




Master in High Performance Computing

# Renewal and Optimization of `ScamPy`: a Python API for painting galaxies on top of dark matter simulations

*Supervisors*:
Andrea Lapi,
Matteo Viel,
Alberto Sartori

*Candidate*:
Tommaso Ronconi

5th edition
2018–2019

# Contents

# Chapter 1

# Introduction

One of the communities that is most actively driving and demanding research and development of both software and hardware is composed by scientists working in Academia. Datasets coming from observations and experiments are constantly growing in number and size. At the same time, the complexity of the scientific models to be computed is growing too, leading to a consequent increase of their computational cost.

Yet, all of this is not accompanied by a growth in awareness of the high performance computing (HPC) methods and best-practices. Certainly courses dedicated to HPC in scientific degree curricula are rare, but this issue is not only due to a lack of education in the community. Very often, it is the way researchers are forced to work that discourages spending more than the strictly necessary effort for the development of scientific software.

In the *publish-or-perish* environment that Academia offers to its researchers, survival is subdue to the submission of results. And the production of good software is still rarely considered a result worth the submission. As a consequence, scientists are often unmotivated in taking care of their implementations. When the aim is to produce results worth to be published, it is acceptable to have an application that works only for the specific purpose it has to accomplish and that runs only with the specific setup of the machine it has been developed on. Since both the accessibility and the portability of the codes are limited, this unavoidably leads to the continuous development of applications serving the very same purpose. Furthermore, scientific codes are often not designed for extracting the best from the performances offered by the computational facilities they are running in.

In this work we have addressed what is so often overlooked in a scientific effort with similar objectives. We have performed a HPC-driven development of the instrument we were needing to get scientifically meaningful results.

Part of the implementation was already in place when we started, developed without HPC-awareness by the same authors of this work. Our intent was not only to boost the performances of our code, but also to make it more accessible, to ease cross platform installation and to generally set-up a flexible tool. Since the API we present has been designed to be easily extensible, in the future we will also be able to evolve our current research towards novel directions. Furthermore, this effort would hopefully also encourage

new users to adopt our tool.

Under this perspective, HPC methods and advanced programming techniques become an enabling technology for doing science. As much as experiments are accurately designed to have the longest life-span possible, we have taken care of designing our software for a long term use.

In this manuscript we describe in detail the several aspects of our API, starting, in the remaining part of this Chapter, from the definition of the scientific problem and the computational bottlenecks that motivated this effort. In Chapter 2 we describe the structure of the library along with the algorithm that it implements. We will give insights on the most critical sections of the computation by providing a detailed description of their C++ implementation in Section 2.2. In Chapter 3 we provide the complete list and description of all the improvements we added to the original implementation. We have validated the final result under both the scientific and the performances perspective, we show this evaluations in Chapter 4. Finally, in Chapter 5, we summarize our main results and we outline the plan for the future applications of the instrument presented in this work.

## 1.1 Scientific problem

Cosmological N-body simulations are a fundamental tool for assessing the non-linear evolution of the large scale structure (LSS). With the increasing power of computational facilities, cosmological N-body simulations have grown in size and resolution, allowing to study extensively the formation and evolution of dark matter (DM) haloes. Our confidence on the reliability of these simulations stands on the argument that the evolution of the non-collisional matter component only depends on the effect of gravity and on the initial conditions. While for the first we can rely on a strong theoretical background, with analytical solutions for both the classical gravitation theory and for a wide range of its modifications, for the latter we have measures with high accuracy [1] of the primordial power spectrum of density fluctuations.

The formation and evolution of the luminous component (i.e. galaxies and intergalactic baryonic matter) though, is far from being as much well understood. Several possible approaches have been tried so far to asses this modeling issue which can be divided into two main categories. On one side, *ab initio* models, such as N-body simulations with a full hydrodynamical treatment and semi-analytical models, are capable of tracing back the evolution in time of galaxies within their DM host haloes. On the other side, *empirical* (or *phenomenological*) models are designed to reproduce observable properties of a target population of objects in a given moment of their evolution. These latter class of methods is typically cheaper in terms of computational power and time required for running.

The former class of approaches is tuned to reproduce the LSS of the Universe at present time, and therefore its reliability at the highest redshifts tends to decrease. On the other hand empirical models are by design particularly suitable for addressing the modelling of the high redshift Universe, but they rely on the availability of high redshift

observations of the population to be modelled.

Our motivation for the original development of the API we present in this work was, indeed, to study a particular period in the high redshift Universe. Specifically, our aim was to model the distribution of the first sources that started to shed light on the neutral medium in the early Universe, triggering the process called *Reionization*.

The history of the Universe is a history of cooling: when electrons and protons start to form the energy of the medium is too high and radiation is coupled with matter. This prevents negatively and positively charged particles to combine and form atoms but, as the medium cools down while the Universe volume increases, radiation and matter decouple. This happens at around redshift $z \approx 1100$ when temperatures reach approximately 3000 K. From this moment on, the mean free path of photons becomes infinite and recombination makes the Universe completely neutral.

The period that follows is called the Dark Ages. Since today we observe the Universe being completely ionized, we know that the Dark Ages have an ending. From the observation of the absorbed spectra of distant quasars [2, 3] and as results from the Cosmic Microwave Background measure of the optical depth of Thomson scattering at reionization [1], we also know that by redshift $z = 6$ reionization of hydrogen has been completed.

Nevertheless, our knowledge of the reionization history and of the main players involved is still uncertain. The redshift at which hydrogen is though to start reionizing is included in a wide window between $z \approx 30$ and $z \approx 15$. But, most importantly, we do not know much of how it happened. It is not sure whether the process started from the lower density regions then expanded towards the higher density ones or the other way round. We do not have access yet to a tomographic picture of neutral and ionized matter at that epoch and, even when these measures will be available by the observation of 21 cm intensity mapping, we will need physical models to interpret them.

Most of these uncertainties are the consequence of the limited knowledge we have about the sources producing the ionizing radiation needed to complete reionization. Our aim is to study the history of reionization, focusing on the role played by the observed sources of high energy radiation. Active Galactic Nuclei (AGN), in spite of the high flux of ionizing photons they produce, might have played a secondary role. The most accepted hypothesis is that the highly star forming galaxies, which were present in the early Universe, have been the main responsible for the production of ionizing photons.

Most of the studies in this sense have been focused on exploiting the statistical properties of observed populations. To perform a systematic three-dimensional study of the structure and evolution of the ionized bubbles developing around said sources, it is inevitable to use simulations.

The most suitable approach then would be to use empirical modeling, for its capability of reproducing the observed statistics of a target population by construction at any redshift. This requires to define the hosted-object/hosting-halo connection for associating to the underlying DM distribution its baryonic counterpart. In the next Section we will describe the Sub-halo Clustering and Abundance Matching technique, an extension of the classical Halo Occupation Distribution (HOD) method. This class of methods is widely

used in the community but, in spite of their popularity, at the best of our knowledge there is no reference library available specifically developed for this task.

## 1.2 Sub-halo Clustering & Abundance Matching

Our approach for the definition of the hosted-object/hosting-halo connection is based on the Sub-halo Clustering and Abundance Matching (SCAM) technique [4]. With the standard HOD approach, hosted objects are associated to each halo employing a prescription which is based on the total halo mass, or on some other proxy of this information (e.g. halo peak velocity, velocity dispersion). On the other side, Sub-halo abundance matching (SHAM) assumes a monotonic relation between some observed object property (e.g. luminosity or stellar mass of a galaxies) and a given halo property (e.g. halo mass). While the first approach is capable of, and extensively used for, reproducing the spatial distribution properties of some target population, the second is the standard for providing plain DM haloes and sub-haloes with observational properties that would otherwise require a full-hydrodynamical treatment of the simulation from which these are extracted.

The SCAM prescription aims to assemble both approaches, providing a parameterised model to fit both some observable abundance and the clustering properties of the target population. The approach is nothing more than applying HOD and SHAM in sequence:

1. the occupation functions for central, $N_{\rm cen}(M_h)$, and satellite galaxies, $N_{\rm sat}(M_h)$, depend on a set of defining parameters which can vary in number depending on the shape used. These functions depend on a proxy of the total mass of the host halo. We sample the space of the defining parameters using a Markov-Chain Monte-Carlo (MCMC) to maximize a likelihood built as the sum of the $\chi^2$ of the two measures we want to fit, namely the two-point angular correlation function at a given redshift, $\omega(\theta, z)$, and the average number of sources at a given redshift, $n_g(z)$:

$$\log \mathcal{L} \equiv -\frac{1}{2}\left(\chi^2_{\omega(\theta,z)} + \chi^2_{n_g(z)}\right) , \tag{1.1}$$

   The analytic form of both $\omega(\theta, z)$ and $n_g(z)$, depending on the same occupation functions $N_{\rm cen}(M_h)$ and $N_{\rm sat}(M_h)$, can be obtained with the standard halo model [see e.g. 5], which we describe in detail in Section 1.2.1. How these occupation functions are used to select which sub-haloes will host our mock objects is reported in Section 2.1.1.

2. Once we get the host halo/subhalo hierarchy with the abundance and clustering properties we want, as guaranteed by Eq. (1.1), we can apply our SHAM algorithm to link each mass (or, equivalently, mass-proxy) bin with the corresponding luminosity (or observable property) bin.

When these two steps have been performed, the mock-catalogue is built.

### 1.2.1 The halo model

It provides a halo-based description of nonlinear gravitational clustering and is used at both low and high redshift [6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17]. The key assumption of this model is that the number of galaxies, $N_g$, in a given dark matter halo depends only on the halo mass, $M_h$. Specifically, if we assume that $N_g(M_h)$ follows a Poisson distribution with mean proportional to the mass of the halo $M_h$, we can write

$$\langle N_g \rangle (M_h) \propto M_h \tag{1.2}$$

$$\langle N_g (N_g - 1) \rangle (M_h) \propto M_h^2 \tag{1.3}$$

From these assumptions it is possible to derive correlations of any order as a sum of the contributions of each possible combination of objects identified in single or in multiple haloes. To get the models required by Eq. (1.1) we only need the 1-point and the 2-point statistics. We derive the first as the mean abundance of objects at a given redshift. The average mass density in haloes at redshift $z$ is given by

$$\overline{\rho}(z) = \int \mathrm{d}M_h \, M_h \, n(M_h, z) \tag{1.4}$$

where $n(M_h, z)$ is the halo mass function. With Eq. (1.2) we can then define the average number of objects at redshift $z$, hosted in haloes with mass $M_{\rm min} \leq M_h \leq M_{\rm max}$, as

$$n_g(z) \equiv \int_{M_{\rm min}}^{M_{\rm max}} \langle N_g \rangle (M_h) n(M_h, z) \, \mathrm{d}M_h \ . \tag{1.5}$$

The 2-point correlation function, $\xi(r, z)$ derivation would require to treat with convolutions, therefore we prefer to obtain it by inverse Fourier-transforming the non-linear power spectrum $P(k, z)$, whose derivation can instead be treated with simple multiplications:

$$\xi(r, z) = \frac{1}{2\pi^2} \int_{k_{\rm min}}^{k_{\rm max}} \mathrm{d}k k^2 P(k, z) \frac{\sin(kr)}{kr} \ . \tag{1.6}$$

$P(k, z)$ can be expressed as the sum of the contribution of two terms:

$$P(k, z) = P_{\rm 1h}(k, z) + P_{\rm 2h}(k, z) \ , \tag{1.7}$$

where the first, dubbed *1-halo term*, results from the correlation among objects belonging to the same halo, while the second, dubbed *2-halo term*, gives the correlation between objects belonging to two different haloes.

The 1-halo term in real space is the convolution of two similar profiles of shape $\widetilde{u}(k, z|M_h)$

$$\begin{aligned} \widetilde{u}(k, z|M_h) = \frac{4\pi \rho_s r_s^3}{M_h} \Bigg\{ & \sin(kr_s) \big[ \mathrm{Si}((1+c)kr_s) - \mathrm{Si}(kr_s) \big] - \frac{\sin(ckr_s)}{(1+c)kr_s} \\ & - \cos(kr_s) \big[ \mathrm{Ci}((1+c)kr_s) - \mathrm{Ci}(kr_s) \big] \Bigg\} \ , \end{aligned} \tag{1.8}$$

where $c$ is the halo concentration, $\rho_s$ and $r_s$ are the scale density and radius, respectively, of the NFW profile and the sine and cosine integrals are

$$\mathrm{Ci}(x) = . \int_t^\infty \frac{\cos t}{t}\,\mathrm{d}t \text{ and } \mathrm{Si}(x) = \int_0^x \frac{\sin t}{t}\,\mathrm{d}t\,. \tag{1.9}$$

Eq. (1.8) provides the Fourier transform of the dark matter distribution within a halo of mass $M_h$ at redshift $z$. By weighting this profile by the total number density of pairs, $n(M_h)(M_h/\overline{\rho})^2$, contributed by haloes of mass we obtain:

$$\begin{aligned} P_{1\mathrm{h}}(k,z) &\equiv \int \mathrm{d}M_h\, n(M_h,z) \left(\frac{M_h}{\overline{\rho}}\right)^2 \left|\widetilde{u}(k,z|M_h)\right|^2 = \\ &= \frac{1}{n_g^2(z)} \int_{M_{\min}}^{M_{\max}} \mathrm{d}M_h \langle N_{\mathrm{g}}(N_{\mathrm{g}}-1)\rangle(M_h) n(M_h,z) \left|\widetilde{u}(k,z|M_h)\right|^2, \end{aligned} \tag{1.10}$$

with $n(M_h,z)$ the halo mass function for host haloes of mass $M_h$ at redshift $z$ and where in the second equivalence we used Eqs. (1.3) and (1.5) to substitute the ratio $\left(M_h/\overline{\rho}\right)^2$.

The 2-halo term is more complicated and for a complete discussion the reader should refer to [5]. Let us just say that, for most of the applications, it is enough to express the power spectrum in its linear form, corrections to this approximation are mostly affecting the small-scales which are almost entirely dominated by the 1-halo component. This is mostly because the 2-halo term depends on the biasing factor which on large scales is deterministic. We therefore have that, in real-space, the power coming from correlations between objects belonging to two separate haloes is expressed by the product between the convolution of two terms and the biased linear correlation function (i.e. $b'_h(M'_h)b_h(M''_h)\xi_{\mathrm{lin}}(r,z)$). The two terms in the convolution provide the product between the Fourier-space density profile $\widetilde{u}(k,z|M_h)$, weighted by the total number density of objects within that particular halo (i.e. $n(M_h)(M_h/\overline{\rho})$). In Fourier space, we therefore have

$$\begin{aligned} P_{2\mathrm{h}}(k,z) \equiv &\int \mathrm{d}M'_h\, n(M'_h)\, \frac{M'_h}{\overline{\rho}}\, \widetilde{u}(k,z|M'_h)\, b(M'_h) \\ &\int \mathrm{d}M''_h\, n(M''_h)\, \frac{M''_h}{\overline{\rho}}\, \widetilde{u}(k,z|M''_h)\, b(M''_h)\, P_{\mathrm{lin}}(k,z) = \\ =&\frac{P_{\mathrm{lin}}(k,z)}{n_g^2(z)} \left[\int_{M_{\min}}^{M_{\max}} \mathrm{d}M_h\, \langle N_{\mathrm{g}}\rangle(M_h)\, n(M_h)\, b(M_h,z)\, \widetilde{u}_h(k,z|M_h)\right]^2 \end{aligned} \tag{1.11}$$

with $b(M_h)$ the halo bias and $P_{\mathrm{lin}}(k,z)$ the linear matter power spectrum evolved up to redshift $z$. For going from the first to the second equivalence we have to make two assumptions. First we assume self-similarity between haloes. This means that the two nested integrals in $\mathrm{d}M'_h$ and $\mathrm{d}M''_h$ are equivalent to the square of the integral in the rightmost expression. Secondly, we make use of Eqs. (1.2) and (1.5) to substitute the ratio $M_h/\overline{\rho}$.

The average number of galaxies within a single halo can be decomposed into the contributions of the probability to have a central galaxy, $N_{\mathrm{cen}}(M_h)$ and of the average

number of satellite galaxies, $N_{\rm sat}(M_h)$,

$$\langle N_g \rangle (M_h) \equiv N_{\rm cen}(M_h) + N_{\rm sat}(M_h) \tag{1.12}$$

which are precisely the occupation functions we already mentioned in Section 1.2. There is no physically motivated shape for these two functions, which tipically have parameterised shapes. By tuning this parameterisation we can obtain the prescription for defining the hosted-object/hosting-halo connection.

Eq. (1.3) can also be approximated to

$$\begin{aligned} \langle N_{\rm g}(N_{\rm g}-1)\rangle(M_h) &\approx \langle N_{\rm cen} N_{\rm sat}\rangle(M_h) + 2\,\langle N_{\rm sat}(N_{\rm sat}-1)\rangle(M_h) \approx \\ &\approx N_{\rm cen}(M_h)\, N_{\rm sat}(M_h) + N^2_{\rm sat}(M_h) \end{aligned} \tag{1.13}$$

Thus we can further decompose the 1-halo term of the power spectrum as the combination of power given by *central-satellite* couples (cs) and *satellite-satellite* couples (ss):

$$P_{1h}(k,z) \approx P_{cs}(k,z) + P_{ss}(k,z)\ , \tag{1.14}$$

When dealing with observations, it is often more useful to derive an expression to get the projected correlation function, $\omega(r_p, z)$, where $r_p$ is the projected distance between two objects, assuming flat-sky. From the Limber approximation [18] we have

$$\begin{aligned} \omega(r_p,z) &= \mathcal{A}\big[\xi(r,z)\big] = \mathcal{A}\big\{\mathcal{F}\big[P(k,z)\big]\big\} = \mathcal{H}_0\big[P(k,z)\big] = \\ &= \frac{1}{2\pi}\int \mathrm{d}k\, k\, P(k,z)\, J_0(r_p k) \end{aligned} \tag{1.15}$$

where $J_0(x)$ is the $0^{\rm th}$-order Bessel function of the first kind. Reading the expression above from left to right, we can get the projected correlation function by Abel-projecting the 3D correlation function $\xi(r,z)$. The definition in Eq. (1.6), therefore is obtained by Abel-transforming the Fourier transform of the power spectrum. This is equivalent to perform the zeroth-order Hankel transform of the power spectrum, which leads to the last equivalance in Eq. (1.15).

