



GalaPy—Implementation strategies of the spectral modelling tool for galaxies in Python[☆]

Tommaso Ronconi^{ID*}, Andrea Lapi

INAF-IRA, Via Pietro Gobetti, 101, Bologna, 40129, Italy

ICSC - Centro Nazionale di Ricerca in High Performance Computing, Big Data e Quantum Computing, Via Magnanelli 2, Bologna, Italy

SISSA, Via Bonomea, 265, Trieste, 34136, Italy

ARTICLE INFO

Keywords:

Galaxy Formation and Evolution
Bayesian sampling
Multi-wavelength emission
Software architecture

ABSTRACT

We present the computational design and implementation of GalaPy, a hybrid C++/Python library for the spectral energy distribution (SED) modelling of galaxies. Originally introduced in Ronconi et al. (2024), GalaPy has been developed within the Italian galaxy formation and cosmology community as part of the ICSC – Centro Nazionale di Ricerca in High Performance Computing, Big Data e Quantum Computing. The library combines the performance of compiled C++ routines with the flexibility of Python, enabling efficient generation and fitting of physically motivated SED models.

We describe the object-oriented architecture of the code, its hybrid parallelisation strategy, and the optimisations that ensure portability and minimal memory overhead. Parallel execution relies on a combination of vectorised array programming, shared-memory concurrency, and distributed-memory message passing.

Recent updates include Bayesian evidence-based model selection and a fully analytical, panchromatic active galactic nucleus component. These additions further improve the physical realism and the statistical power of the framework. GalaPy thus provides a modular and extensible platform for galaxy modelling, designed to interface and adapt seamlessly to the next generation of large-scale astrophysical analyses.

1. Introduction

Modern astrophysical research is entering an era in which the availability of data, rather than their scarcity, sets the main challenge. Current and upcoming facilities such as *JWST*, *Euclid*, the *Vera C. Rubin Observatory*, and the *Square Kilometre Array* will produce datasets of unprecedented size and complexity, spanning from the X-ray to the radio domain. Efficiently handling, analysing, and interpreting these massive data volumes requires advanced computational frameworks capable of exploiting the full potential of modern High-Performance Computing (HPC) architectures and, increasingly, Machine Learning (ML) techniques. In this context, the Exascale era poses new requirements on code design, including efficient memory management, hybrid parallelisation, and portability across heterogeneous computing platforms.

Within this rapidly evolving computational landscape, spectral modelling remains a cornerstone for interpreting the physical properties of galaxies. The spectral energy distribution (SED) encodes information about a galaxy's star formation history, chemical enrichment, and

dust content, yet modelling such complexity at scale remains computationally demanding. Most existing SED-fitting tools were originally developed for low-redshift and UV/optically bright sources and rely on empirical templates that limit physical interpretability and scalability (e.g. Da Cunha et al., 2008; Battisti et al., 2019; Boquien et al., 2019; Johnson et al., 2021).

To address these issues, we developed GalaPy, a hybrid C++/Python library designed for high-performance SED modelling of galaxies (Ronconi et al., 2024). The project originates in the context of the GOTH¹ (Galaxy Observational and Theoretical Astrophysics Group, SISSA, IFPU, INAF/IRA) and is part of the activities of the ICSC – Centro Nazionale di Ricerca in High Performance Computing, Big Data e Quantum Computing. GalaPy couples a physically motivated, modular modelling framework with an optimised compiled backend, enabling the generation of complex SEDs on-the-fly with minimal memory footprint. It supports fully Bayesian parameter inference and interfaces seamlessly with widely used sampling libraries such as *emcee* (Foreman-Mackey et al., 2013) and *dynesty* (Speagle, 2020).

[☆] This article is part of a Special issue entitled: 'HPC in Cosmology and Astrophysics' published in Astronomy and Computing.

* Corresponding author.

E-mail address: tommaso.ronconi@inaf.it (T. Ronconi).

