MLI.

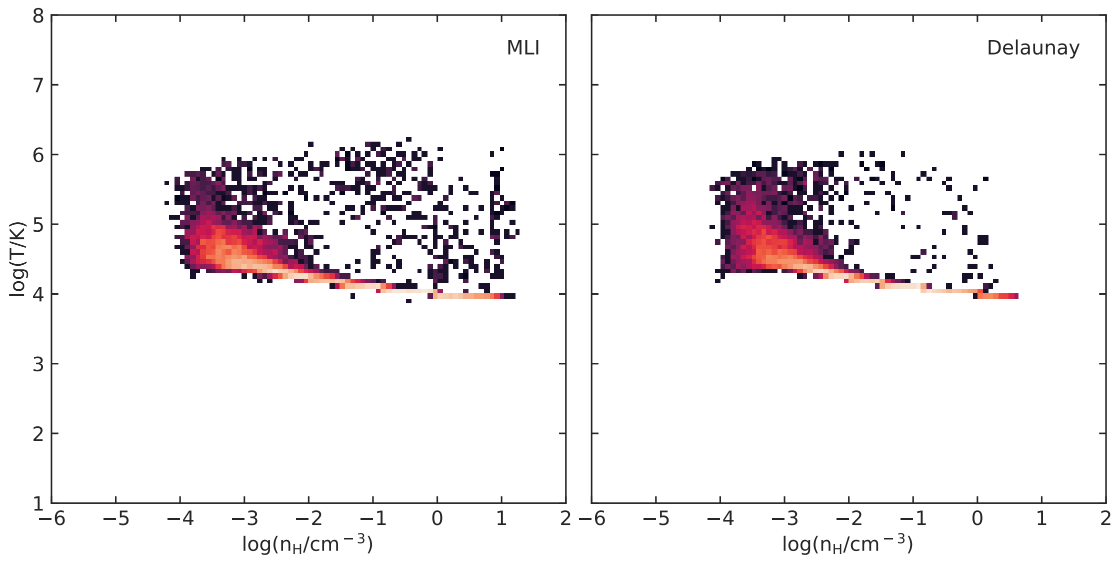


Figure 17: Phase space diagrams of two Gasoline2 simulations, showing temperature and hydrogen density. Both simulations started with the same initial conditions, but used different cooling interpolations. The left simulation used MLI, while the right simulation used Delaunay based interpolation. Both figures show the phase space of the gas inside the virial radius at $z = 2.36$.

Despite the issues with the DIP implementation, both diagrams show similar results. They both show a distinct distribution of particles along $T = 10^4$ K for hydrogen densities $10^{-2.5}\,\mathrm{cm}^{-3} < n_H < 10^1\,\mathrm{cm}^{-3}$. This corresponds to the temperature region identified as being particularly difficult to interpolate with MLI in the previous sections, although the lower hydrogen densities mean that the worst errors are avoided.

At lower hydrogen densities this distribution fans out into a cloud in the phase space. In both cases it is within the ranges $10^{-4}\,\mathrm{cm}^{-3} < n_H < 10^{-2}\,\mathrm{cm}^{-3}$ and $10^4\,\mathrm{K} < T < 10^6\,\mathrm{K}$, although in the MLI figure it extends to slightly lower hydrogen densities. This means that in both cases the particle distribution stops just above the low hydrogen density regime that causes consistent errors for MLI (see section 4.2).

One notable difference between the two approaches is the number of outliers. The MLI simulation shows a consistent distribution of outliers in the regions $10^{-2}\,\mathrm{cm}^{-3} < n_H < 10^1\,\mathrm{cm}^{-3}$ and $10^4\,\mathrm{K} < T < 10^6\,\mathrm{K}$. While these are also present for DI, they are much less pronounced. These outliers are gas particles impacted by supernova feedback. In the MLI case the tail to large densities extends beyond 10 cm^{-3}, which is the threshold for star formation, while in the Delaunay it does not, which means that the new simulation likely forms less stars. This leads to less supernovae and subsequently less outliers.

Figure 18 shows the mass distribution of the particles in relation to the hydrogen densities and temperatures, at high redshifts early in the simulation and low redshifts late in the simulation. In general, the DI and MLI curves are similar.