Eq. (1.15) though, is valid as long as we are able to measure distances directly in an infinitesimal redshift bin, which is not realistic for real data. Our projected distance depends on the angular separation, $\theta$, and the cosmological distance, $d_C(z)$, of the observed object

$$r_p(\theta, z) = \theta \cdot d_C(z)\ . \tag{1.16}$$

By projecting the objects in our lightcone on a flat surface at the target redshift, we are summing up the contribution of all the objects along the line of sight. Therefore the two-point angular correlation function can be expressed as

$$\omega(\theta, z) = \int \mathrm{d}z \frac{\mathrm{d}V(z)}{\mathrm{d}z} \mathcal{N}^2(z)\, \omega[r_p(\theta,z), z] \tag{1.17}$$

where $\frac{\mathrm{d}V(z)}{\mathrm{d}z}$ is the comoving volume unit and $\mathscr{N}(z)$ is the normalized redshift distribution of the target population. If we assume that, in the redshift interval $[z_1, z_2]$, $\omega(\theta, z)$ is approximately constant, we can then write

$$\omega(\theta, z) \approx \left[\int_{z_1}^{z_2} \mathrm{d}z \frac{\mathrm{d}V(z)}{\mathrm{d}z} \mathscr{N}^2(z)\right] \cdot \omega[r_p(\theta), \overline{z})] \tag{1.18}$$

where $\overline{z}$ is the mean redshift of the objects.

## 1.3 HPC Motivation

As we were starting with this project, a first basic implementation of the main algorithm was already in-place. We will describe the details of the algorithm in Chapter 2, as for now let us just say that we had a rudimentary C++ version of the routines computing the halo-model functions (Section 1.2.1). Even though we were able to obtain measures of the angular two-point correlation function, $\omega(\theta, z)$ and of the average number of objects, $n_g(z)$, at given redshift, we encountered a bottle-neck in our development.

To sample the parameter space using a MCMC sampler, the halo model estimates have to be computed tens of thousands of times. Unfortunately, with our old halo model implementation, every single estimate was taking tens of seconds to compute, even though we were already exploiting multi-threading of for loops, fostered by OpenMP [19], where this was possible. This would have meant more then a day of computation for a single chain of our sampler to run. At this stage of the development, for each walker of the MCMC sampler the computation of a single step of the chain used to take times of the order of ten seconds. By running a walker per socket of a NUMA machine we were able to have two walkers running at a time. To make matters worse, with the external sampler we were using, it was not possible to parallelize the chains on distributed memory. Since it is recommended to have a number of walkers at least twice the number of parameters of the model, and given that in the best case scenario we have to sample a four-dimensional parameter space, it was clear that a serious optimization of the application was necessary.[1]

To understand how to optimize our implementation, we started by obtaining a basic profiling of the main executable that was computing the halo model estimates. For this purpose we used `perf`, a performance analyzing tool, available from Linux kernel version 2.6.31, which has access to the internal counters present in the hardware of any machine. In Fig. 1.1 we show the output of an event-based profiling in which we have measured the instructions per cycle and the percentage of branch-misses. Roughly, in a well performing application, the number of instructions per cycle should be in the range $1 \div 4$, while the number of branch-misses should be the smallest possible. Given the reasonably good

[1] E.g. if we consider 24 hours of computation per walker, 2 walkers running on parallel per run for a minimum of 8 walkers in total, this would have meant 4 days on the Ulysses cluster. Since the queue that allows to reserve the longest time on Ulysses provides 48 hours of computation, this also means having to implement a framework that allows for check-points in the computation and for the production of restart files, thus complicating substantially the code.

```
Performance counter stats for './2pt_angular.x param/param_z1.00.ini':

  465,937,743,125      cycles:u
  577,483,658,544      instructions:u            #    1.24  insn per cycle
   86,200,466,569      branches:u
      388,496,041      branch-misses:u           #    0.45% of all branches

     24.768249779 seconds time elapsed
```

Figure 1.1: Output of a simple event based profiling using `perf` in which we have measured the outputs of the instruction counter and of the branch counters. Time of computation is also shown.

values we measured, one could say that the implementation was performing discreetly well. Yet, the time of execution was unacceptably large.

We therefore performed a sample-based profiling to get in dept on the function calls that mostly affect computation. In Fig. 1.2 we show the complete call-tree of the executable. Each block represents a called function and provides the percentage of the whole calls to lower level functions with respect to the total; the percentage of the total execution time spent in each block is also provided. Yellow/red blocks are the ones that consume the most of the resources.

In Fig. 1.3 we show two close-up views of the call-tree. On the left panel we have spotted the function that performs the largest number of calls to lower level functions. It is an integration routine imported from the `gsl_integration` external library [part of the GNU Scientific Library, 20]. If we look at the analytical form of a typical function derived by the halo model (see e.g. Eq. (1.5) and Eq. (1.17)) integrals are unavoidable.

The right panel of Fig. 1.3 though, gives us another crucial information. Almost all the calls to the two functions `pow()` and `exp()` that appear to consume the vast majority of the resources, are coming from the integration routine. We can therefore conclude that the integration is not efficient because the way the integrands are computed is not.

If we inspect the analytical expressions of Section 1.2.1 we can see a huge number of ratios and calls to complex cosmological functions such as the mass-function of DM haloes or the density profile of haloes in Fourier Space (FS). Both these two functions involve power laws, exponentials, ratios and, in the case of the density profile in FS, also the sine and cosine integrals.

Prompted by these results, we outlined a radical optimisation strategy for our application:

- Rebuild the code with advanced programming techniques & HPC best practices.
  - Implement the core functions avoiding heavy to compute function-calls (minimize the number of operations and calls to computationally expensive functions, such as `pow()`).
  - Development of a manageable tool for interpolation.
  - Employ multi-threading where needed.

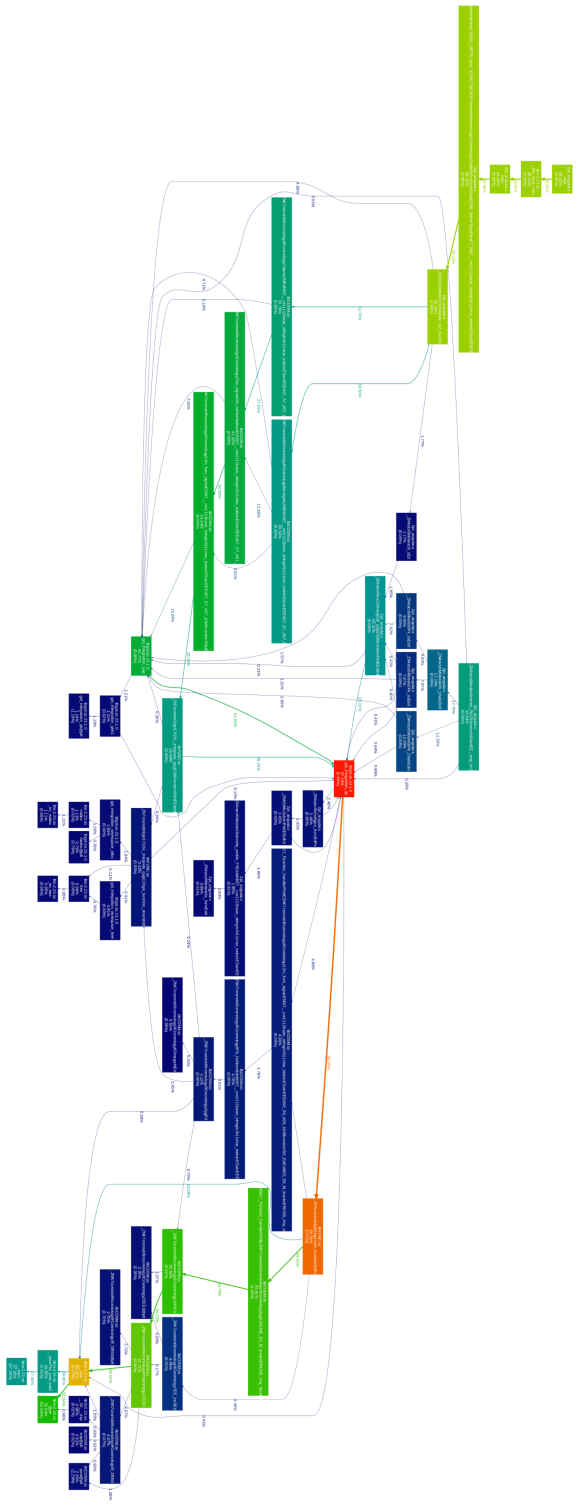

Figure 1.2: Call-tree of the executable performing the most computationally demanding operations needed by our first version of the application.

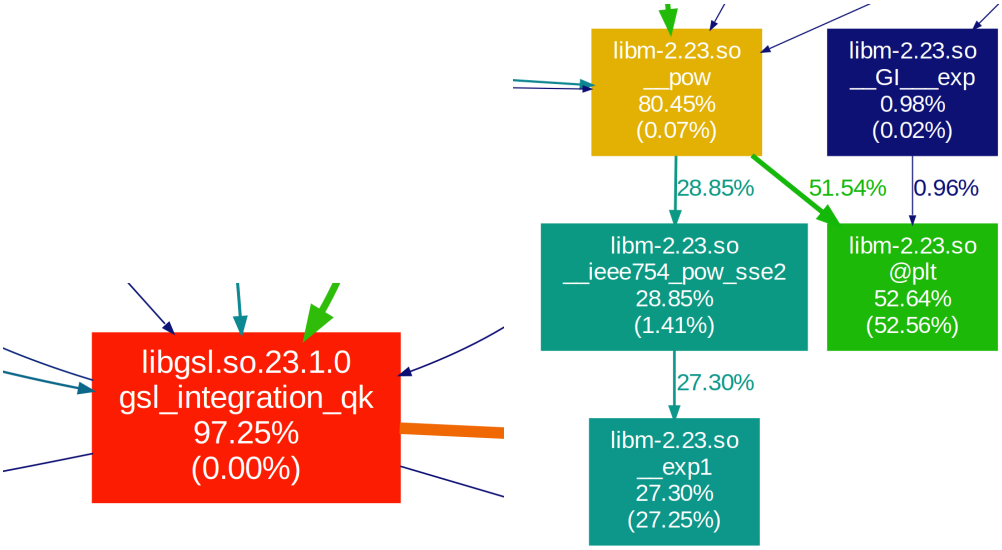


Figure 1.3: Close up view of the function performing the largest number of calls (*left panel*) and of the functions in which the application spends the most of its time (*right panel*).

- Modularization and testing.
  - By keeping the implementation design strictly object-oriented we favour the modularization of its components; the collection of all the modular components should be organised in a Git repository. This allows for a rational development of the required feature while keeping track of the modifications.
  - Each unitary component of the source code should be tested at its limits; the development of an integrated testing environment for this purpose is desirable.
  - In order to guarantee consistency during code extension, the implementation updates uploaded to the remote GitHub repository should be checked with a Continuous Integration platform.
  - Integration of a build system to detach from native compilation and, therefore, to boost portability and simplify cross-compiler testing is also desirable.
- Detachment from `CosmoBolognaLib` to allow for more flexibility and extensibility while unburdening the build.
- Implementation of a python user interface to both simplify the access to the API functionalities and to allow usage with the support of popular novel libraries.
- Integration of an external sampler for MCMC parameter inference.

# Chapter 2

# The ScamPy library

In this section we introduce ScamPy, our highly-optimized and flexible library for "painting" an observed population on top of the DM-halo/subhalo hierarchy obtained from DM-only N-body simulations. We will give here a general overview of the algorithm on which our API is based. We will then describe the key aspects of our C++ implementation, pointing out how it is intended for future expansion and further optimization. A detailed description of all the components of the API itself will be presented in Chapter 3. The full documentation of ScamPy, with a set of examples and tutorials, will be available with the publication of the scientific paper extracted from this work.

## 2.1 Algorithm overview

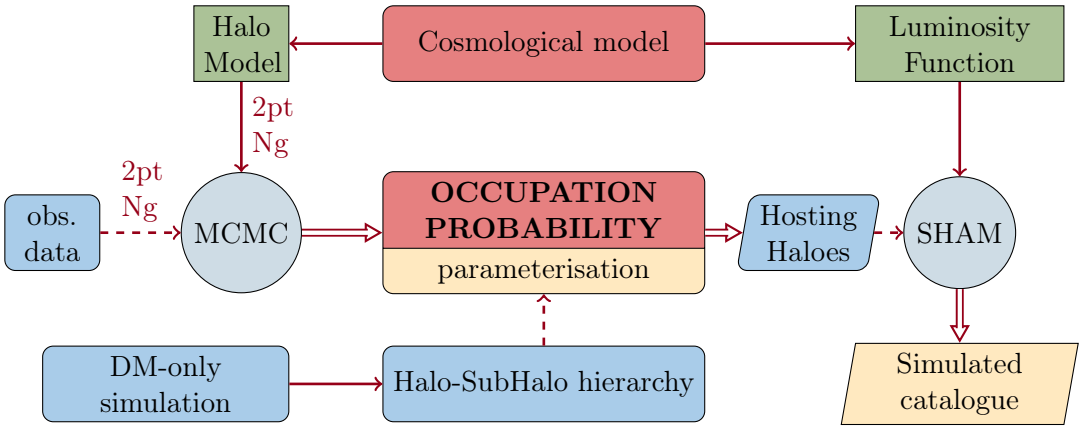


Figure 2.1: Flowchart describing the main components of the algorithm. In red the two main kernel modules. Green rectangles dub models from which the workflow depends. Round gray circles are for engines that operate on some inputs. Cyan is for inputs, yellow and parallelograms for outputs.

In Figure 2.1 we give a schematic of the main components of the ScamPy API.

All the framework is centred around the occupation probabilities, namely $N_{\rm cen}(M_h)$ and $N_{\rm sat}(M_h)$, which define the average numbers of central and satellite galaxies hosted within the sub-halo hierarchy. Several parameterisations of these two functional forms exist. One of the most widely used is the standard 5-parameters HOD model, with the probability of having a central given by an activation function and the number distribution of satellites given by a power-law:

$$N_{\rm cen}(M_h) = \frac{1}{2}\left[1 + {\rm erf}\left(\frac{\log M - \log M_{\rm min}}{\sigma_{\log M_h}}\right)\right] \tag{2.1}$$

$$N_{\rm sat}(M_h) = \left(\frac{M_h - M_{\rm cut}}{M_1}\right)^{\alpha_{\rm sat}} \tag{2.2}$$

where $M_{\rm min}$ is the characteristic minimum mass of halos that host central galaxies, $\sigma_{\log M_h}$ is the width of this transition, $M_{\rm cut}$ is the characteristic cut-off scale for hosting satellites, $M_1$ is a normalization factor, and $\alpha_{\rm sat}$ is the power-law slope. Our API provides users with both an implementation of the Eqs. (2.1) and (2.2), and the possibility to use their own parameterisation by inheriting from a base `occupation_p` class.[1] Given that both the modeling of the observable statistics (Section 1.2.1) and the HOD method used for populating DM haloes depend on these functions, we implemented an object that can be shared by both these sections of the API. As shown in Fig. 2.1, the parameters of the occupation probabilities can be tuned by running an MCMC sampling. By using a likelihood as the one exposed in Section 1.2, the halo-model parameterisation that best fits the observed 1- and 2-point statistics of a target population can be inferred.[2]

The chosen cosmological model acts on top of our working pipeline. Besides providing the user with a set of cosmographic functions for modifying and analysing results on the fly, it plays two major roles in the API. On the one hand it defines the cosmological functions that are used by the halo model, such as the halo-mass function or the DM density profile in Fourier space. On the other hand it provides a set of luminosity functions that the user can associate to the populated catalogue through the SHAM procedure. This approach is not the only one possible, as users are free to define their own observable property distribution and provide it to the function that is responsible for applying the abundance matching algorithm.

When the HOD parameterisation and the observable-property distribution have been set, it is possible to populate the halo/sub-halo hierarchy of a DM-only catalogue. In Alg. 1 we outline the steps required to populate a halo catalogue with mock observables. We start from a halo/subhalo hierarchy obtained by means of some algorithm (e.g. SUBFIND) that have been run on top of a DM-only simulation. This is loaded into a `catalogue` structure that manages the hierarchy dividing the haloes in *central* and *satellite* subhaloes.[3]

[1] The documentation of the library will comprehend a tutorial on how to achieve this.

[2] In the documentation website we will provide a step-by-step tutorial using `Emcee` [21].

[3] For the case of SUBFIND run on top of a GADGET snapshot this can be done automatically using the `catalogue.read_from_gadget()` function. We plan to add similar functions for different halo-finders (e.g. ROCKSTAR and SPARTA) in future extensions of the library.

**Algorithm 1** Schematic outline of the steps required to obtain a mock galaxy catalogue with ScamPy.

```
// Load Halo/Subhalo hierarchy
// (e.g.  from SUBFIND algorithm)
halo_cat = catalogue(  chosen from file  )

// Choose occupation probability function
OPF = OPF( HOD parameters )

// Populate haloes
gxy_array = halo_cat.populate( model = OPF )

// Associate luminosities
gxy_array = SHAM( gxy_array, SHAM parameters )
```

Our `catalogue` class comes with a `populate()` member function that takes an object of type occupation probability as argument and returns a trimmed version of the original catalogue in which only the central and satellite haloes hosting an object of the target population are left. We give a detailed description of this algorithm in Section 2.1.1. When this catalogue is ready the SHAM algorithm can be run on top of it to associate at each mass a mock-observable property. Cumulative distributions are monotonic by construction, therefore it is quite easy to define a bijective relation between the cumulative mass distribution of haloes and the cumulative observable property distribution of the target population. This algorithm is described in Section 2.1.2.