¹ gotha.com.

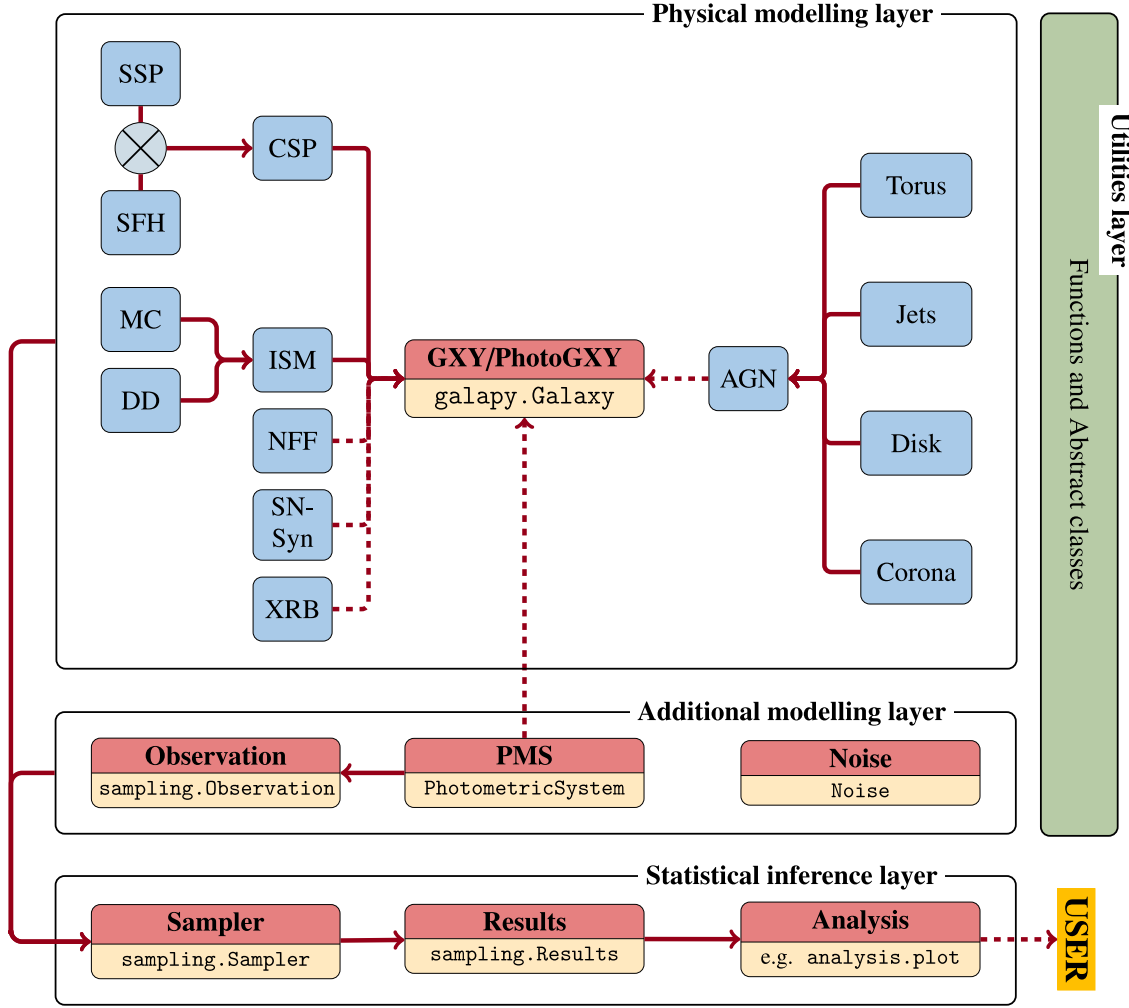


Fig. 1. The flux-diagram schematically depicts the dependency tree of python objects in the GalaPy library. We highlight different purpose and accessibility level for users with different colours. Blue blocks contain python objects which can be accessed by users but are not recommended for every-day usage. Red/orange splitted blocks contain the objects which are intended as a user-interface to internal functionalities (and their position in terms of modules/sub-packages within the orange lower part). The green box marks support functions and abstract classes which are used for the lower levels of library implementation, user access to this layer is recommended mainly for customisation. All the acronyms in the flux-diagram refer to the physical components implemented in the model. A complete explanation can be found in [Ronconi et al. \(2024\)](#) and in Section 2.1, here we just expand all of them for completeness: SSP = Simple Stellar Population; SFH = Star Formation History; CSP = Composite Stellar Population; MC = Molecular Cloud; DD = Diffuse Dust; ISM = Inter-Stellar Medium; NFF = Nebular Free-Free; SN-Syn = Super-Nova Synchrotron; XRB = X-ray Binaries; AGN = Active Galactic Nucleus; GXY = Galaxy; PhotoGXY = Photometric Galaxy; PMS = Photometric System.

This contribution focuses on the computational aspects of GalaPy, detailing its internal structure and object-oriented implementation (Section 2), parallelisation strategy, performance characteristics (Section 3) and what will be added in the forthcoming updates of the library (Section 4). We discuss how the design philosophy, originally driven by the physical modelling of galaxy evolution, has evolved to meet the practical constraints imposed by Python-based scientific ecosystems. Finally, we outline the limitations currently imposed by Python's parallel execution model and identify possible future directions for tighter C++/Python coupling and improved scaling on HPC infrastructures (Sections 3 and 5).

2. Object oriented implementation

GalaPy has a hybrid implementation which allows us to exploit both the performance efficiency of a compiled language (C++) as well as the flexibility of an interpreted language (Python). The Bushido of GalaPy software development can be summarised in few points:

- The computational core, written in Object-Oriented C++, handles all modelling tasks involving the most demanding mathematical operations. Physical components described in [Ronconi et al. \(2024\)](#) are implemented as independent classes sharing a common interface for parameter configuration and emission computation as a function of wavelength, age, and metallicity. During construction, quantities independent of model parameters are pre-computed and cached, reducing runtime operations while keeping the RAM footprint minimal². This design balances the speed of SED grid interpolation with the flexibility of on-the-fly computation. All objects are serialisable and fully Python-compliant via pickling. In addition to physical models, the compiled layer provides optimised data structures, interpolation routines based

² The size of the available SSPs vary from ~15 to ~80 MB and have therefore a size negligible with respect of typical RAM sizes. On recent machines with a L3 cache of some tens of MB could also allow to load some of the SSP libraries completely on the registers.

on irregular interval binary search tree algorithms (further commented in Ronconi et al., 2024), and efficient loading of SSP libraries to improve CPU cache usage. The implementation relies solely on the C++ STL, tested on both Linux and Unix operative systems, ensuring portability and easy installation.

- **Python** deals with the interplay between all the components and modules, internal and external, that build up the library. It also provides the user-interface and an extensive documentation. Lastly, the terminal commands allowing for quick-access parameterised SED-fitting that come out-of-the-box with library installation (e.g. the `galapy-fit` command) are implemented as Python entry-points. By importing the `galapy` package and sub-packages the GalaPy API is exposed, allowing for complete customisation of the algorithms as well as providing the tools for astrophysical modelling and analysis of the sampling results.
- **pybind11** is a library to generate Python bindings of compiled C++ code. We bind compiled classes and expose our optimised C++ implementation to the Python interface providing access to our functions to users. All the functions that can be applied to arrays of values are vectorised, providing a straightforward integration with the most common python packages for scientific computing (e.g. NumPy and SciPy) and therefore allow for array programming. We have chosen this strategy because, compared to a CPython wrapping layer, it delivers bindings with negligible latency while providing a more intuitive interface.