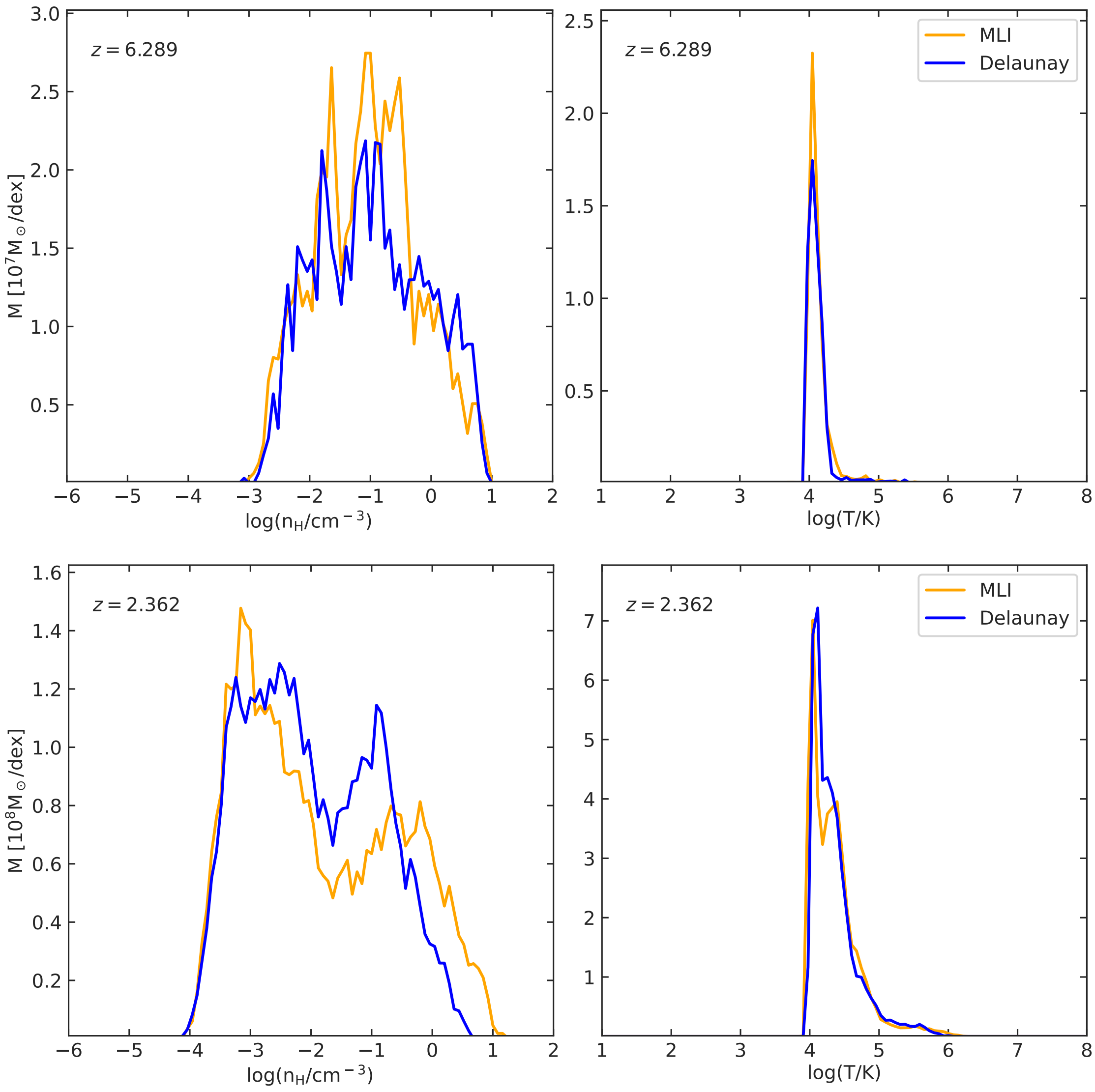


Figure 18: Evolution of the density (left) and temperature (right) mass weighed distribution functions for the gas inside the virial radius of the simulation shown in fig. 17. The upper row shows the simulations at $z = 6.29$, near the beginning of their runs. The bottom row shows them at $z = 2.36$ near their end.