### 2.1.1 Populating algorithm

Input subhalo catalogues are trimmed into hosting subhalo catalogues by passing to the `populate()` member function of the class `catalogue` an object of type `occupation_p`.

In Algorithm 2 we describe this halo occupation routine. For each halo $i$ in the catalogue, we compute the values of $\langle N_{\rm cen}\rangle(M_i)$ and $\langle N_{\rm sat}\rangle(M_i)$. To account for the assumptions made in our derivation of the halo model, we select the number of objects each halo will host by extracting a random number from a Poisson distribution. For the occupation of the central halo this reduces to extracting a random variable from a Binomial distribution: $N_{\rm cen} = \mathscr{B}(1, \langle N_{\rm cen}\rangle_i)$. While, in the case of satellite subhaloes, we extract a random Poisson variable $N_{\rm sat} = \mathscr{P}(\langle N_{\rm sat}\rangle_i)$, then we randomly select $N_{\rm sat}$ satellite subhaloes from those residing in the $i^{\rm th}$ halo.

In Fig. 2.2 we show a 4 Mpc/$h$ thick slice of a simulation with box side lenght of 64 Mpc/$h$, the background colour code represents the density field traced by all the subhaloes found by the SUBFIND algorithm, smoothed with a Gaussian filter, while the markers show the positions of the subhaloes selected by the populating algorithm. We will show in Section 4.1 that this distribution of objects reproduces the observed statistics. It is possible to notice how the markers trace the spatial distribution of the underlying

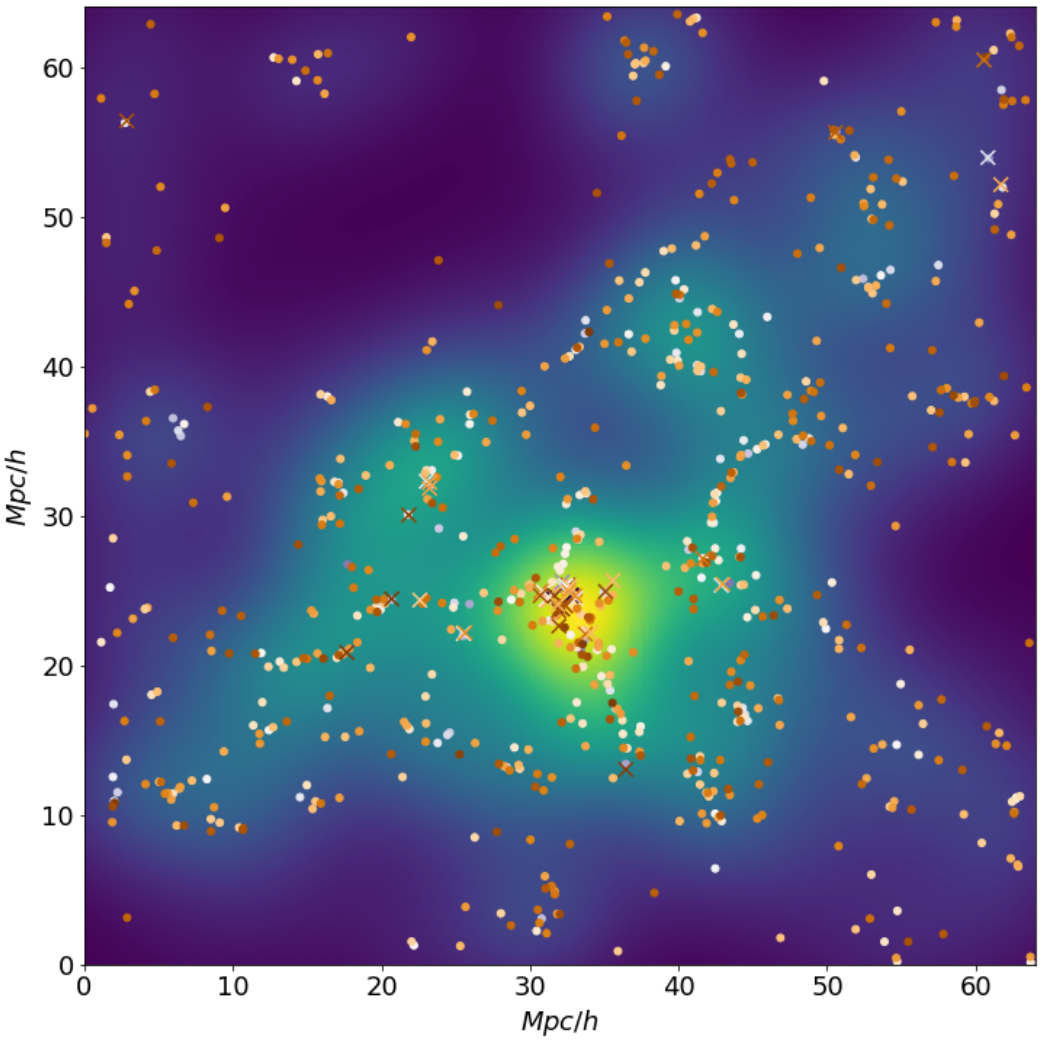


Figure 2.2: A 4 Mpc/$h$ thick slice of a populated catalogue obtained from a DM-only simulation with 64 Mpc/$h$ box side lenght. The colour code on the background shows the smoothed DM density field (with density increasing going from darker to brighter regions) while the markers show our mock galaxies (with color representing lower to higher luminosity going from brighter to darker). Circles are for centrals and crosses for satellites.

**Algorithm 2** Description of the `populate` (model = OPF) function. This is an implementation of the HOD prescription, where the assumptions made to define the halo model (i.e. the average number of objects within a halo follows a Poisson distribution with mean $\langle N_g \rangle(M_h)$) are accounted for.

```
// Iterate over all the haloes in catalogue
for halo in catalogue do

    // Compute probability of central
    p_cen ← model.N_cen( halo.mass )

    // Define a binomial random variable
    select ← random.Binomial(1, p_cen)
    if select then
        halo ← central

    // Compute average number of satellites
    N̄_sat ← model.N_sat( halo.mass )

    // Define a Poisson random variable
    N_sat = random.Poisson( N̄_sat )
    halo ← select randomly N_sat objects among satellites
```

DM density field.

### 2.1.2 Abundance matching algorithm

When the host subhaloes have been selected we can run the last step of our algorithm. The `abundance_matching()` function implements the SHAM prescription to associate to each subhalo an observable property (e.g. a luminosity or the star formation rate of a galaxy). This is achieved by defining a bijective relation between the cumulative distribution of subhaloes as a function of their mass and the cumulative distribution of the property we want to associate them.

An example of this procedure is shown in Fig. 2.3. We want to set, for each subhalo, the UV luminosity of the galaxy it hosts. In the left panel we show the cumulative mass-distribution of subhaloes, $\mathrm{d}N(M_{\mathrm{subhalo}})$, with the dashed green region being the mass resolution limit of the DM subhaloes in our simulation after the populating algorithm has been applied. On the right panel we show the cumulative UV luminosity function, which is given by the integral

$$\Phi(M^{\mathrm{UV}} < M^{\mathrm{UV}}_{\mathrm{lim}}) = \int_{-\infty}^{M^{\mathrm{UV}}_{\mathrm{lim}}} \frac{\mathrm{d}\Phi}{\mathrm{d}M^{\mathrm{UV}}}\,\mathrm{d}M^{\mathrm{UV}} \tag{2.3}$$

where $M^{\mathrm{UV}}_{\mathrm{lim}}$ is the limiting magnitude of the survey data we want to reproduce (marked by a dashed red region in Fig 2.3). We find the abundance corresponding to each mass

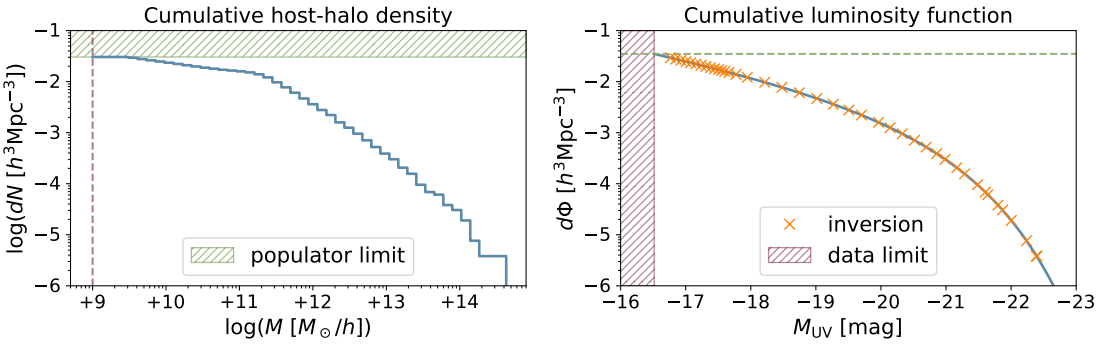


Figure 2.3: Abundance matching scheme. *Left panel:* cumulative number density of host subhaloes divided into regular logarithmic bins (solid blue step-line). The green dashed band shows the limit imposed by the resolution in mass of the simulation, which is inherited by the host catalogue obtained with the populator algorithm. *Right panel:* cumulative luminosity function (solid blue line). The dashed red band shows the limit imposed by the magnitude limit of the observed target population. Orange crosses mark the positions, bin-per-bin, of the abundances measured in each bin of the left panel. In both the left and the right panel we reported with a dashed line of corresponding colour the limit imposed by the resolution of the catalogue (dashed green line in right panel) and by the magnitude limit of the survey (dashed red line in the left panel).

bin (blue step curve in the left panel) and we compute the corresponding luminosity by inverting the cumulative luminosity function obtained with Eq. (2.3):

$$M^{\rm UV}(M_{\rm subhalo}) = \Phi^{-1}[\,{\rm d}N(M_{\rm subhalo})]\ . \tag{2.4}$$

The result of this matching is shown by the orange crosses in the right panel of Fig. 2.3. At the time we are writing, the scatter around the distribution of luminosities can be controlled by tuning the bin-width used to measure ${\rm d}N(M_{\rm subhalo})$. We plan to extend this functionality of the API by adding a parameter for tuning this scatter to the value chosen by the user.

## 2.2 C++ implementation details

"**Polymorphism** — *providing a single interface to entities of different types. Virtual functions provide dynamic (run-time) polymorphism through an interface provided by a base class. Overloaded functions and templates provide static (compile-time) polymorphism.*"

(Bjarne Stroustrup, father of the C++ language)

We give here some details on the C++ implementation of the algorithm described above. We report on the advanced Object-Oriented Programming (OOP) strategies we

adopted to boost performances and allow flexibility of the code. With an extensive use of both static and dynamic polymorphism we reduce code replication to a minimum and keep the code general without using conditional-statements, while preserving flexibility (Sections 2.2.1 and 2.2.2).

Based on these strategies we have also developed a powerful utility module for interpolation (Section 2.2.3) which allowed us to boost dramatically the computation performances while keeping control on the precision of the computation (as we will show in Section 4.3).

### 2.2.1 The `cosmology` class

We have implemented a templated class for cosmological computations. Since this class embeds all the kernel functions that are necessary for the scientific goal of the API, we extensively tested all its output results, both by comparing them with values found in literature and by collecting equivalent measures from another library developed for the same purpose [22].[4]

There are indeed several public implementations of the cosmological functions needed in our work. We have implemented our version, for two main reasons

- performances: all the operations required are implemented in the most computationally efficient way, minimizing the ratios and calls to costly functions. Further accelaration is achieved by using our `interpolator` type.
- for installation simplicity: having our own implementation provides an easy-to-install API which requires the minimum system setup possible for working.

Nevertheless, being aware of the wide range of public libraries serving the same purpose, we decided to keep our cosmology interface open for working with other implementations. The `cosmology` class is indeed derived from a base `cosmo_model` type, which constitutes a C++ interface to our implementation or third-party libraries providing the same functionalities.

In Fig. 2.4 we show the call graph of our default implementation of the `cosmology` class. It inherits (solid blue arrow) from our `cosmo_model` structure which has access (dashed purple arrow) to the `interpolator` class through the private variables `Ez_f`, `P0_f` and `zE_f`. The dashed yellow arrow marks the relation between the template instance `interpolator< gsl_log_interp >` and the template class `interpolator< T >` from which it was instantiated.

### 2.2.2 The `halo_model` class

This class provides the user with the full implementation of all the functions presented in Section 1.2.1. Since it provides methods and functions that have to be called tens of thousands of times, most of the C++ section of the library has been developed in order to make this class work with the best possible performances. Fig. 2.5 shows the

[4] CosmoBolognaLib website

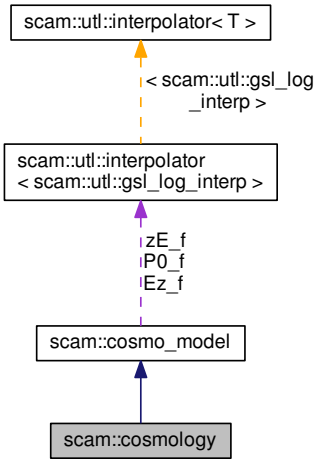


Figure 2.4: Call-graph of the `cosmology` class in our default implementation. The solid blue arrow is used to visualize a public inheritance relation between two classes. A purple dashed arrow is used if a class is contained or used by another class. The arrow is labeled with the variables through which the pointed class or struct is accessible. A yellow dashed arrow denotes a relation between a template instance and the template class it was instantiated from. The arrow is labeled with the template parameters of the instance. *(Obtained with Doxygen)*

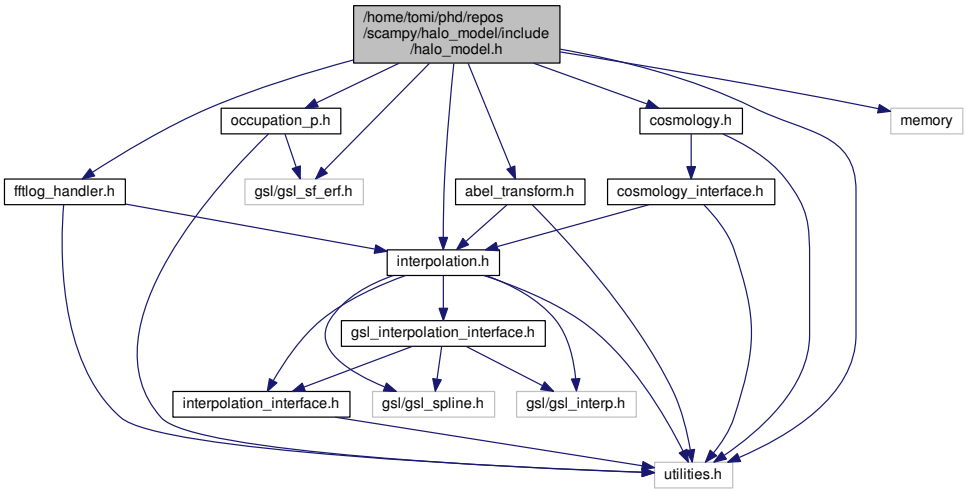


Figure 2.5: Dependency graph of the `halo_model.h` header. The arrows mark calls to internal or external headers. *(Obtained with Doxygen)*

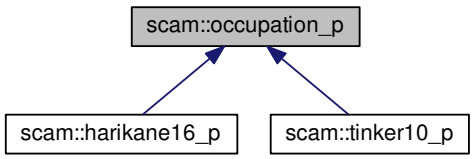


Figure 2.6: Inheritance graph of the classes providing the occupation probability functions which are already defined in the default implementation of ScamPy. *(Obtained with Doxygen)*

dependency tree of the `halo_model` header. It can be noticed that it is directly connected with almost all the other headers provided by our `c++` implementation. Furthermore, the only two external headers included that are not part of the C++ standard are actually required by the `interpolator` class which is defined in the `interpolation.h` header. We describe this core functionality of our API in Section 2.2.3.

The constructor of our `halo_model` class takes two mandatory arguments:

- an object of type `cosmology` which provides the cosmological functions required by the halo model;
- an object of type `occupation_p`, which provides the shape of the occupation probabilities of central, $N_{\rm cen}$, and satellite, $N_{\rm sat}$, objects.

The latter is again a polymorphic custom type, an `occupation_p` base class provides a set of pure virtual functions which have to be overloaded in the derived classes. In ScamPy's default implementation there are two different derived classes, which define slightly different parameterisations for the HOD algorithm [17, 23]. We show the inheritance graph of these custom types in Fig 2.6.

Member functions of the `halo_model` type provide the models needed by the likelihood defined in Eq. (1.1). To infer the best-fitting parameterisation we have to run a MCMC algorithm in order to sample the parameter space. This means to compute the models tens of thousands of times. In order to speed these calculations we define at the time of construction of the `halo_model` object a large number of interpolated functions. This operation makes construction computationally expensive: the constructor time scales with the thinness required for the interpolation grid. It takes approximately 2 seconds on a Intel i7 laptop with four physical processor and hyper-threading disabled to compute on a grid with 200 cells. The advantage of this setup though, is that we have to call the constructor only one time per run. Anyways, to speed up the constructor, we are computing the values of the functions in the interpolation grid by spawning multiple threads using the OpenMP API.

On the other hand, the computation of the models needed by the likelihood of Eq. 1.1 becomes extremely fast: a full-model computation runs in approximately 3 milliseconds.

This makes it also possible to run a full MCMC parameter space sampling on a regular laptop in a reasonable amount of time. Sampling a 4-dimensional parameter space with 8 walkers for $10^5$ chain-steps, takes around 5 minutes. We have widely benchmarked the performances of this class, results are shown in Section 4.3.