In the hydrogen density relation there is a shift from a single major peak around 10^{-1} cm^{-3} at $z = 6.29$ to a double peak at 10^{-3} cm^{-3} and 10^{-1} cm^{-3}. This suggests that there are two major groups of simulation particles, those that move to lower densities over time and those that do not change much or slightly increase in density. Notably, in the MLI case the low density peak is an order of magnitude larger than the high density peak, whereas in the DI case they are of almost equal size. In the DI simulation the high density peak is also at lower densities than in the MLI simulation.

The mass distribution relative to the temperature only shows a single, narrow peak near $T \approx 10^4$ K. This peak broadens slightly over time, and in the MLI case develops a smaller sub-peak at $T = 10^{4.5}$ K. While the DI simulation's curve is slightly less sharp at $z = 6.29$, at $z = 2.36$ it matches the MLI curve very well. This is unexpected, since the faulty DIP implementation should produce larger cooling values and thus lower temperatures.

In conclusion, despite the issues with the DIP implementation, both simulations show very similar results. Since the faulty DIP implementation is expected to produce cooling values that are too large, this suggests that the current multilinear interpolation leads to some overcooling. This matches some findings of the previous section, where MLI produced Λ values that were consistently too large for certain regions. More work is required to investigate this situation further.

4.3.2 OpenGadget3

For time reasons it was not possible to execute a full Gadget-simulation. However, it was possible to run a small test case that only evolves the simulation for two timesteps. It used a slightly different parameter space than the Gasoline2 case; there were no custom SEDs, instead it features a single metallicity parameter Z. This made it a three dimensional parameter space overall. The redshift was constant at $z = 1$.

There are already several cooling interpolation implementations in OpenGadget3. These include Sutherland & Dopita (1993) and Wiersma et al. (2009), which are compared against DIP here. They differ in several aspects.

Figure 19 illustrates some of these differences. It shows the absolute value of the net cooling function $|\Lambda_{\mathrm{net}}|$, which includes both cooling and heating effects (cf. section 2), as a function of the metallicity. Crucially, only those simulation particles are drawn which had temperatures in the range $10^5\,\mathrm{K} < T < 5 \times 10^5\,\mathrm{K}$, where both hydrogen density and metallicity have a particularly large effect on the cooling.

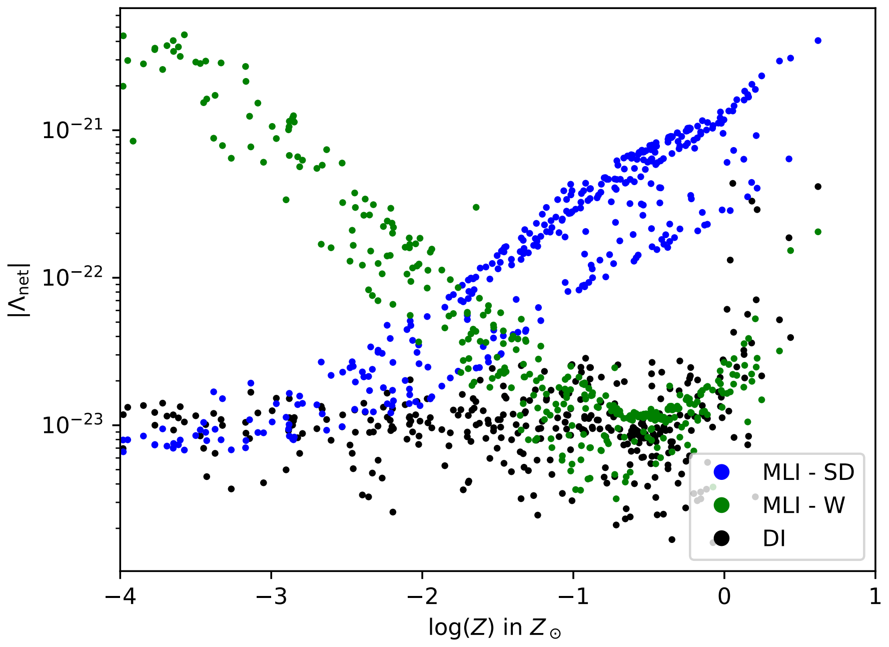


Figure 19: The net cooling function, which takes heating effects into account, as a function of metallicity Z. Only simulation particles in the temperature range $10^5\,\mathrm{K} < T < 5 \times 10^5\,\mathrm{K}$ are shown. The colors correspond to different cooling implementations.