### 2.2.3 The `interpolator` class

The `c++` interpolation module we have developed is probably the most complex piece of code in all the API. It is deeply soaked in C++ OOP and mixes a variety of techniques to work. This complexity though allowed us to reach outstanding optimization and speed-up. The effort spent in designing and implementing this functionality definitely paid off.

The basic idea driving our implementation is to have a tool for making computations faster while maintaining the instrument manageable. To achieve the speed-up necessary for running MCMC samplers we needed fast computations of complex functions and, possibly, a fast integration scheme. Using a polynomial interpolation algorithm, such as a cubic spline, offers the following advantages:

- functions are defined by couples of arrays storing the x- and y-axis values of the function within some interval and with thinness, therefore precision, defined by the user;
- by using a polynomial interpolation, all the integrals reduce to an analytical form, thus not requiring any particular algorithm to be solved: the time to integrate a function inside a given interval becomes virtually zero;
- inversion of a continuous function also becomes trivial, since it only requires to switch the arrays storing the x- and y-axis and rebuild the interpolator. As a consequence, also zero finding becomes trivial since it requires to first invert the function and then evaluate it at zero.

The user interface to the functionalities of the module is provided by a class `interpolator`, defined in the header

`scampy/interpolation/include/interpolation.h`

which is listed in Appendix A.2.1. This templated class provides, by using static-polymorphism, a set of functions and overloaded operators to make all the operations extremely manageable. The crucial member functions are

- overload of the function call operator (`operator()`);
- overload of the algebraic operators (`operator+`, `operator*` and the equivalent in-place operators);
- function `integrate( min, max )` for interpolated integration in the interval [ `min`, `max` ].

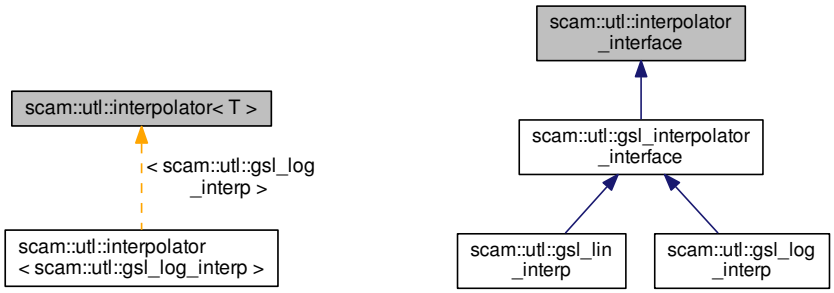


Figure 2.7: *Left panel:* relation between the default template instance of the `interpolator` class and the template class it is instantiated from. *Right panel:* Inheritance graph of the classes derived from the `interpolator_interface` custom type. Color codes are the same of Fig. 2.4. *(Obtained with Doxygen)*

With this interface the possibilities are huge, consider for example an integrand such as one of those defined in Section 1.2.1. If we have interpolators defined for each of the functions composing the integrand, we can obtain the interpolator for the integrand by calling only algebraic operators and then compute the integral by simply calling the dedicated member function.

The only disadvantage of this custom type is that it requires to compute in advance functions on a grid of given thinness. Even though in certain cases this might not be efficient, this operation has to be performed only once. Therefore this method easily provides a huge speed-up when said functions have to be called over and over again, for example when infering the values of some function parameters in a MCMC scheme.

The `interpolator` class is templated on the `interpolation_interface` type (Fig. 2.7, left panel). The latter is a pure virtual polymorphic type and is defined in the header

```
scampy/interpolation/include/interpolation_interface.h
```

which is listed in Appendix A.2.2. This makes inheritance from `interpolator_interface` mandatory, defining a precise framework for the usage of our `interpolator` object. Yet, since the template instance of the `interpolator` type is resolved at compile time, we avoid the runtime overhead due to dynamic polymorphism.

Having a pure virtual base class introduces an additional pointer to the memory load. Nevertheless, even in the application that most severely relies on the instantiation of `interpolator` objects (i.e. the `halo_model` object) the number of instances is tipically in the order of $10^2$. Considered these numbers and the gain in performance we obtain, using this set-up is worth the additional burden we introduce in memory. Furthermore, having a polymorphic template type makes the implementation extremely clear and easy to handle.

Types derived from `interpolation_interface` have to overload a set of mandatory functions to perform the construction of the interpolation scheme, the evaluation in a sub-grid position and integration. We provide our API with two different classes of this type:

- `gsl_lin_interp`: to perform interpolation on a linearly equispaced grid;
- `gsl_log_interp`: to perform interpolation on a logarithmically equispaced grid.

Both of these classes inherit from a bridging pure-virtual class called `gsl_interpolator_interface` whose role is to wrap methods from the `gsl_interpolation` library, defined in the headers:

```
gsl/gsl_interp.h
gsl/gsl_spline.h
```

This is one of the few external libraries used in this work and comes with accelerators and several possible interpolation schemes. Given its wide diffusion both among users from the Cosmology and Astrophysics community and among HPC facilities, the inclusion of this external dependency should not affect the interoperability of our API.

# Chapter 3

# Renewal & Optimization

In the context of software development for scientific usage and in general whenever the development is intended to the use in Academia, way too often the developer overlooks some crucial aspects that would make the usage flexible and that would favour the public to adopt it for their needs. It is a well known problem that most of the time a code that was intended for a usage limited to a small group becomes wide spread in the community. The lack in the diffusion of good practices in software development like cross-platform testing or the production of a reasonable documentation for the components of the API is also an issue, both for a public usage and for internal maintainability or extension.

In the development of our API for mock galaxy catalogue building we have considered all of these issues and outlined a strategy for keeping the software ordered and easy to read, while maintaining the computation efficient. The usage of advanced programming techniques along with the design of a handy class dedicated to interpolation also allowed to boost dramatically the performances of our code.

In this chapter we give an extensive overview of what we have done in this sense. Firstly, in Section 3.1 we describe our framework's structure and describe how future extensions should be managed. In Section 3.2 we describe the functionalities intended for the API accessibility and that would most impact in its diffusion to the community.

## 3.1 API structure & bushido

In this Section we give an overview of the framework we have developed, by describing its modularization and the main components that build up both the API and the repository where it is hosted.

The overall structure can be divided broadly into 4 main components:

- **C++ part** of the implementation - it mainly deals with the most computationally expensive sections of the algorithm (partially exposed in Section 2.2).
- **Python part** of the implementation - it provides the user interface and implements sections of the algorithm that do not need to be severely optimised (described through Section 2.1 and in Section 3.1).

- **Tests**, divided into **unit tests** and **integration tests**, are used for validation and consistency during code development. We address their description in Section 3.2.2.
- **Documentation**, provides the user with accessible information on the library's functionalities. It is described in Section 3.2.3.

In the spirit of modularization, both the test section and the documentation section are treated internally as modules of the library, and their development is, by some extent, independent to the rest of the API. Furthermore, being not essential for the API operation, their build is optional.

Let us discuss here about the source code structure. The C++ and Python implementations are treated separately and have different modularization strategies. As we already anticipated, the main role of a C++ section is in the first place to boost performances. Nevertheless, these core functionalities are also functioning independently and documented for this purpose.

Each logical piece of the algorithm (see Section 2.1 and Figure 2.1) has been implemented in a different module. Part of this division is available in both C++ and Python. This has been obtained through the implementation of source C++ code with a C-style interface enclosed in an `extern "C"` scope to produce shared-libraries with C-style mangling. All the source and header files whose names end with the `_c_interface.*` appendix are serving these purpose. They contain a set of function declarations and definitions to call the corresponding C++ functions and constructors/destructors. Their build produces a set of `_c_wrap.so` shared libraries that are then imported in Python by using the `ctypes` module.

In Table 3.1 we list all the C++ modules composing our API. In the second column we mark whether the given module comes with a C interface, note that in the last column all the modules marked with "yes" also produce a corresponding `c_wrap` shared library. All of these modules are hosted in different sub-directories with similar structure:

- `src` sub-directory, containing all the source files (`.cpp` extension);
- `include` sub-directory, containing all the header files (`.h` extension);
- a `meson.build` script for building.

Conversely, all the Python implementation is hosted by the same repository sub-directory. While keeping the two sets of sources physically separated, this separation contributes to both outline the structure of the Python interface that will be produced once the library has been installed. This structure also helps the auto-documentation of the docstrings written in the definition of all the functions and custom-types of the python section.

In Table 3.2 we provide a complete list of all the Python-modules provided to the user. They are divided between the C++ wrapped and the Python only ones. All of them are part of the `scampy` package that users can import by adding a

```
/path/to/install_directory/python
```

Table 3.1: C++ modules of the API. The first column lists the module names, the second marks whether the module is provided with a C-interface, the third column provides a short description of the library purpose and the last one shows the flag to be passed to the linker for including the libraries produced.

| **Module** | **C-wrap** | **Purpose** | **Libraries** |
|---|---|---|---|
| `utilities` | no | Provides a set of internal utility functions for vector management, integration, root finding, bit manipulation | `lutilities` |
| `interpolator` | yes | Templated classes and functions for cubic-spline interpolation in linear and logarithmic space | `linterpolation`, `linterpolation_c_wrap` |
| `fftlog_wrap` | no | Wraps in C++ a Fortran external library for 1D Fast Fourier Transform in logarithmic space. | `lfftlog_wrap` |
| `cosmology` | yes | Provides the interface and an implementation for cosmological computations that span from cosmographic to Power-Spectrum dependent functions, computations are boosted with interpolation | `lcosmo`, `lcosmo_c_wrap` |
| `halo_model` | yes | Provides classes for computing the halo-model derivation of non-linear cosmological statistics. It also provides the occupation probability functions. | `lhalo_model`, `lhm_c_wrap`, `locp_c_wrap` |

Table 3.2: Python modules of the API. The first column lists the module names and the second provides a short description of the module purpose. We divided the table in two blocks, separating the modules of the package that depend on the C++ implementation from those that have a pure Python implementation.

| **Module** | **Purpose** |
|---|---|
| | Wrapped from C-interface |
| `interpolator` | See Table 3.1 |
| `cosmology` | See Table 3.1 |
| `halo_model` | Provides classes for computing the halo-model derivation of non-linear cosmological statistics. |
| `occupation_p` | Provides the occupation probability functions implementation. |
| | Python-only |
| `objects` | Defines the objects that can be stored in the class `catalogue` of the `scampy` package, namely `host_halo`, `halo` and `galaxy`. |
| `gadget_file` | Contains a class for reading the halo/sub-halo hierarchy from the outputs of the SUBFIND algorithm of GaDGET. |
| `catalogue` | It provides a class for organizing a collection of host-haloes into an hierarchy of central and satellite haloes. It also provides functionalities for authomatic reading of input files and to populate the Dark Matter haloes with objects of type `galaxy`. |
| `abundance_matching` | Contains routines used for running the SHAM algorithm. |

to their `PYTHONPATH`.

A sub-package dubbed `cwrap` is also present inside the `scampy` package. In its sub-directory the `cwrap.py` file is installed at build-time. This is only importable locally by the modules listed in Table 3.2. It contains all the function calls, made through `ctypes`, to the C-interface shared libraries. It also provides the corresponding configurations of function argument types and returned type.

### 3.1.1 External dependencies

Very often scientific codes depend severely on external libraries. Even though a golden rule when programming, especially with a HPC intent, is to *not reinvent the wheel*, external dependencies have to be treated carefully. If the purpose of the programmers is to provide their software with a wide range of functionalities, while adopting external software where possible, the implementation can easily become a *dependency hell*.[1]

For this reason we decided to keep the dependence on external libraries to a reasonable minimum. The leitmotiv being, trying not to be stuck on bottlenecks requiring us to import external software, while maintaining the implementation open to the usage along with the most common scientific softwares used in our field.

The `c++` section of the API depends on the following external libraries:

- **GNU Scientific Library** [20, version 2 or greater]: this library is widely used in the community and compiled binary packages are almost always available in HPC platforms.

- **FFTLog** [24]: also this library is a must in the cosmology community. In our API we provide a `c++` wrap of the functions that were originally written in Fortran90. We have developed a patch for the original implementation that allows to compile the project with Meson (see Appendix B for details).

- **OpenMP**: one of the most common APIs for multi-threading in shared memory architectures.

We are aware that a large collection of libraries for cosmological calculations is already available to the community. The intent of our `cosmology` module is not to substitute any of these (as explained in Section 2.2.1). By using polymorphism (both static and dynamic) we tried to keep our API as much flexible as possible. We explicitly decided to not force the dependence to any specific Boltzmann-solver to obtain the linear power spectrum of matter perturbations (see Section 1.2.1), the choice is left to the user.

Furthermore, the choice of Python to build the user interface, allowed to easily implement functions that do not require any other specific library to work. An example is the `abundance_matching` module, which is almost completely independent to the rest

[1] *Thou shall beware of the dependency hell,*
*for it makes portability unfeasible*
*and your day will turn miserable.*

of the API: the only other internal module needed is the `scampy.object` but all its functionalities can be obtained by using python lambdas and numpy arrays.

The only other python libraries used in ScamPy are:

- **CTypes** which is part of the python standard and is used for connecting the C-style binaries to the Python interface.
- **Numpy** which, despite not being part of the standard, is possibly the most common python library on Earth and provides a large number of highly optimized functions and classes for array manipulation and numerical calculations.

### 3.1.2 Extensibility

Simplifying the addition of new features has been one of our objectives from the first phases of development. We wanted to be able to expand the functionalities of the API, both on the C++ side, in order to boost the performances, and on the Python side, in order to use the API for a wide range of cosmological applications.

This is easily achieved with the modular structure we have built up. Adding a new `c++` module reduces to including a new set of headers and source files in a dedicated sub-directory. In Listing 3.1 we show the suggested structure said directory should have.

```
$ tree module_name
module_name/
|--- include
|    |--- header_1.h
|    |--- ...
|    +--- header_N.h
|--- meson.build
+--- src
     |--- src_1.cpp
     |--- ...
     +--- src_M.cpp

2 directories, (N + M + 1) files
```

Listing 3.1: Example tree-structure of `c++` module directory. It hosts two sub-directories for organizing header and source files separately and a build script.

Note that, in order to integrate the new feature in the build system, adding a `meson.build` script is necessary. In Section 3.2.1 we describe the chosen build system in detail. The last step of the integration would then be to add the directive

```
subdir( 'module_name' )
```

to the root `meson.build` script.

Adding new modules to the Python interface is even simpler, as it only requires to add a new dedicated file in the `python/scampy` sub-directory. Eventually, it can be also appended to the `__all__` list in the `python/scampy/__init__.py` file of the package. In this case, it is not necessary to intervene on the build system as it will automatically install the new module along with the already existing ones.

## 3.2 Best practices

Outlining a precise definition of all the recommendations and directives that are included in the wide name of *best practices* is not easy. They are indeed a set of practices that a software developer is recommended to follow when designing a piece of code. However, since they span from indications on how the code itself should be written to the metadata that have to be provided to the user, what has to be considered and what is not crucial changes case by case. An intelligent design of the API structure (which we discussed in Sec. 3.1) and the usage of a distributed control system, such as Git, can be considered best practices as well. Here we will outline all the features we have provided our library with, that did not find place in any other Section of this manuscript.

### 3.2.1 Build system

Automation of the build process involves the development of scripts and the usage of software to compile source code into binary code and/or to install it in the chosen position of the filesystem.

When we first started working on our API, the build system was based on a set of Makefiles dedicated to the compilation of each different module of the source code. While the software was growing larger though, we realised that it would be far more maintainable and user-friendly porting everything in a build-script generator.

For the automation of the build process we choose the Meson build system [25]. Meson is free and open-source software written in Python, under the Apache License 2.0. It is intended for performance and for boosting the programmer productivity.

Much like `CMake` [26], it does not build directly, but it rather generates files to be used by a native build tool. Meson has been first developed for Linux but, being written in Python, it works on most of the Unix-like operative systems (OS), including macOS, as well as Microsoft Windows and others. This interoperability is also achieved by using different native build tools for different OS: it uses `ninja` [27] in Linux, `MSbuild` [28] on Windows and `xcode` [29] on macOS. It supports a wide range of compilers, including the GNU Compiler Collection, Intel compilers and Clang.

The build is controlled by a master `meson.build` script located in the root directory of the repository. In this file we first specify the basic setup of the build, setting

- the project type, `cpp` in our case;
- some default options to be used at compile time:
  - the C++ standard, i.e. `c++14`;
  - the build type, i.e. `plain`, `debug`, `debugoptimized` or `release`;
  - the directory structure of the compiled library, i.e. the installation prefix and all its eventual subdirectories (lib, include, share).
- the release version and license for the source code;

- the optional compile time flags.

We then list the external dependencies required to build the API, these will be either searched by meson using pkgconfig or downloaded from the specified mirrors and installed in the specified position.

Lastly, we go through each subdirectory hosting the different modules composing the API. As described in Section 3.1, each module composing our API is stored in a dedicated dirictory which comes with a `meson.build` file that provides the build directives proper to the module. All these build scripts have the very same structure, making them easily reproducible in case of extension of the API with new modules and functionalities.

```
###############################################################################
# Internal includes

inc_dir = 'include'
inc_utl = include_directories( inc_dir )

###############################################################################
# Utilities Lib

utl_lib = library( 'utilities',
                   [ 'src/utilities.cpp',
                     'src/bit_manipulator.cpp' ],
                   include_directories : inc_utl,
                   dependencies : [ gsl_dep ],
                   install : true  )

utl_dep = declare_dependency( link_with : utl_lib,
                              include_directories : inc_utl )

lib_headers += [ root + '/utilities/' + inc_dir + '/utilities.h',
                 root + '/utilities/' + inc_dir + '/error_handling.h',
                 root + '/utilities/' + inc_dir + '/bit_manipulator.h' ]

###############################################################################
```

Listing 3.2: Example `meson.build` script file for the module `utilities` of ScamPy.

In List. 3.2 we show the `meson.build` script file that builds the `utilities` module of our API. It is divided in two sections, the first one specifies the location of the header files and stores it in an object of type `include_directories`. The second one deals with building and installation directives. Firstly, it defines the `library` object, assigning a name, listing all the source files that have to be compiled, and specifing the internal and external dependencies.