Sutherland & Dopita (1993) shows a clear linear dependence between $|\Lambda_{\mathrm{net}}|$ and Z in log space, which is the expected behavior if no other parameters come into play. By contrast, Wiersma et al. (2009) shows a clear inversely linear relationship, which turns positive near $Z = -0.5$. The DIP values are constant for most metallicites, and rise for $Z > -0.5$.

These stark differences might be a consequence of the different assumptions that went into these cooling tables. In particular, Sutherland & Dopita (1993) does not take hydrogen density n_H into account except at very low temperatures, whereas Wiersma et al. (2009) and DIP do. It is possible that low hydrogen densities counteract the effects of high metallicity. This does not explain the differences between DIP and Wiersma et al. (2009), however, and there is reason to assume that the latter does not make use of its n_H-dependence in OpenGadget3 (see below).

Figure 20 shows $|\Lambda_{\mathrm{net}}|$ as a function of temperature. This time the entire particle distribution is plotted. For each of the three cooling codes two distinct branches are visible: To the left, there is the heating branch, where heating effects are stronger than cooling effects; to the right, there is the cooling branch, where cooling effects are stronger. The pole where these two branches meet demarcates the equilibrium temperature for that cooling implementation.

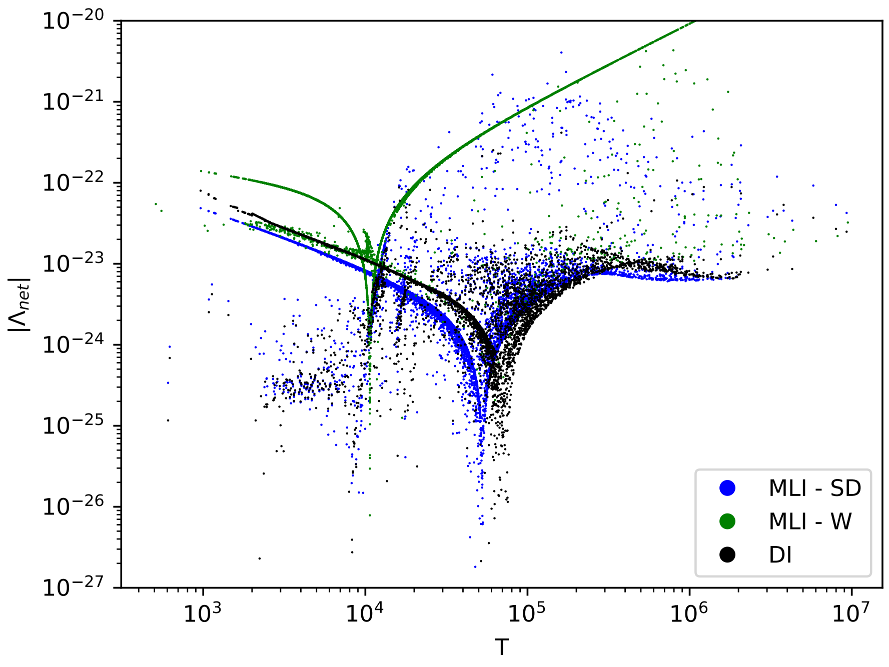


Figure 20: The net cooling of the OpenGadget3 simulation particles as a function of temperature. For technical reasons only 10000 out of the 27266 total particles are shown.