It then declares the compiled module as an internal dependency and adds all the header files to the list of files that will be copied in the

`/path/to/install_prefix/include_dir`

sub-directory of the compiled library.

Installation using meson and the native build tool is trivial and as simple as calling the following commands from the repository's root directory:

- Linux and Unix-like systems:

```
$ meson /path/to/build_dir -Dprefix=/path/to/install_prefix
$ ninja -C /path/to/build_dir install
```

- Microsoft Visual Studio:

```
$ py meson /path/to/build_dir --backend vs2015/ninja
```

Note that we have added two configuration options to decide whether to build the API with testing and documentation enabled:

```
-DENABLE_TEST=true/false
-DENABLE_DOC=true/false
```

The default values are set to `false` to speed-up the build and to avoid the necessity of additional external dependencies.

### 3.2.2 Testing

We have developed a testing environment to guarantee consistency and validity of the API modules. Tests are treated as a module and are hosted in a dedicated subdirectory of the repository. The environments are different for unit tests and integration tests, we will here give a brief description of the organisation of these two sub-modules and the structure for potential expansions. The master branch of the official API's repository already contains some tests to check that its core functionalities are behaving as expected. Further tests will be added by the time the scientific paper we are extracting from this work is accepted for publication.

#### Unit tests

We use google's `gtest` for the unitary testing of the API in all its C++ components. `gtest` offers a handy testing environment with a wide range of macros. It provides assertions for binary comparison and for checking the success of commands.

The environment requires a dependency to the `gtest` library which, in our build system, we treat as a subproject.

Integration of the google testing facility in the Meson build system is straightforward. By compiling the project with the flag

```
-DENABLE_TEST=true
```

we have to simply run the command

```
$ ninja -C /path/to/build_dir test
```

Ninja will take care of running all the defined tests and inform the user about which have been passed and which not.

#### Continuous integration

Continuous Integration (CI) is the practice of merging small code changes frequently rather than merging a large change at the end of a development cycle. The goal is to build healthier software by developing and testing smaller increments. We are using Travis CI [30] for continous integration testing. It is a platform that automatically builds and tests code changes, providing immediate feedback on the success of the modification. This is achieved by linking the Travis CI platform to the remote GitHub repository of the project. It then automatically clones the repository into a remote docker machine with the specifications provided by the programmer.

The development of specific integration tests is still a work in progress. At the current state we are letting the CI tool to check that the build is successful and that the unit tests are passing. Everything is being built successfully, makeing us confident on the interoperability and on the cross-platform compilation capability of the API we have developed.

### 3.2.3 Documentation

Users of scientific software are well aware of how crucial it is to have at disposal an extensive and accessible documentation. In the development of ScamPy, we adopted these prescriptions by writing down both the unitary components documentation and a set of in-code comments to help understanding the source. We have also exploited several automation tools for the production of the manual in HTML, latex and/or man format.

We are using an hybrid system to have automatic documentation of both the C++ and the Python functions and classes. The build of the documentation proceeds as follows:

- we run Doxygen [31] to produce the C++ documentation in XML format;
- by running Sphinx [32] we have a user-friendly environment based on, but not limited to, reStructuredText directives to produce clear HTML pages. Sphinx also supports extensions, we use:
  - breathe to convert the XML code generated by Doxygen into HTML understandable by Sphinx;
  - autodoc for automatic interpretation and conversion in HTML format of Python docstrings.

Finally, with this setup, it is straightforward to upload automatically the produced documentation online through Read the Docs [33]. This online platform provides continuous integration of documentation by linking it to the remote GitHub repository.

# Chapter 4

# Verification & Validation

We have extensively tested all the functions building up our API in all their unitary components. As we have already shown, we developed a testing machinery, included in the official repository of the project, to run these tests in a continuous integration environment. This will both guarantee consistency during future expansions of the library as long as provide users with a quick check that the build have been completed successfully.

In this section we show that our machinery is satisfying the requirements, both in terms of scientific results and performances. Specifically, in Section 4.1 we will show that the mock-catalogues obtained with ScamPy reproduce the observables we want.

We have also tested our API for the accuracy in reproducing cross-correlations in Section 4.2. Even though there is no instruction in the algorithm that guarantees this behaviour, using the halo model it is trivial to obtain predictions for the cross-correlation of two different populations of objects.

All the validation tests have been obtained by assuming a set of reasonable values for the HOD parameters. The resulting occupation probabilities have been then used to populate a set of halo/subhalo catalogues. These catalogues have been obtained by running on the fly the FoF and SUBFIND algorithms on top of a GaDGET-3 simulation. The cosmological parameters used for this simulation are summarised in Table 4.1. The

Table 4.1: Fiducial cosmological parameters of the N-body simulation used in this Section.

| $h$ | $\Omega_{\rm CDM}$ | $\Omega_b$ | $\Omega_\Lambda$ | $\mathcal{A}_s$ | $n_s$ |
|---|---|---|---|---|---|
| 0.67 | 0.27 | 0.05 | 0.68 | $2.1e-9$ | 0.96 |

simulation we run has 64 Mpc/$h$ boxside lenght and $512^3$ DM particles. Even though this is far from having a mass and a spatial resolution high enough for getting scientifically significant results, especially at high redshift, it is enough for testing. The speed of the populating algorithm depends on the number of objects contained in the input catalogue, thus we have preferred to run tests on a smaller catalogue.

From the plan we outlined in Section 1.3, one of the requirements of this effort was to

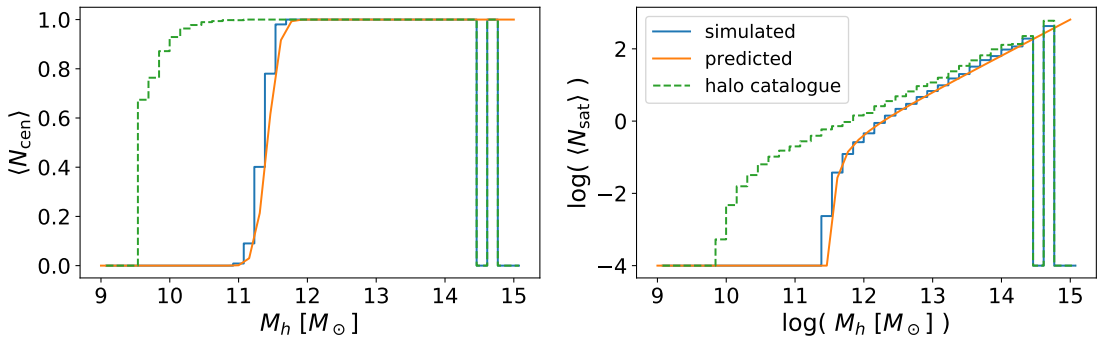


Figure 4.1: Occupation probability functions for central subhaloes (*left panel*) and satellite subhaloes (*right panel*). The orange solid line marks the model we want to reproduce. The green step-wise dashed line and the blue step-wise solid line mark the distributions measured on the subhalo catalogue before and after having applied our populating algorithm.

boost the performances of the algorithm. In Section 4.3, we will present some measures of the API performances to show the improvements we have obtained.

## 4.1 Observables

Here we present measurements obtained after both the populating algorithm of Section 2.1.1 and the abundance matching algorithm of Section 2.1.2 have been applied to the DM-only input catalogue. Applying the abundance matching algorithm does not modify the content of the populated catalogue, besides associating to each mass an additional observable-property.

In the two panels of Fig. 4.1 we show the abundances of central and satellite subhaloes as a function of the halo mass. The dashed green step-lines show the distribution in the DM-only input catalogue, while the orange solid line marks the distribution defined by the occupation probability functions. By applying the populating algorithm (Section 2.1.1) to the input catalogue we obtain the distributions marked by the solid blue step-lines, which are in perfect agreement with the expected distribution.

We then draw a random Gaussian sample around the halo model estimates of the abundance and clustering, respectively Eqs. (1.5) and (1.6), of objects with the above selection of occupation probabilities. This will be our *mock dataset*. We then run an MCMC sampling of the parameter space, with the likelihood of Eq. (1.1), to infer the set of parameters that best fit the mock dataset. For sampling the parameter space we used the Emcee [21] Affine Invariant MCMC Ensemble sample, along with ScamPy python interface.

After we obtain the best-fit parameters, we produce 10 runs of the full pipeline described in Alg. 1. In doing this, we are producing 10 different realisations of the resulting mock catalogue. Since the selection of the host-subhaloes is not deterministic,

this procedure allows to obtain an estimate of the errors resulting from the assumptions of the halo model. Finally, we use the Landy-Szalay [34] estimator in each of the populated catalogues to measure the 2-point correlation function:

$$\xi(r) = \frac{DD(r) - 2DR(r) + RR(r)}{RR(r)} \tag{4.1}$$

where $DD(r)$ is the normalised number of unique pairs of subhaloes with separation $r$, $DR(r)$ is the normalised number of unique pairs between the populated catalogue and a mock sample of objects with random positions, and $RR(r)$ is the normalised number of unique pairs in the random objects catalogue. We then measure with Eq. (4.1) the clustering in each of the 10 realisations and we compute the mean and standard deviation of these measures in each radii bin.

The results are shown in Fig. 4.2 for redshift $z = 0$ and 4.3 for redshifts $z = 2, 4, 6, 8$. In the upper panel of Fig. 4.2 we show the mock dataset with triangle markers and errors, the lines show the halo model estimate of the 2 point correlation function (with the different contributes of the 1- and 2-halo terms). The circle markers show the average measure obtained from our set of mock-catalogues.

In the lower panel of Fig. 4.2 and in the four panels of Fig. 4.3 we show the distance of both the mock-dataset and of the average measure with respect to the inferred model. First of all, let us notice that the models inferred are in good agreement with the mock dataset. The accuracy of the measure with respect to the model, instead, decreases as redshift increases. At the largest scales we have a decrease in power of the measured 2-point correlation function. This is a known weakness of the HOD method. In literature there have been a lot of effort in quantifying and correcting this effect [see e.g. 35, 36, for two recent works], which is thought to result from a concurrence of box-size effects, cosmic-variance and assembly bias.

On the other hand, the limited spatial resolution of our simulation box is responsible for the discrepancies at the smaller scales. Since at redshift greater than $2 \div 3$ the number of sub-structures within haloes found by the SUBFIND algorithm becomes substantially smaller, the discrepancy becomes larger.

The discrepancies at small scales might be partially due to the transition between 1- and 2-halo term, as the bump in the measures seams to suggest especially at redshift $z = 4$. These discrepancies could be corrected by applying the same pipeline to an N-body simulation with higher resolution.

The requirement of reproducing the 1-point statistics of the original catalogue is necessary to have the expected observational property distribution in the output mock-catalogue. This requirement guarantees that the abundance matching scheme will start associating the observational property from the right position in the cumulative distribution, i.e. from the abundance corresponding to the limiting value that said property has in the survey.

In Fig. 4.4, we show the example case of the UV luminosity function. The halo-model prediction for the total abundance of sources is $n_g^{\rm hm}(z) = 3.49 \cdot 10^{-2}$ $[h^3{\rm Mpc}^{-3}]$, while we measure $n_g^{\rm pop}(z) = (3.06 \pm 0.04) \cdot 10^{-2}$ $[h^3{\rm Mpc}^{-3}]$ in the populated catalogue.

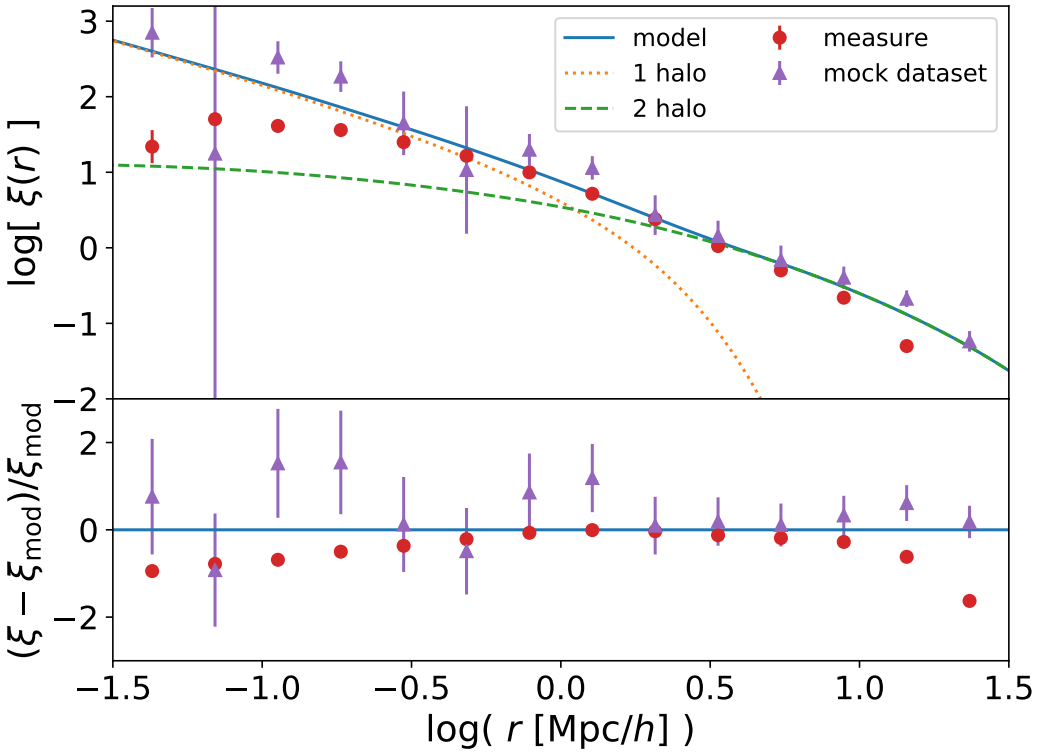


Figure 4.2: Validation of the two point correlation function at redshift $z = 0$. *Upper panel:* Comparison between the mock clustering dataset (triangles), the best-fit halo-model prediction (solid line) and the mean and standard deviation of the clustering measured with the Landy-Szalay estimator on the 10 realisations (cicles and errors). *Lower panel:* distance ratio between the best-fit halo-model prediction and the mock-dataset/averaged-measure.

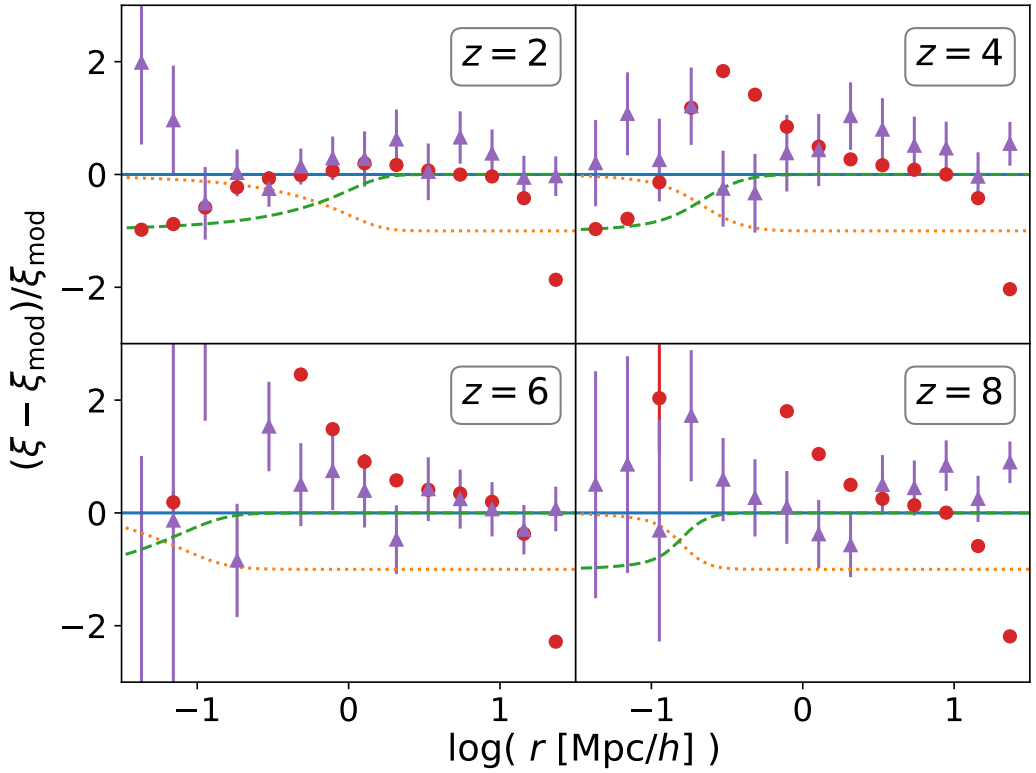


Figure 4.3: Same as lower panel of Fig. 4.3 for redshift $z = 2, 4, 6, 8$. Here we also show the modeled 1-halo (orange dotted line) and 2-halo (green dashed line) terms for reference.

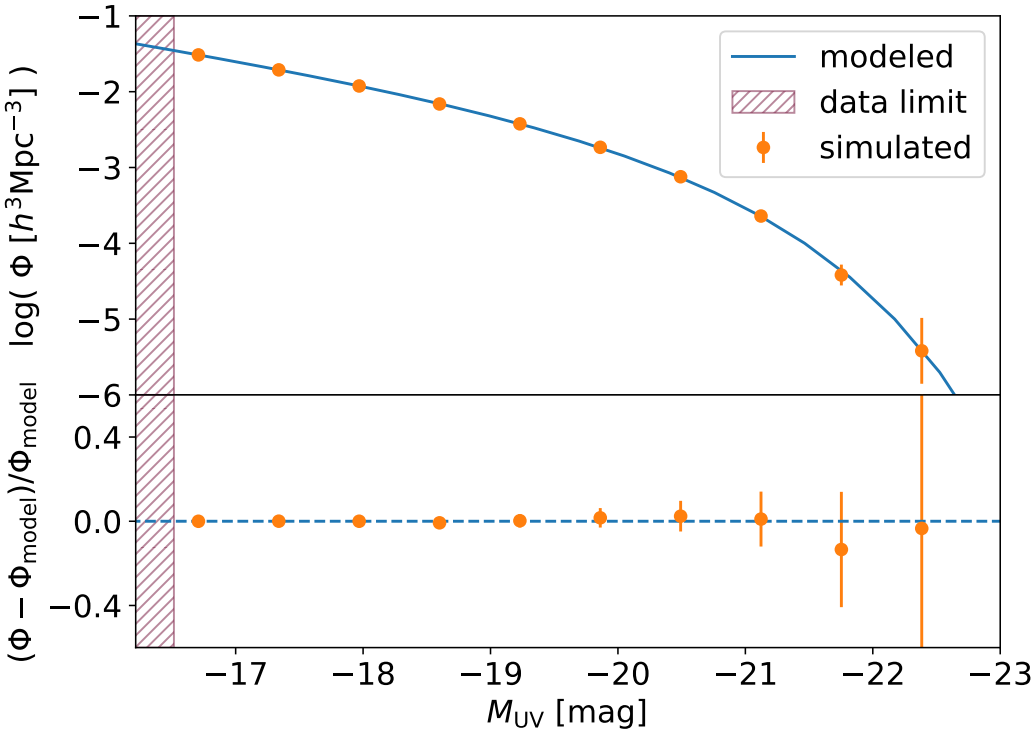


Figure 4.4: Cumulative luminosity function at redshift $z = 0$. *Upper panel:* the blue solid line marks the model prediction while the orange circles with errors the distribution measured on the populated catalogue. The hatched red region marks the limiting magnitude $M_{\rm lim}^{\rm UV}$. *Lower panel:* distance ratio between the luminosity function measured on the populated catalogue and the model.

In Fig. 4.4, we mark with orange cicles the cumulative luminosity function measured on the mock-catalogue after the application of our API. For comparison we also show the luminosity function model we are matching (blue line) and the observation limit of the target population (red dashed region).

As it is shown in the lower panel of Fig. 4.4, the distance ratio between the expected distribution and the mock distribution is lower than $\approx 10\%$ over all the range of magnitudes.

## 4.2 Multiple populations cross-correlation

Even though in our API there is no prescription for this purpose, it is interesting to test how the framework performs in predicting the cross-correlation between two different populations. This quantity measures the fractional excess probability, relative to a random distribution, of finding a mock-source of population 1 and a mock-source of population 2, respectively, within infinitesimal volumes separated by a given distance.

It is simple to modify Eqs. (1.10) and (1.11) to get the expected power spectrum of the cross-correlation. For the 1-halo term this is achieved by splitting the $(M_h/\overline{\rho})^2$ of Eq. (1.10) in the contribution of the two different populations, which leads to the following equation:

$$P_{1h}^{(1,2)}(k,z) = \frac{1}{n_g^{(1)}(z) n_g^{(2)}(z)} \int_{M_{\min}}^{M_{\max}} N_g^{(1)}(M_h) N_g^{(2)}(M_h) n_h(M_h) |\widetilde{u}_h(k, M_h, z)|^2 \mathrm{d}M_h \tag{4.2}$$

where quantities referring to the two different populations are marked with the superscripts (1) and (2).

In the case of the 2-halo term, obtaining an expression for the cross-correlation requires to divide the two integrals of Eq. (1.11) in the contributions of the the two different populations, leading to

$$\begin{aligned} P_{2h}^{(1,2)}(k,z) = & \frac{P_m(k,z)}{n_g^{(1)}(z) n_g^{(2)}(z)} \cdot \\ & \left[ \int_{M_{\min}}^{M_{\max}} N_g^{(1)}(M_h) n_h(M_h) b_h(M_h, z) \widetilde{u}_h(k, M_h, z) \mathrm{d}M_h \right] \cdot \\ & \left[ \int_{M_{\min}}^{M_{\max}} N_g^{(2)}(M_h) n_h(M_h) b_h(M_h, z) \widetilde{u}_h(k, M_h, z) \mathrm{d}M_h \right] \end{aligned} \tag{4.3}$$

We get the cross-correlation of the two mock-populations using a modification [37] of the Landy-Szalay estimator

$$\xi^{(1,2)}(r) = \frac{D_1 D_2(r) - D_1 R_2(r) - D_2 R_1(r) + R_1 R_2(r)}{R_1 R_2(r)} \tag{4.4}$$

where $D_1 D_2(r)$, $D_1 R_2(r)$, $D_2 R_1(r)$ and $R_1 R_2(r)$ are the normalized data1-data2, data1-random2, data2-random1 and random1-random2 pair counts for a given distance $r$.

In Fig. 4.5 the red circles show the cross-correlation measured with Eq. (4.4) for two different mock-populations with a dummy choice of the occupation probabilities' parameters. Errors are measured using a bootstrap scheme with 10 sub-samples. For comparison, in the upper panel we also show the halo-model prediction of the two-point correlation function, obtained with Eqs. 4.2 and 4.3, separated in 1- and 2-halo term contribution. The lower panel of Fig. 4.5 shows the distance ratio between the measure and the model prediction, which is lower than 40% over almost all the scales inspected.

## 4.3 Performances & Benchmarking

We have measured the performances of our API's main components and benchmarked the scaling and efficiencies of the computation at varying precision and work-load. The code from which we started the development was written only in C++ and it was almost completely serial. We were using extensively an external cosmological library not intended for high performance computing to get most of the cosmological functions we

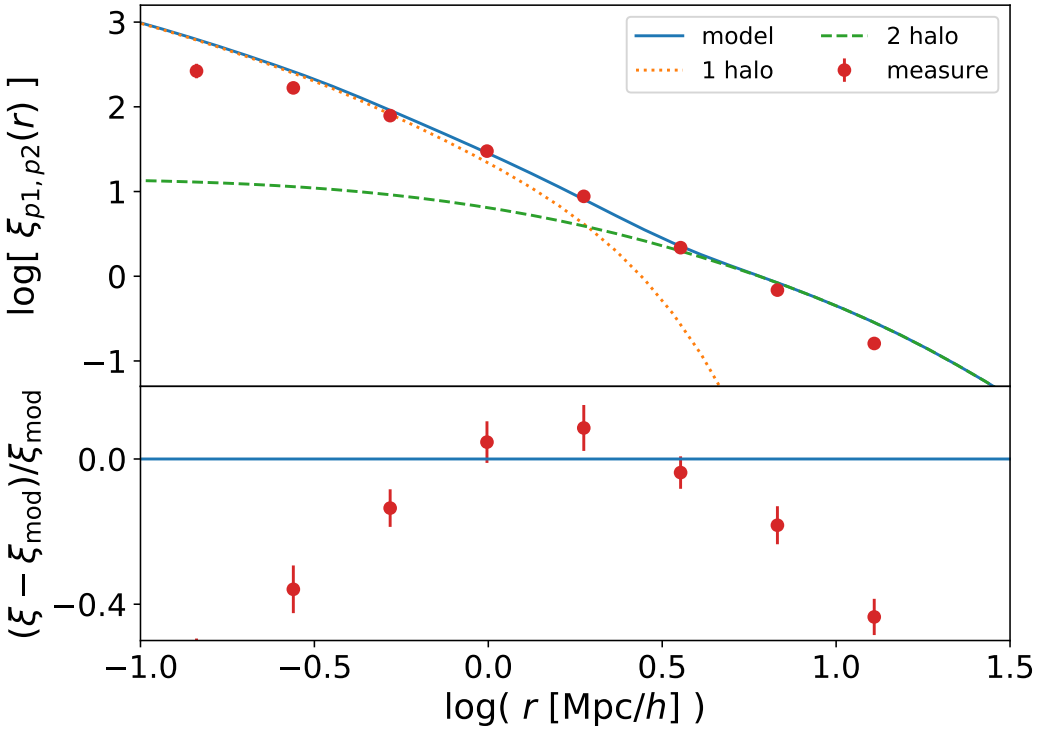


Figure 4.5: Same comparison as in Fig. 4.2 for the cross-correlation function, measured with the modified Landy-Szalay estimator of Eq. (4.4), between two dummy mock-populations at redshift $z = 0$.

were using and this, besides making compilation difficult for the big amount of external dependencies required, was affecting severely the overall performances.

In our work we have developed a hybrid API in which the heaviest function are computed in C++ and then wrapped in python using C as bridging language. This allows to keep computation fast, while providing a user-friendly interface which can exploit easily a wide range of external libraries available in python.

We will show here a set of time measures performed on the two main components of the library: the `cosmology` class and the `halo_model` class. These are the two classes that will be mostly used in real-life applications of our API. In the original C++-only implementation, the functions now provided by the `cosmology` class were imported from the external cosmological library, while the `halo_model` class was already implemented.

We re-implemented from scratch most of the functions that were already present in our heritage collection of codes while adding new functionalities and wrapping everything in python.

First of all, in Tab. 4.2, we show the execution time of the same function called from different languages. Since our hybrid implementation requires to bridge through C to wrap in python the optimisations obtained in C++, it is interesting to compare their respective execution time. Along with the execution time in C and python, we also provide the ratio with respect to the reference C++ time, $t_{\rm C++}$, for the same function. All the times are expressed in nanoseconds.

Table 4.2: Execution time in nanoseconds of the same function in different languages. For the C and python case, we also show the ratio with respect to the C++ execution time. The timings reported are the average of 10 runs on the 4 physical cores with hyper-threading disabled of a laptop with Intel® Core™i7-7700HQ 2.80GHz CPU.

| **Function** | **C++** | **C** | $t_{\rm C}/t_{\rm C++}$ | **Python** | $t_{\rm py}/t_{\rm C++}$ |
|---|---|---|---|---|---|
| `cosmology` class | | | | | |
| c.tor | 2.520e+05 | 2.780e+05 | 1.103 | 8.892e+05 | 3.528 |
| $d_C(z)$ | 2.083e+03 | 3.200e+03 | 1.536 | 5.984e+03 | 2.873 |
| $n(M,z)$ | 1.186e+08 | 1.187e+08 | 1.001 | 1.197e+08 | 1.009 |
| `halo_model` class | | | | | |
| c.tor | 3.011e+09 | 3.024e+09 | 1.004 | 2.961e+09 | 0.983 |
| $n_g(z)$ | 1.917e+04 | 2.280e+04 | 1.190 | 2.851e+04 | 1.488 |
| $\xi(r,z)$ | 3.396e+06 | 8.729e+06 | 2.570 | 1.784e+06 | 0.525 |

We are showing 3 typical member calls that are representative of the functionalities provided by the two classes. For both of them we measured the constructor time (c.tor), the time for executing a function that returns a scalar ($d_C(z)$ and $n_g(z)$) and the execution time for a function returning an array ($n(M,z)$ and $\xi(r,z)$). It can be noticed that, especially for the `cosmology` class, by calling the same function in C and python, the execution time increases. The worst case is the `cosmology` class constructor time that

looses a factor $\sim 3.5$ in python. It has to be noticed though, that the execution time is lower than a millisecond and, since the constructor is the member function that is called the less, this is not severely affecting the overall performance of the python interface.

Nonetheless, because of the larger number of function calls required by moving from one language to another, loosing some performance is expected. What we did not expect is the gain in performance we are getting when moving to python, as it is shown in the last column of the `halo_model` class box of Tab. 4.2. This behaviour might be due to the different way memory is allocated, accessed and copied in python with respect to C++/C. Moreover, the timers used for measuring the execution in the different languages are different. Even by comparing measures taken with the same precision, it is not guaranteed to have the same accuracy. We will further investigate on this behaviour in the future.

We have then tested the execution time of the `halo_model` constructor and member functions at varying work-load. In our implementation (see Sec. 2.2.2), the `halo_model` class requires to define a set of `interpolator` objects at construction time. The interpolation accuracy, as explained in Sec. 2.2.3, depends on the resolution of the interpolation grid. In the `halo_model` class, at fixed limits of the interpolation interval, this is controlled by the *thinness* input parameter, which takes typical values $50 \div 200$ in real-life applications.

In Fig. 4.6 we show how the constructor-time varies with varying thinness in the range $10 < thin < 10^3$. The plot is obtained by calling 10 times the constructor per each thinness value and then averaging (solid lines). The shaded region marks the best and worst execution time among the 10 runs. Instead of the actual execution time we show the percent distance with respect to perfect linear scaling (dashed line) for both the C++ case (blue) and the python case (orange). We define the percent distance at given thinness as

$$\%\ \text{distance}(thin) \equiv 100 \cdot \frac{t(thin) - t_{\rm lin}(thin)}{t_{\rm lin}(thin)} \tag{4.5}$$

where $t_{\rm lin}(thin)$ is the execution time for given thinness in the linear scaling case, computed with respect to the C++ case. In the white text box of Fig. 4.6 we also show the C++ constructor time for $thin = 8$, as a reference. As the picture shows, the scaling is almost perfectly linear, with a maximum distance of the 0.06% in the C++ case.

Possibly the most crucial aspect of the whole API is the time taken by the computation of a full-model. With the term "full-model" we mean the execution of the two functions for computing the halo-model estimate of the 1- and 2-point statistics, namely $n_g(z)$ and $\xi(r, z)$. In an MCMC framework, while the constructor is called only once, these two functions are called tens of thousands of times. In our heritage implementation, as also pointed out in Sec. 1.3, this computation was taking times of the order of 10 seconds, making the parameter-space sampling virtually unfeasible in a reasonable amount of time.

As also shown in Tab. 4.2, the execution of the two single functions now takes an amount of time which is in the order of the millisecond in the C++ case. We can also notice that the execution time of a full-model is dominated by the computation of the

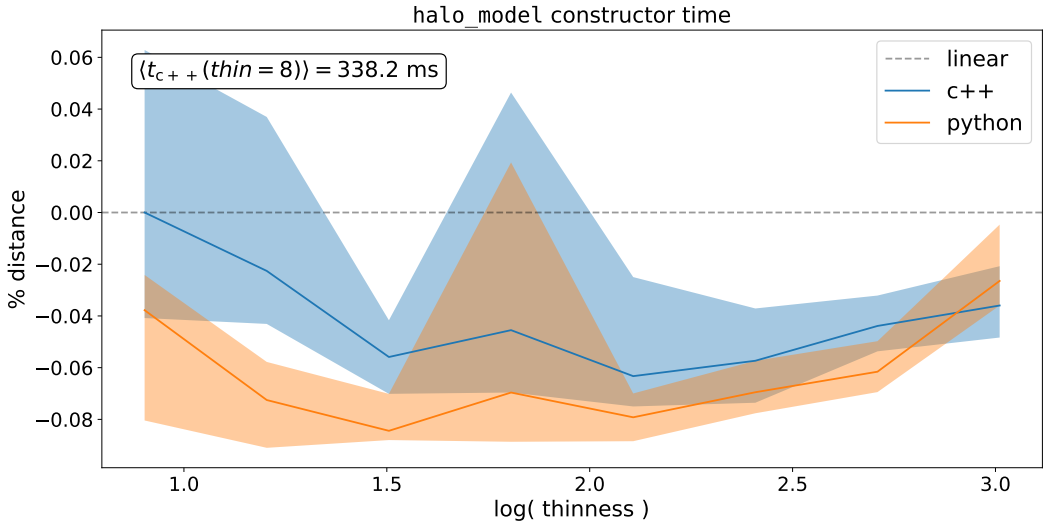


Figure 4.6: Percent distance between the constructor time scaling at varying thinness and the linear scaling case. For the python case, the percentage is computed with respect to the C++ time to ease the comparison. For reference, we also show in the white text-box the measured constructor time with thinness = 8.

two point correlation function, $\xi(r, z)$. Since this function is operating on a vector and returning a vector, it is reasonable to expect that its execution time varies with the work-load, i.e. with the vector size.

In Fig. 4.7 we show the percent distance, defined as in Eq. (4.5), of the average full-model execution time at varying work-load (solid lines) with respect to the perfect linear scaling case (dashed line), in the range $2^3 \leq \text{load} \leq 2^{14}$. The measures are obtained by averaging the results of 10 runs in both C++ (blue) and python (orange). The shaded regions mark the best and worst performance among all the runs at varying workload. It can be noticed that, by increasing the work-load, the average execution time gets up to 15% worse than perfect linear scaling. This is due to some latency introduced by the necessity of Fourier transforming the power spectrum to get the clustering measure. We have to point out though, that the typical work-load is in the range $5 \div 15$ for real-life applications and that the execution time in this cases is of the order of the millisecond.

In Tab. 4.3 we quantify the gain in performances obtained with our optimisation of the halo model functions. We have compared the time spent by the old and the new implementation to perform the same task. In terms of performances, as Tab. 4.3 clearly shows, the impact of our renewal is dramatic. We gained almost a factor of 10 for the constructor time and **up to a factor $10^4$ (ten thousand!)** for the full-model time.