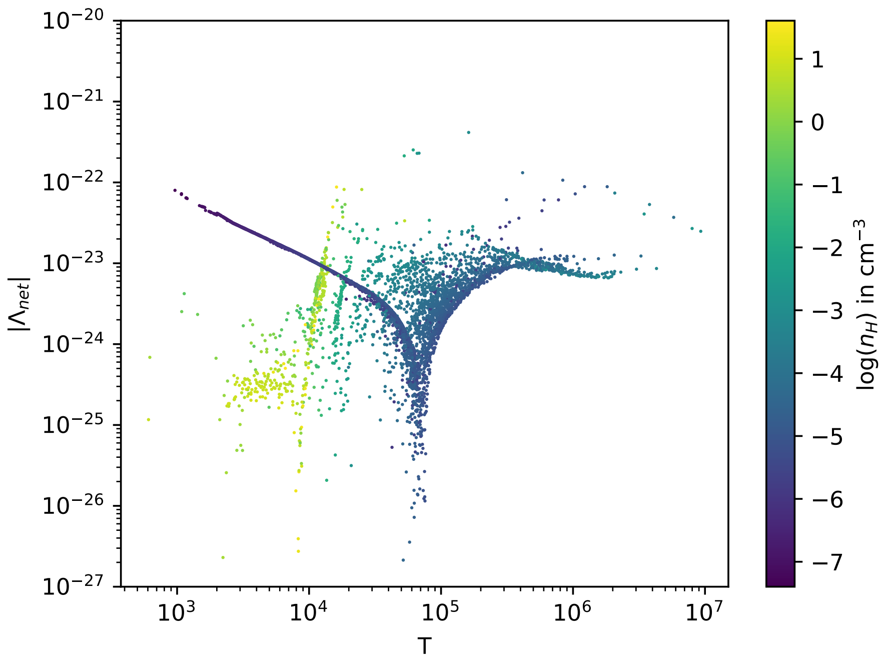


Figure 21: The net cooling of the OpenGadget3 simulation particles as a function of temperature. They are colored according to their hydrogen density.

The DIP and Sutherland & Dopita (1993) distributions are very similar. The equilibrium temperature of Sutherland & Dopita (1993) is at about $10^{4.4}$ K, whereas the equilibrium temperature of DIP is slightly higher at around $10^{4.6}$ K. Accordingly, DIP shows a slightly more dominant heating branch. This behavior is consistent with the minor overcooling found with Gasoline2 in section 4.3.1.

The metallicity dependence of the Sutherland & Dopita (1993) tables is also visible in this figure. There is a diffuse cloud of particles in the 10^4 K to 10^6 K range that contains outliers. While some of these stem from DIP, the majority are in fact the same high metallicity particles produced by the Sutherland & Dopita (1993) tables in fig. 19.

The distribution of the particles simulated using the Wiersma et al. (2009) cooling tables is a major outlier. It leads to a much lower equilibrium temperature at 10^4 K, and cooling that is over two orders of magnitude greater than the other two codes for larger temperatures. The particle distribution is also much narrower and shows only few outliers.

The reasons for the magnitude differences are not quite clear. It is unlikely that flawed calculations are the reason, since Wiersma et al. (2009) uses Cloudy simulations to calculate values, like DIP. It is possible that there is some issue with their implementation into OpenGadget3, or that their calculations are done with different units internally that are not properly converted for output files.

However, the narrowness of this distribution suggests that the Wiersma et al. (2009) implementation does not make proper use of the hydrogen density values contained within those tables. Figure 21 shows the particle distribution created using DIP, colored according to the particles' hydrogen densities. There are some clearly visible structures.

First, the particles that experience the strongest net heating are also the particles that have the lowest hydrogen densities. This is expected, since collisional cooling is not very effective at low densities.

Further, there is a noticeable aggregation of high density particles that experience significantly stronger cooling than the other particles. They form two streaks near $T = 10^4$ K, which are likely part of a larger cooling branch that exists at this hydrogen density. The equilibrium temperature for these particles would also be near $T = 10^4$ K. It is possible that the Wiersma et al. (2009) does erroneously not take n_H into account, and uses a fixed hydrogen density at around $n_H = 1\,\mathrm{cm}^{-3}$ instead.

In conclusion, DIP adds yet another cooling implementation to the ones that already exist within OpenGadget3. Comparisons to Sutherland & Dopita (1993) and Wiersma et al. (2009) suggest that some overcooling may be present here, too, and that Sutherland & Dopita (1993) is likely the more accurate of the existing cooling codes.

5 Conclusion

Accurate calculation of the cooling function Λ is an important part of modern cosmological simulations. Since these calculations are expensive, precomputed cooling tables are used to interpolate values that approximate Λ in a fraction of the time.

However, the conventionally used multilinear interpolation (MLI) is increasingly insufficient to interpolate the cooling function as cosmological simulations grow in physical accuracy. While fast, their simplistic grid based approach produces values that can be off by an order of magnitude and requires an exponentially rising number of samples as the cooling model grows in complexity.

This thesis shows that leaving regular grids behind is a valid approach to overcome this issue. Two new publicly available pieces of software, CHIPS and DIP, have been developed to make use of amorphous meshes, in order to increase the interpolation accuracy and reduce the required number of samples. They use Delaunay triangulations and barycentric coordinates in order to achieve this goal (see section 3.1).

The Cloudy based Heuristic and Iterative Parameter space Sampler (CHIPS) is a python package, that uses an adaptive sampling algorithm to focus the sampling of the parameter space on those regions, that require particular attention to achieve consistent results (see section 3.2.1). It provides plenty of options to give the user the control they need over their sampling, with multiple optional exit conditions, the ability to use custom SEDs as parameters, and the option to load data from previous runs as starting point for new ones.

The Delaunay Interpolation Program (DIP) is a C/C++ library that utilizes the data generated by CHIPS (see section 3.2.2). It uses a custom "simplex flipping" algorithm to identify the simplex within a triangulation that contains a given target point, and uses this simplex to interpolate Λ. It makes use of modern object oriented programming principles, and provides special redshift interpolation if desired by the user, as well as an interface layer that allows it to be used by pure C code.

Section 4 examines the performance of these algorithms. Section 4.1 demonstrates that the adaptive sampling algorithm implemented in CHIPS works as expected, while section 4.2 shows that Delaunay based interpolation (DI) is not only more accurate than MLI, but that it also requires far less samples. It cuts down the peak errors in certain regions by an order of magnitude, and provides a reduction of the median error by as much as 77% in certain regimes, all while using only half the number of points.

However, the two major downsides of DI are its runtime and memory usage. Despite having theoretically superior time complexity, DI is several times slower than the conventional method due the dimensionalities at which the codes operate. To keep execution times reasonable, the entire triangulation needs to be cached in memory at runtime. This offsets the advantage gained from requiring a lower number of samples.

To demonstrate the practical applicability of the mesh based approach, DIP was integrated into two large cosmological simulation codes, Gasoline2 and OpenGadget3. Unfortunately, due to time constraints there were some issues with the Gasoline2 integration that could not be resolved, and it was not possible to execute a full run with OpenGadget3.

Despite this, it could be shown that DIP is stable even for large simulations, and that the principles behind Delaunay based interpolation work. In spite of some flawed data it could be seen that conventional Gasoline2 interpolation likely leads to overcooling. The comparison to the existing OpenGadget3 cooling implementations showed that DIP produces realistic results in practical applications, and further supports the notion that existing cooling codes slightly overestimate cooling effects.

The full capabilities of Delaunay based Λ interpolation has not yet been explored. Future work should attempt to find a more efficient interpolation algorithm that does not rely on expensive cached triangulations. Since barycentric coordinates will always require solving matrices, it is unlikely that an implementation as fast as MLI can be found for $d \leq 10$. However, an $O(d^3)$ algorithm that does not require caching might be possible and would already constitute a huge improvement.

It might also be possible to use CHIPS to provide training data for a machine learning approach to interpolate Λ; it could help such an approach to learn the structures and pitfalls of the cooling function better than if it only had grid based data to learn from, and neural networks are generally efficient to evaluate, sidestepping the performance issue.

Either way, CHIPS and DIP should prove useful tools for future cosmological simulations.

A Cloudy template

Variations of the following Cloudy input file were used for the simulations in this thesis:

```
CMB redshift {z}
table HM12 redshift {z}
metals {Z} log
hden {nH}
constant temperature {T}
stop zone 1
iterate to convergence
print last
print short
set save prefix "{fname}"
save overview last ".overview"
save cooling last ".cool"
```

The variables in curly braces are filled in using python string formatting.