Finally, we have measured how the constructor time scales with increasing number of multi-threading processors. We did not perform this measure for the full-model computation because, in the perspective of using it in a MCMC framework with parallel

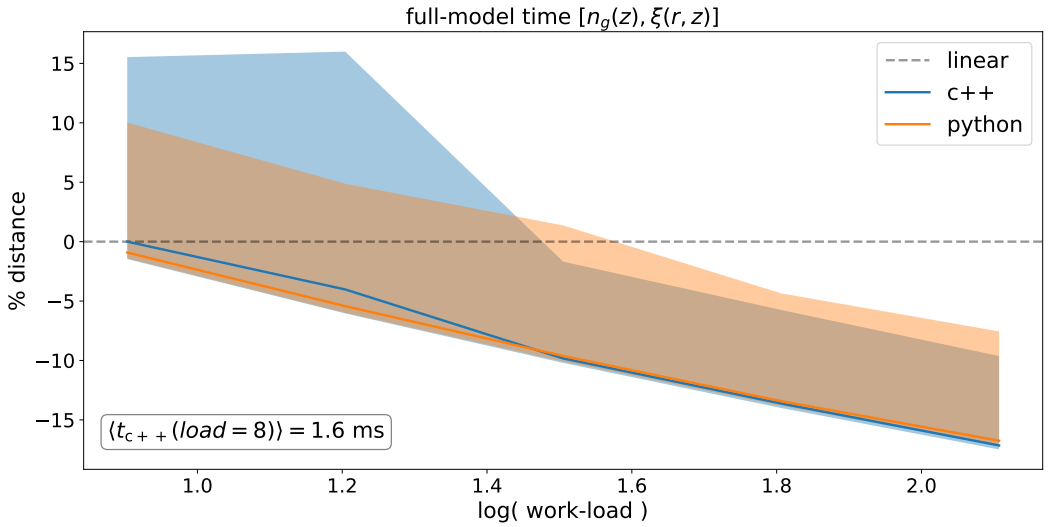


Figure 4.7: Percent distance between the full-model time scaling at varying work-load and the linear scaling case. For the python case, the percentage is computed with respect to the C++ time to ease the comparison. For reference, we also show in the white text-box the measured constructor time with work-load = 8.

Table 4.3: Performance comparison between the old and the new implementation. We have both tested the constructor time and the full-model time. Measures have been performed at fixed work-load and varying thinness and vice-versa. All the times are expressed in nanoseconds.

| **thinness** | **work-load** | **c.tor** | | **full-model** | |
|---|---|---|---|---|---|
| | | **old** | **new** | **old** | **new** |
| Varying thinness | | | | | |
| 64 | 16 | 4.549e+9 | 0.585e+9 | 16.601e+9 | ∼2.0e+6 |
| 128 | 16 | 9.071e+9 | 1.212e+9 | 19.467e+9 | ∼2.0e+6 |
| 256 | 16 | 18.107e+9 | 2.643e+9 | 21.182e+9 | ∼2.0e+6 |
| Varying work-load | | | | | |
| 64 | 16 | 4.549e+9 | ∼0.6e+9 | 16.689e+9 | 1.997e+6 |
| 64 | 128 | 4.548e+9 | ∼0.6e+9 | 16.510e+9 | 1.566e+6 |
| 64 | 1024 | 4.545e+9 | ∼0.6e+9 | 16.574e+9 | 1.934e+6 |

walkers, the full-model will be computed always serially.

We present measures of both the constructor time *strong scaling* and *weak-scaling*. While the first measures the scaling with processor number at fixed thinness, the latter measures the scaling at thinness increasing proportionally with the processor number.

First of all, let us define the *speed-up*

$$S(p) = \frac{t(1)}{t(p)} \tag{4.6}$$

where $p$ is the number of processors and $t(p)$ is the time elapsed running the code on $p$ processors. This quantity measures the gain in performances one should expect when having access to larger parallel systems.

We also define the efficiency for the strong and the weak scaling case:

$$\begin{aligned} E_{\text{strong}}(p) &= \frac{S(p)}{p} \\ E_{\text{weak}}(p) &= S(p) \end{aligned} \tag{4.7}$$

This quantity roughly measures the percentage of exploitment of the parallel system used. Thus, providing a hint of how much the serial part of the code is affecting the gain we can expect from spawning multiple threads.

We run these measures on a node from the `regular` partition of the SISSA Ulysses cluster.[1] Each of these nodes provide two shared memory sockets with 10 processors each. We measured the constructor time by averaging the results of 100 runs where the threads number has been controlled by setting

```
export OMP_NUM_THREADS=$ii
export OMP_PLACES=cores
export OMP_PROC_BIND=close
```

where `ii` varies in the set $\{1, 2, 4, 8, 16, 20\}$ and where the last two commands control the affinity of the processes spawned.

In Fig. 4.8 we show the speed-up (upper panel) and efficiency (lower panel) of the strong scaling. The dashed line marks perfect linear speed-up in the upper panel, and 100% efficiency in the lower panel. Even though it is far from being perfect, the speed-up shows a constantly increasing trend. The efficiency seams to get constant around the 60% for $p \geq 16$, but a larger parallel system would be necessary for getting a more precise measure.

To conclude, in Fig. 4.9, we show the the weak scaling efficiency case. The thinness, at given processors number $p$, is set to $thin = 50 \cdot p$. As the picture shows, the efficiency seams to become almost constant at $p \gtrsim 8$ for both the C++ and python case, with a value between 70% and 80%.

[1] Please refer to the website for detailed informations.

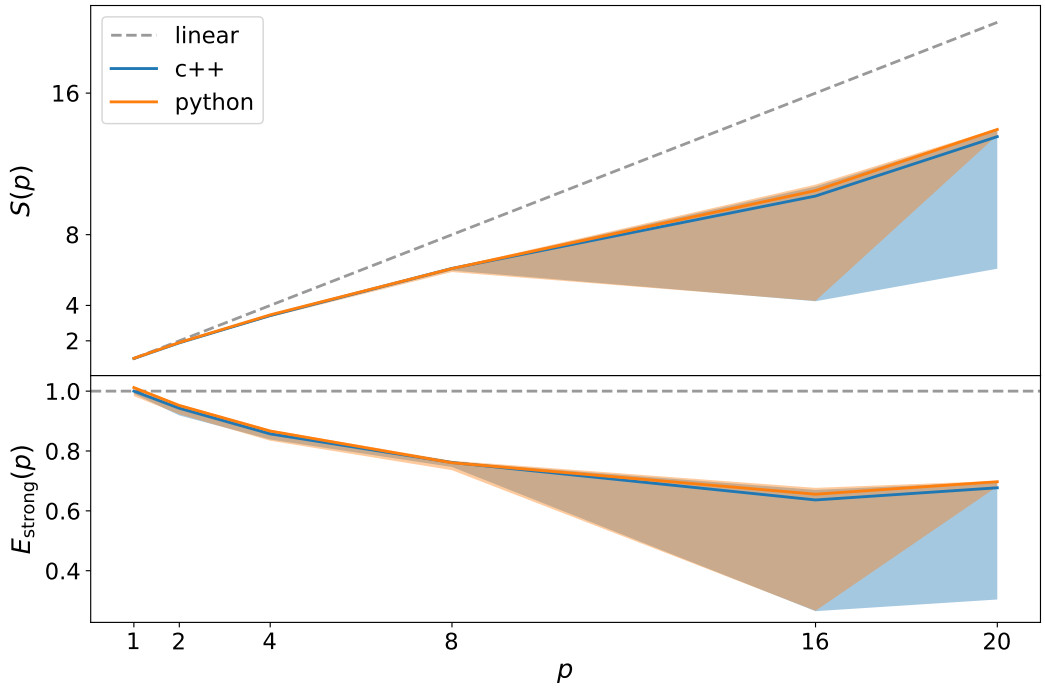


Figure 4.8: Strong-scaling speed-up (*upper panel*) and efficiency (*lower-panel*) of the constructor time at fixed thinness and varying number of multi-threading processors. The solid line marks the average of 100 runs while the shaded region marks the best and worst result area. We run the tests on a full computing-node of the SISSA Ulysses cluster.

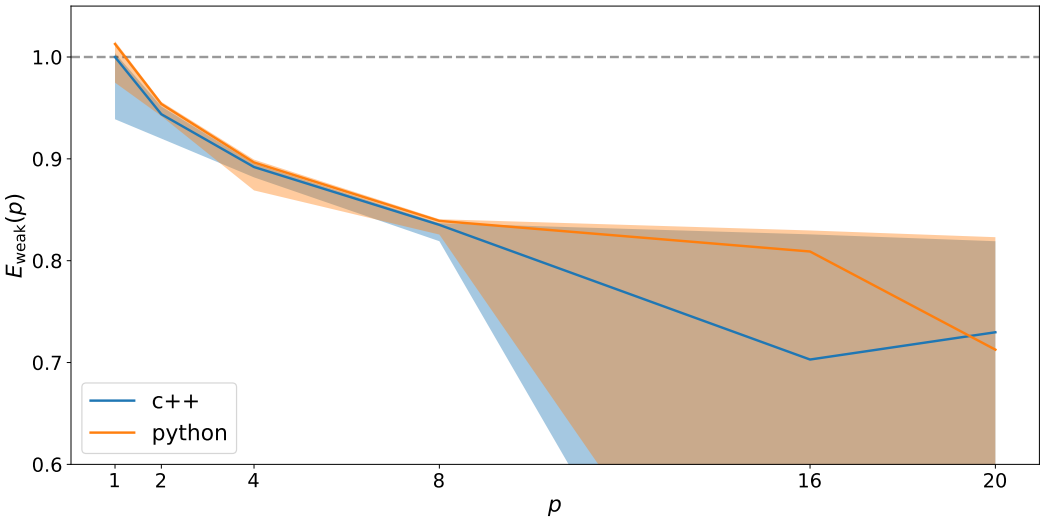


Figure 4.9: Weak-scaling efficiency of the constructor time at thinness growing proportionally to the number of multi-threading processors. The solid line marks the average of 100 runs while the shaded region marks the best and worst result area. We run the tests on a full computing-node of the SISSA Ulysses cluster.

# Chapter 5

# Summary and future developments

One of the most compelling open problems in cosmology is the study of reionization. In this epoch the radiation, produced by the sparse sources that were starting to form, ionized the intergalactic medium, making the Universe transparent. Even though we have proofs that this process took place, we do not know precisely either its physics nor the exact time it should have started. We have some hypothesis though on the nature of the sources that have produced the ionizing radiation. By modelling this populations of astrophysical objects, we could get insights on the process of reionization, study its time evolution and propose hypothesis which could be tested with future experiments.

In this work we have performed an HPC-driven renewal of a scientific application we developed for painting observed populations of objects on top of DM-only N-body cosmological simulations. In the context of reionization, this could help in studying the spatial distribution properties and evolution in time of the ionized bubbles that might have developed around sources of ionizing radiation.

Cosmology is an incremental science: the investigation can always be refined and drive to novel directions. Therefore, while the development of the API was prompted by a specific problem, our design of the restructured code was intended for extensibility. This has been achieved by providing the API with a modular structure and by exploiting Object-Oriented programming, both in C++ and in python. With an extensive usage of polymorphism we obtained a flexible application that can be both used by itself as well as along with other libraries.

The main results of our work are listed below.

- We performed a radical renewal of the original application. We have rewritten most of the functions composing the `cosmology` and the `halo_model` classes. In doing this we have taken care of minimizing both the number of operations and of computationally expensive function calls. Using the CTypes python standard library, we wrapped all the relevant C++ functionalities in python. We have detached from all the external dependencies that were burdening on the build process. while maintaining the possibility to include those functionalities after having built the library. The latter is the combined result of the application of polymorphism on the C++ side and of the implementation of a flexible python interface. The internal

computations have been boosted also by implementing a module for interpolation, which is available in either c++, C and python. Finally, we have completed the integration of all the library functionalities that were missing for completing the algorithm.

- All the development of the API repository has been done with awareness of the best-practices in HPC and, generally, in programming. We have documented the source code and made the documentation accessible by using automated tools designed for this purpose (doxygen and sphinx for translating source code docstrings into html, readthedocs for uploading the html online). The code has been tested during development and the integration of these tests to the API repository is almost complete. We backed-up our GitHub repository with a Continuous Integration tool for getting immediate feedback on the consistency of the library while changes are added. Building the API has been delegated to the Meson build system, which guarantees portability and allows to detach the API from the native environment it has been developed on.

- We have validated all the results returned by the several functions and classes of our library from the scientific perspective. The functions reach the expected accuracy on the cosmological simulations we have tested them. Further investigations with high resolution simulations are still required, and will be completed by the time the scientific paper describing this effort is submitted for publication. Furthermore, we tested the Emcee MCMC sampler [21] for finding the best-fit parameters of the occupation probabilities $N_{\rm cen}$ and $N_{\rm sat}$.

- We have benchmarked the principal components of the API, measuring their performances in different situations. We have tested the scaling at varying work-load of the functions called the most, obtaining satisfying results.

- All these upgrades and additions lead to an outstanding boost in the performances of the application. The computation of a full model, necessary for sampling the parameter space of the occupation probabilities, earned a factor of $10^4$ in execution. From the $\sim 10$ seconds it was spending in our first naive implementation, it now computes in a matter of milliseconds. This also meant not having anymore the necessity of running MCMC chains on distributed memory systems. For regular usage, computations can be easily performed in few minutes from a personal computer.

We will add to the API further miscellaneous functionalities, such as the possibility to download and install it with both `pip` and `conda`, in the next months. We will also complete the online documentation by adding examples and tutorials and extending the explanation of the functionalities provided by ScamPy. When this will be completed we will submit a scientific paper presenting the ScamPy API to the community which will also contain a couple of dummy example applications.

By the time we are writing, we are completing the first scientific application of ScamPy. We are performing a preliminary study on the ionization properties of the

high redshift Universe which result from the injection of ionizing photons from Lyman Break Galaxies (LBG). The dataset we are mocking [17] reaches redshift $z = 7$, which is enough for our preliminary study. We are now able to measure locally on simulations the ionized hydrogen filling factor at different redshifts. This also allows to produce tomographic measures of the ionization state of the medium at varying cosmic time. Furthermore, we can also directly measure the ionized bubble size distribution, which is a quantity that, up to now, has been only modeled indirectly [38].

To conclude, we are planning to extend our work both on the scientific side, by exploring new directions, and on the computational side, by implementing an efficient N-tree algorithm for the optimisation of the neighbor search in both DM-halo catalogues and mock-galaxy catalogues. An incomplete list of the directions we are planning to pursue follows.

- *Extensive application to the reionization problem* - Up to now we have mainly applied ScamPy functionalities to a proof-of-concept framework. Nonetheless, with the dataset at our disposal, an extensive study of the role played by LBG in reionization is possible, provided that larger N-body simulations with high resolution are available.

- *Application of the algorithm to multiple populations* - An application of the same pipeline to the other major players that are known to be involved on the reionization of hydrogen, e.g. AGNs, would be trivial, provided that suitable datasets are available.

- *Extension to different cosmologies* - By now all our investigations have been performed assuming a ΛCDM cosmology. To extend these results to other cosmological models would only require modest modifications of the source code and to obtain the corresponding DM-only N-body simulations.

- *Machine learning extension of the halo occupation model* - Using the halo occupation distribution (HOD) is straightforward and a lot of literature is available on the topic. This approach, though, comes with the limit that all the properties of the observed population have to be inferred from the mass of the host halo. This is known to be a rough approximation. To overcome this limit we plan to use instead a neural network (NN) model of the host halo occupation properties where the inputs of the NN are a set of known features of the halo/sub-halo hierarchy, such as the local environment around the halo and the dispersion velocity within the halo.

# Appendix A

# Source code

We provide here some significant sections of our implementation and refer the user to the GitHub repository (https://github.com/TommasoRonconi/scampy) for deeper source code mining.

## A.1 Generic `cosmo_model` header

The following listing contains the generic header for the definition of a structure of type `cosmo_model` which allows for customization of the kernel cosmological model, as explained in Sec. 2.2.1. It provides the declaration of all the mandatory member functions necessary for ScamPy.

```
/**
 *  @file some_cosmo_model.h
 *  @brief Provides the framework for cosmological computations
 *  @author Tommaso Ronconi
 *  @author tronconi@sissa.it
 */

#ifndef __COSMOLOGY_INTERFACE__
#define __COSMOLOGY_INTERFACE__

/// General namespace of the library
namespace scam {

  struct cosmo_model {

    // Here internal variables of the structure:
    // [ ... ]

    /// function to set internal variables
    void set_internal ( /* list of internal variables that can be set */ );

    /**
     * @name Constructors/Destructors
     * @{
     */

    /// default constructor
```

```
    cosmo_model ();

    /// custom constructor
    cosmo_model ( /* list of variables */ );

    /// default destructor
    virtual ~cosmo_model () = default;

    /// @} End of Ctor/Dtor

    /**
     * @name Cosmographic functions
     * @{
     */

    /// Hubble parameter at given redshift
    double H_z ( const double zz );

    /// Hubble-distance at given redshift
    double d_H ( const double zz );

    /// Comoving distance at given redshift
    double d_C ( const double zz );

    /// Angular diameter distance at given redshift
    double d_A ( const double zz );

    /// comoving volume unit in [ Mpc^3 h^{-3} rad^{-1} ]
    double comoving_volume_unit ( const double zz );

    /// comoving volume in [ Mpc^3 h^{-3} ]
    double comoving_volume ( const double zz );

    /// the age of the Universe at the time the photons were emitted in [ Gyr ]
    double cosmic_time ( const double zz );

    /// @} End of cosmographic functions

    /**
     * @name Matter density related functions
     * @{
     */

    /// critical density in [ Msol Mpc^{-3} h^2 ]
    double rho_crit ( const double zz );

    /// matter density parameter at given redshift
    double OmegaM ( const double zz );

    /// Computes the amplitude of the growing mode D(z) as a function of redshift
    double DD ( const double & zz );

    /// @} End of matter density related functions

    /**
     * @name Power spectrum dependent functions
     * @{
     */

    /// linear matter power spectrum at given redshift
    double Pk ( const double & kk, const double & zz = 1.e-7 );
```

```
    // Mass function
    double dndM ( const double & mm, const double & zz = 1.e-7 );

    // Halo-bias
    double hbias ( const double & mm, const double & zz = 1.e-7 );

    /// @} End of power spectrum dependent functions


  }; //endstruct cosmo_model

} //endnamespace scam

#endif //__COSMOLOGY_INTERFACE__
```

## A.2 Implementation of the interpolation functionalities

We will here show the two most significant header files that allow versatility and flexibility of the interpolation. An object of type `interpolator` can be defined as follows:

```
// for scam::utl::lin_vector()
#include <utilities.h>
// for scam::utl::interpolator<T>
#include <interpolator.h>
// for std::cout
#include <iostream>

using namespace scam::utl;

int main () {

    std::vector< double > xv = lin_vector( 10, 0., 100. );
    std::vector< double > fv = xv;

    interpolator< gsl_lin_interp > interp_func ( xv, fv );

    std::cout << "Value at x = 25:\t" << interp_func( 25. ) << "\n";

    return 0;

}
```

which prints

```
> Value at x = 25: 25.
```

on standard output.