B Cumulative Error Distribution for Different Sample Sizes

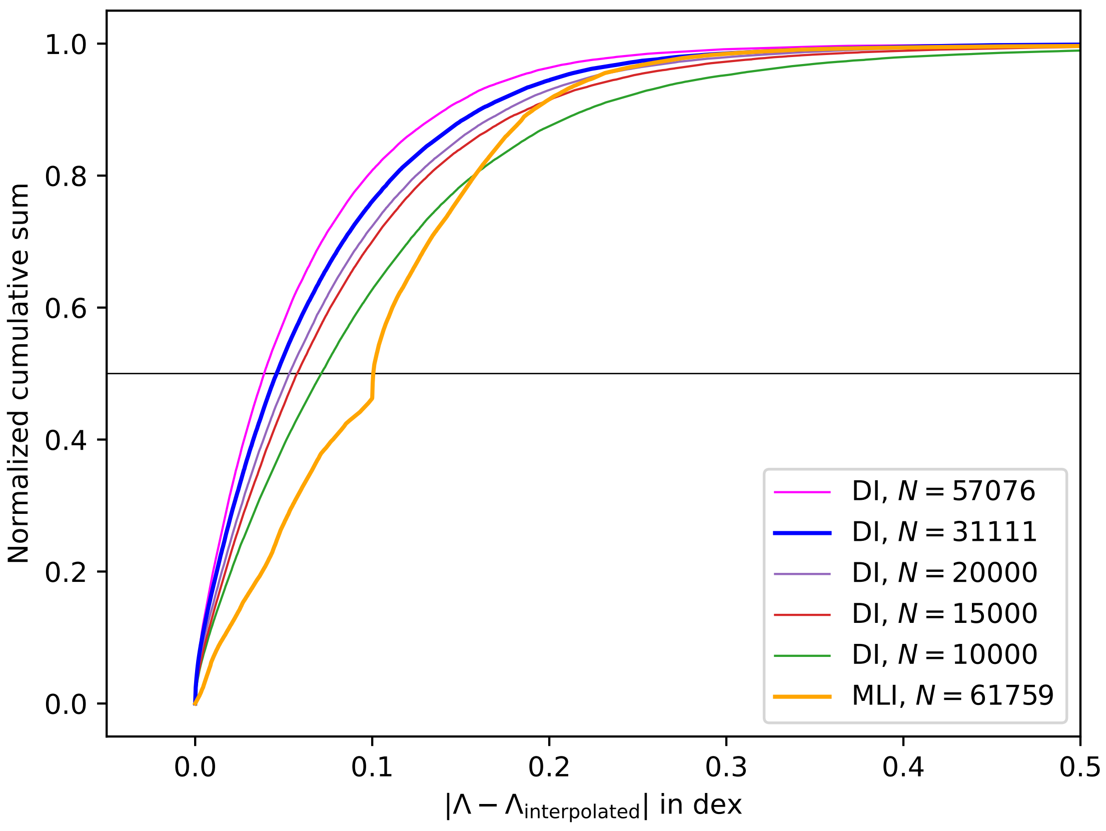


Figure 22: The cumulative error distribution for Delaunay based interpolation at different sample sizes, performed as in section 4.2.2. From left to right, in descending order of samples counts: Extended CHIPS data set (magenta), regular CHIPS data set (blue), subsampled CHIPS data set with $N = 20000$ (purple), subsampled CHIPS data set with $N = 15000$ (red), subsampled CHIPS data set with $N = 10000$ (green), and MLI data set for reference (orange). All subsampled data sets used the regular CHIPS data set (blue curve) as base. The CHIPS data sets perform as well as or better than MLI down to $N = 20000$. This is only a third of the samples used for the regular grid.

References

Barber C. B., Dobkin D. P., Huhdanpaa H., 2013, Qhull: Quickhull algorithm for computing the convex hull (ascl:1304.016)

Berg M., Kreveld M., Overmars M., 2008, Computational Geometry: Algorithms and Applications, doi:10.1007/978-3-540-77974-2.

Bridson R., 2007, in ACM SIGGRAPH 2007 Sketches. SIGGRAPH '07. Association for Computing Machinery, New York, NY, USA, p. 22–es, doi:10.1145/1278780.1278807

Brown K. Q., 1979, Information Processing Letters, 9, 223

Coxeter H., 1969, Introduction to geometry, 2nd Ed.. John Wiley and Sons

Dopita M. A., Sutherland R. S., 2003, Astrophysics of the diffuse universe

Draine B. T., 2011, Physics of the Interstellar and Intergalactic Medium

Farebrother R., 1988, Linear Least Squares Computations, doi:10.1201/9780203748923.

Faucher-Giguère C.-A., Lidz A., Zaldarriaga M., Hernquist L., 2009, ApJ, 703, 1416

Ferland G. J., et al., 2017, Rev. Mexicana Astron. Astrofis., 53, 385

Gnat O., Ferland G. J., 2012, ApJS, 199, 20

Gnedin N. Y., Hollon N., 2012, ApJS, 202, 13

Guibas L. J., Knuth D. E., Sharir M., 1992, Algorithmica, 7, 381

Haardt F., Madau P., 2012, ApJ, 746, 125

Hormann K., 2014, in Fasshauer G. E., Schumaker L. L., eds, Approximation Theory XIV: San Antonio 2013. Springer International Publishing, Cham, pp 197–218

Jeans J. H., Darwin G. H., 1902, Philosophical Transactions of the Royal Society of London. Series A, Containing Papers of a Mathematical or Physical Character, 199, 1

Kannan R., Vogelsberger M., Stinson G. S., Hennawi J. F., Marinacci F., Springel V., Macciò A. V., 2016, MNRAS, 458, 2516

Lüders S., 2018

Mo H., van den Bosch F., White S., 2010, Galaxy Formation and Evolution, by Houjun Mo , Frank van den Bosch , Simon White, Cambridge, UK: Cambridge University Press, 2010

Möbius A., 1827, Der barycentrische Calcul. J.A. Barth

Obreja A., Macciò A. V., Moster B., Udrescu S. M., Buck T., Kannan R., Dutton A. A., Blank M., 2019, Monthly Notices of the Royal Astronomical Society, 490, 1518–1538

Omohundro S. M., 1989, Technical report, Five Balltree Construction Algorithms

Osterbrock D. E., Ferland G. J., 2006, Astrophysics of gaseous nebulae and active galactic nuclei

Planck Collaboration et al., 2020, A&A, 641, A6

Ploeckinger S., Schaye J., 2020, Monthly Notices of the Royal Astronomical Society, 497, 4857–4883

Press W. H., Teukolsky S. A., Vetterling W. T., Flannery B. P., 2007, Numerical Recipes 3rd Edition: The Art of Scientific Computing, 3 edn. Cambridge University Press

Richings A. J., Schaye J., Oppenheimer B. D., 2014a, MNRAS, 440, 3349

Richings A. J., Schaye J., Oppenheimer B. D., 2014b, MNRAS, 442, 2780

Seidel R., 1995, Computational Geometry, 5, 115

Shen S., Wadsley J., Stinson G., 2010, MNRAS, 407, 1581

Shewchuk J. R., 2002, Computational Geometry, 22, 21

Springel V., 2005, Monthly Notices of the Royal Astronomical Society, 364, 1105

Sutherland R. S., Dopita M. A., 1993, ApJS, 88, 253

Tielens A. G. G. M., 2010, The Physics and Chemistry of the Interstellar Medium

Wadsley J. W., Keller B. W., Quinn T. R., 2017, MNRAS, 471, 2357

Welzl E., Su P., Drysdale R., 1997, Comput. Geom., 7, 361

Wiersma R. P. C., Schaye J., Smith B. D., 2009, MNRAS, 393, 99

Selbständigkeitserklärung

Hiermit erkläre ich, die vorliegende Arbeit selbständig verfasst zu haben und keine anderen als die in der Arbeit angegebenen Quellen und Hilfsmittel benutzt zu haben.

Stefan Lüders
München, 2021-04-26