### A.2.1 `interpolation.h` header

This header contains the declaration of the type `interpolator`.

```
/**
 *  @file interpolation.h
```

```
 *  @brief Defines the interpolator< T > type
 *  @author Tommaso Ronconi
 *  @author tronconi@sissa.it
 */

#ifndef __INTERPOLATOR__
#define __INTERPOLATOR__

/// internal includes
#include <utilities.h>
#include <interpolation_interface.h>
#include <gsl_interpolation_interface.h>

/// General namespace of the library
namespace scam {

  /// Utilities namespace
  namespace utl {

    /// Declaration of the interpolator type,
    /// templated on the interpolation_interface type
    /// defaulted to gsl_log_interp
    template< class T = gsl_log_interp >
    class interpolator {

    private:

      /// internal copy of the object of type interpolation_interface
      T _interface;

    public:

      /**
      * @name Constructors/Destructors
      * @{
      */

      /// default constructor
      interpolator () = default;

      /// copy constructor
      interpolator ( const T & interface ) : _interface{ interface } {}

      /// custom constructor for building from function pointer
      interpolator ( std::function< double ( double ) > func,
                              const double x_min, const double x_max,
                              const size_t thinness ) noexcept
                     : _interface{ func, x_min, x_max, thinness } {}

      /// custom constructor for building from gridded values
      interpolator ( const std::vector< double > & xv,
                     const std::vector< double > & fv )
                   : _interface{ xv, fv } {}

      /// default destructor
      ~interpolator () = default;

      /// @} End of Ctor/Dtor

      /**
      * @name  Operator overloads
      * @{
```

```
      */

      /// function-call operator overload
      double operator() ( const double xx ) { return _interface.eval( xx ); }

      /// function for integration
      double integrate ( const double aa, const double bb ) {
            return _interface.integrate( aa, bb );
      }

      /// in-place sum operator overload
      interpolator & operator+= ( const interpolator & rhs ) {
                _interface += rhs._interface;
                return * this;
      }

      /// sum operator overload
      friend interpolator operator+ ( interpolator lhs,
                                      const interpolator & rhs ) {
                lhs += rhs;
                return lhs;
      }

      /// in-place multiplication operator overload
      interpolator & operator*= ( const interpolator & rhs ) {
                _interface *= rhs._interface;
                return * this;
      }

      /// multiplication operator overload
      friend interpolator operator* ( interpolator lhs,
                                      const interpolator & rhs ) {
                lhs *= rhs;
                return lhs;
      }

      /// @} End of operator overloads

      /// Here functions to get private and protected variables
      /// [ ... ]

    }; //endclass interpolator

  } //endnamespace utl

} //endnamespace scam

#endif //__INTERPOLATOR__
```

### A.2.2 `interpolation_interface.h` header

This header provides the base type for interfacing with an interpolation method. The declaration of pure-virtual functions makes the implementation of derived classes mandatory. We have implemented two derived classes based on the `gsl/gsl_interpolation` library in the header

`scampy/interpolation/include/gsl_interpolation_interface.h`

we refer the reader to the GitHub repository (https://github.com/TommasoRonconi/scampy) for further details.

```
/**
 *  @file interpolator_interface.h
 *  @brief Provides the base pure-virtual class for interpolation functionalities
 *  @author Tommaso Ronconi
 *  @author tronconi@sissa.it
 */

#ifndef __INTERPOLATOR_INTERFACE__
#define __INTERPOLATOR_INTERFACE__

#include <utilities.h>

/// General namespace of the library
namespace scam {

  /// Utilities namespace
  namespace utl {

    /// declaration of the interpolator_interface type
    class interpolator_interface {

    private:

      /// lower limit of the domain
      double _x_min;

      /// upper limit of the domain
      double _x_max;

    protected:

      /// size of the interpolation grid
      size_t _thinness;

      /// x-axis values
      std::vector< double > _xv;

      /// y-axis values
      std::vector< double > _fv;

    public:

      /**
      * @name Constructors/Destructors
      * @{
      */

      /// default constructor
      interpolator_interface () = default;

      /// custom constructor
      interpolator_interface ( const double x_min, const double x_max,
                               const size_t thinness )
        : _x_min{ x_min }, _x_max{ x_max }, _thinness{ thinness } {

          _xv.reserve( _thinness );
          _fv.reserve( _thinness );
```

```
      }

    /// move constructor
    interpolator_interface ( interpolator_interface && ii )
      : _x_min{ std::move( ii._x_min ) }, _x_max{ std::move( ii._x_max ) },
        _thinness{ std::move( ii._thinness ) },
        _xv{ std::move( ii._xv ) }, _fv{ std::move( ii._fv ) } {}

    /// copy constructor
    interpolator_interface ( const interpolator_interface & ii )
      : _x_min{ ii._x_min }, _x_max{ ii._x_max }, _thinness{ ii._thinness } {

        _xv = ii._xv;
        _fv = ii._fv;

      }

    /// overload of the swap function for move-semantics
    void swap ( interpolator_interface & ii ) noexcept {

      using std::swap;

      swap( this->_x_min, ii._x_min );
      swap( this->_x_max, ii._x_max );
      swap( this->_thinness, ii._thinness );
      swap( this->_xv, ii._xv );
      swap( this->_fv, ii._fv );

      return;

    }

    /// default destructor
    virtual ~interpolator_interface () = default;

    /// @} End of Ctor/Dtor

    /// pure virtual function, computes y-value at position xx
    virtual double eval ( const double xx ) const = 0;

    /// pure virtual function, computes integral in interval [ aa, bb ]
    virtual double integrate ( const double aa, const double bb ) const = 0;

    /// Here functions to get private and protected variables
    /// [ ... ]

  }; // endclass interpolator_interface

 } // endnamespace utl

} //endnamespace scam

#endif //__INTERPOLATOR_INTERFACE__
```

# Appendix B

# FFTLog patch

In Cosmology, FFTLog [24] is the reference library for 1-dimensional fast-fourier-transform of an array where the values have a logarithmic regular binning. It was written 20 years ago and is based on the algorithm by [39].

It is widely used by the community, and often it is already present on a cosmologist machine. We produced a patch for allowing an easy integration with the Meson build system. The patch has been uploaded on GitHub at the address

https://github.com/TommasoRonconi/fftlog_patch

The repository contains a `readme.rst` and a `fftlog` directory with the `meson.build` script that, if copied in the source code original package, allows to compile it while generating metadata recognizable by `pkgconfig`.

Furthermore, we added a compressed version of the `fftlog` directory. This allows us to treat the external dependency to FFTLog as a subproject of our API. In the `subprojects` directory of our API we have inclueded a `fftlog.wrap` script which contains the following directives:

```
[wrap-file]
lead_directory_missing = fftlog

source_url = https://jila.colorado.edu/~ajsh/FFTLog/fftlog.tgz
source_filename = fftlog.tgz
source_hash = [ ... ]

patch_url = https://github.com/TommasoRonconi/fftlog_patch/
                                                raw/master/fftlog_patch.zip
patch_filename = fftlog_patch.zip
patch_hash = [ ... ]
```

If the FFTLog shared object is not found on the system, the Meson build system will automatically download the original source code and copy it in the

`scampy/subprojects/fftlog`

directory. Then it will download and copy within the same directory our patch, making it possible to compile and install the shared object along with the other binaries produced by the build system.

This not only simplifies the build on local machines, but also allows for the usage on remote servers with automated build functionalities, such as Travis CI.

# Bibliography


[1] Planck Collaboration VI. Planck 2018 results. VI. Cosmological parameters. *arXiv e-prints*, page arXiv:1807.06209, Jul 2018.

[2] James E. Gunn and Bruce A. Peterson. On the Density of Neutral Hydrogen in Intergalactic Space. *ApJ*, 142:1633–1636, Nov 1965.

[3] Robert H. Becker, Xiaohui Fan, Richard L. White, Michael A. Strauss, Vijay K. Narayanan, Robert H. Lupton, James E. Gunn, James Annis, Neta A. Bahcall, J. Brinkmann, A. J. Connolly, István Csabai, Paul C. Czarapata, Mamoru Doi, Timothy M. Heckman, G. S. Hennessy, Željko Ivezić, G. R. Knapp, Don Q. Lamb, Timothy A. McKay, Jeffrey A. Munn, Thomas Nash, Robert Nichol, Jeffrey R. Pier, Gordon T. Richards, Donald P. Schneider, Chris Stoughton, Alexand er S. Szalay, Aniruddha R. Thakar, and D. G. York. Evidence for Reionization at z~6: Detection of a Gunn-Peterson Trough in a z=6.28 Quasar. *AJ*, 122(6):2850–2857, Dec 2001.

[4] Hong Guo, Zheng Zheng, Peter S. Behroozi, Idit Zehavi, Chia-Hsun Chuang, Johan Comparat, Ginevra Favole, Stefan Gottloeber, Anatoly Klypin, Francisco Prada, Sergio A. Rodríguez-Torres, David H. Weinberg, and Gustavo Yepes. Modelling galaxy clustering: halo occupation distribution versus subhalo matching. *MNRAS*, 459(3):3040–3058, Jul 2016.

[5] Asantha Cooray and Ravi Sheth. Halo models of large scale structure. *Phys. Rep.*, 372(1):1–129, Dec 2002.

[6] Idit Zehavi, Zheng Zheng, David H. Weinberg, Joshua A. Frieman, Andreas A. Berlind, Michael R. Blanton, Roman Scoccimarro, Ravi K. Sheth, Michael A. Strauss, Issha Kayo, Yasushi Suto, Masataka Fukugita, Osamu Nakamura, Neta A. Bahcall, Jon Brinkmann, James E. Gunn, Greg S. Hennessy, Željko Ivezić, Gillian R. Knapp, Jon Loveday, Avery Meiksin, David J. Schlegel, Donald P. Schneider, Istvan Szapudi, Max Tegmark, Michael S. Vogeley, and Donald G. York and. The luminosity and color dependence of the galaxy correlation function. *The Astrophysical Journal*, 630(1):1–27, sep 2005.

[7] Zheng Zheng, Alison L. Coil, and Idit Zehavi. Galaxy evolution from halo occupation distribution modeling of DEEP2 and SDSS galaxy clustering. *The Astrophysical Journal*, 667(2):760–779, oct 2007.

[8] Frank C. van den Bosch, Surhud More, Marcello Cacciato, Houjun Mo, and Xiaohu Yang. Cosmological constraints from a combination of galaxy clustering and lensing – I. Theoretical framework. *Monthly Notices of the Royal Astronomical Society*, 430(2):725–746, 02 2013.

[9] Surhud More, Hironao Miyatake, Rachel Mandelbaum, Masahiro Takada, David. N. Spergel, Joel R. Brownstein, and Donald P. Schneider. The weak lensing signal and the clustering of boss galaxies. ii. astrophysical and cosmological constraints. *The Astrophysical Journal*, 806(1):2, jun 2015.

[10] James S. Bullock, Risa H. Wechsler, and Rachel S. Somerville. Galaxy halo occupation at high redshift. *Monthly Notices of the Royal Astronomical Society*, 329(1):246–256, 01 2002.

[11] Takashi Hamana, Masami Ouchi, Kazuhiro Shimasaku, Issha Kayo, and Yasushi Suto. Properties of host haloes of Lyman-break galaxies and Lyman $\alpha$ emitters from their number densities and angular clustering. *Monthly Notices of the Royal Astronomical Society*, 347(3):813–823, 01 2004.

[12] Masami Ouchi, Takashi Hamana, Kazuhiro Shimasaku, Toru Yamada, Masayuki Akiyama, Nobunari Kashikawa, Makiko Yoshida, Kentaro Aoki, Masanori Iye, Tomoki Saito, Toshiyuki Sasaki, Chris Simpson, and Michitoshi Yoshida. Definitive identification of the transition between small- and large-scale clustering for lyman break galaxies. *The Astrophysical Journal*, 635(2):L117–L120, dec 2005.

[13] Kyoung-Soo Lee, Mauro Giavalisco, Oleg Y. Gnedin, Rachel S. Somerville, Henry C. Ferguson, Mark Dickinson, and Masami Ouchi. The large-scale and small-scale clustering of lyman break galaxies at $3.5 \leq z \leq 5.5$ from the GOODS survey. *The Astrophysical Journal*, 642(1):63–80, may 2006.

[14] Kyoung-Soo Lee, Mauro Giavalisco, Charlie Conroy, Risa H. Wechsler, Henry C. Ferguson, Rachel S. Somerville, Mark E. Dickinson, and Claudia M. Urry. Mapping the dark matter from uv light at high redshift: an empirical approach to understand galaxy statistics. *The Astrophysical Journal*, 695(1):368–390, mar 2009.

[15] Hildebrandt, H., Pielorz, J., Erben, T., Schneider, P., Eifler, T., Simon, P., and Dietrich, J. P. Gabods: the garching-bonn deep survey * - viii. lyman-break galaxies in the eso deep public survey. *A&A*, 462(3):865–873, 2007.

[16] Fuyan Bian, Xiaohui Fan, Linhua Jiang, Ian McGreer, Arjun Dey, Richard F. Green, Roberto Maiolino, Fabian Walter, Kyoung-Soo Lee, and Romeel Davé. THE LBT BOÖTES FIELD SURVEY. i. THE REST-FRAME ULTRAVIOLET AND NEAR-INFRARED LUMINOSITY FUNCTIONS AND CLUSTERING OF BRIGHT LYMAN BREAK GALAXIES ATZ$\sim$ 3. *The Astrophysical Journal*, 774(1):28, aug 2013.

[17] Yuichi Harikane, Masami Ouchi, Yoshiaki Ono, Surhud More, Shun Saito, Yen-Ting Lin, Jean Coupon, Kazuhiro Shimasaku, Takatoshi Shibuya, Paul A. Price, Lihwai Lin, Bau-Ching Hsieh, Masafumi Ishigaki, Yutaka Komiyama, John Silverman, Tadafumi Takata, Hiroko Tamazawa, and Jun Toshikawa. Evolution of Stellar-to-Halo Mass Ratio at z = 0 - 7 Identified by Clustering Analysis with the Hubble Legacy Imaging and Early Subaru/Hyper Suprime-Cam Survey Data. *ApJ*, 821(2):123, Apr 2016.

[18] D. Nelson Limber. The Analysis of Counts of the Extragalactic Nebulae in Terms of a Fluctuating Density Field. *ApJ*, 117:134, Jan 1953.

[19] OpenMP Architecture Review Board. *OpenMP Application Program Interface, version 4.0*. OpenMP Architecture Review Board, July 2013.

[20] M. Galassi et al. *GNU Scientific Library Reference Manual - Third Edition*, January 2009.

[21] Daniel Foreman-Mackey, David W. Hogg, Dustin Lang, and Jonathan Goodman. emcee: The MCMC Hammer. *PASP*, 125(925):306, Mar 2013.

[22] F. Marulli, A. Veropalumbo, and M. Moresco. CosmoBolognaLib: C++ libraries for cosmological calculations. *Astronomy and Computing*, 14:35–42, Jan 2016.

[23] Jeremy L. Tinker and Andrew R. Wetzel. What does Clustering Tell us About the Buildup of the Red Sequence? *ApJ*, 719(1):88–103, Aug 2010.

[24] A. J. S. Hamilton. Uncorrelated modes of the non-linear power spectrum. *MNRAS*, 312(2):257–284, Feb 2000.

[25] The meson build system. https://mesonbuild.com/index.html.

[26] Cmake reference website. https://cmake.org/.

[27] Ninja reference website. https://ninja-build.org/.

[28] Microsoft build engine. https://docs.microsoft.com/it-it/visualstudio/msbuild/msbuild?view=vs-2019.

[29] Xcode 11 reference website. https://developer.apple.com/xcode/.

[30] Travis ci reference guide. https://docs.travis-ci.com/.

[31] Dimitri van Heesch. Doxygen reference website. http://www.doxygen.nl/.

[32] Georg Brandl and the Sphinx team. Sphinx, python documentation generator. https://www.sphinx-doc.org/en/master/.

[33] Read the Docs et al. Read the docs reference website. https://readthedocs.org/.

[34] Stephen D. Landy and Alexander S. Szalay. Bias and Variance of Angular Correlation Functions. *ApJ*, 412:64, Jul 1993.

[35] Gillian D. Beltz-Mohrmann, Andreas A. Berlind, and Adam O. Szewciw. Testing the Accuracy of Halo Occupation Distribution Modelling using Hydrodynamic Simulations. *arXiv e-prints*, page arXiv:1908.11448, Aug 2019.

[36] Boryana Hadzhiyska, Sownak Bose, Daniel Eisenstein, Lars Hernquist, and David N. Spergel. Limitations to the "basic" HOD model and beyond. *arXiv e-prints*, page arXiv:1911.02610, Nov 2019.

[37] J. González-Nuevo, A. Lapi, L. Bonavera, L. Danese, G. de Zotti, M. Negrello, N. Bourne, A. Cooray, L. Dunne, S. Dye, S. Eales, C. Furlanetto, R. J. Ivison, J. Loveday, S. Maddox, M. W. L. Smith, and E. Valiante. H-ATLAS/GAMA: magnification bias tomography. Astrophysical constraints above ~1 arcmin. *J. Cosmology Astropart. Phys.*, 2017(10):024, Oct 2017.

[38] A. Roy, A. Lapi, D. Spergel, and C. Baccigalupi. Observing patchy reionization with future CMB polarization experiments. *J. Cosmology Astropart. Phys.*, 2018(5):014, May 2018.

[39] J. D. Talman. NumSBT: A subroutine for calculating spherical Bessel transforms numerically. *Computer Physics Communications*, 180(2):332–338, Feb 2009.