2.1. Structure of the library

A complete description of the public classes and functions implemented in the GalaPy package is available in the on-line documentation, in the section Python API.³ We provide in Fig. 1 a flowchart schematising the main python objects and sub-package inter-dependence in the library. The library can be seen as divided into 4 main conceptual blocks, the physical modelling layer (implementing the astrophysical components of the model), an additional modelling layer taking care of the observers limitations (i.e. noisy data, photometric system and data points with errors), a sampling layer which wraps the external samplers and exposes the terminal entry-points for fast access and an utilities layer where analysis and support functions are implemented.

We hereby provide just a short description of the package structure and of the functionalities provided by each module/sub-package. GalaPy contains modules in the top-level package and on sub-packages as well, divided as follows

```
galapy
|-- galapy.analysis
|-- galapy.configuration
|-- galapy.internal
|-- galapy.io
'-- galapy.sampling
```

The content of each sub-package provides different functionalities:

- **galapy**: the root package contains all the modules providing access to the models described in Ronconi et al. (2024). Specifically:
 - `galapy.StarFormationHistory`: contains the class SFH that can be used to build either empirical, In-Situ or interpolated SFH models.
 - `galapy.CompositeStellarPopulation`: contains functions for listing and loading SSP libraries and the CSP class, used to build composite stellar populations.
- `galapy.InterStellarMedium`: provides access to the dust-model. It defines several objects: a base `ismPhase` class, from which two derived classes inherit, MC and DD, modelling the attenuation and emission due to the two separate dust components; additionally, a ISM type is defined, wrapping the other two components and combining their contributions.
- `galapy.NebularFreeFree`, `galapy.Synchrotron` and `galapy.XRayBinaries`: these modules implement (optional) the additional sources of stellar continuum. They respectively define the classes NFF, SNSYN and XRB.
- `galapy.ActiveGalacticNucleus`: provides a common interface for modelling the AGN by selecting one of the two models available in the library: either loading the (Fritz et al., 2006) templates and consistently adding on top of them an eventual X-ray contribution, or by using the new analytical model mentioned in previous Sections. These functionalities are accessed through the class AGN, defined in this module.
- `galapy.Cosmology` and `galapy.InterGalacticMedium`, define respectively the classes CSM and IGM, whose implementation provide access to astrometric functions and the inter-galactic hydrogen attenuation model, respectively.
- `galapy.Galaxy` and `galapy.Handlers` define utility classes and methods designed to ease modelling through the Python API. In particular, the former defines the class GXY which wraps up the models implemented in the other modules, their interplay and parameter settings, optimising the performances through a minimisation of the number of operations. By instantiating one of this objects, i.e.

```
from galapy.Galaxy import GXY
gxy = GXY( age = 1.e9, redshift = 1.0 )
```

users can easily modify the value of the free-parameters (e.g. `gxy.set_parameters(age = 1.e10)`), get the emission or flux (e.g. `flux = gxy.SED()`), or compute derived parameters (e.g. `Mstar = gxy.sfh.Mstar(1.e8)`, for the stellar mass at an age of $\tau = 10^8$ years). Note that all these functionalities are obtained by a combination of tools implemented in the modules listed above. The latter, `galapy.Handlers` module, is designed for managing the free-parameters when sampling.

- `galapy.PhotometricSystem`: implements the class PMS that can be used to manage band-pass transmission filters, both loaded from the database or user-defined.
- **galapy.sampling**: contains sub-modules used for sampling the parameter space, i.e. the two sub-modules `Sampler` and `Results` which unify the interface to the different sampling algorithms implemented in the library along with their results note in particular, the `Results` class, the `Observation` module which collects observational datasets and the `Statistics` sub-module, defining statistical functions such as likelihoods and estimators.
- **galapy.analysis**: provides the two sub-modules `funcs` and `plot`, both defining functions that facilitate the analysis of sampling results. While the former mainly produces tables with the estimates of several statistics, the latter produces plots of fitted SEDs, residuals and posteriors.
- **galapy.io**: used to load and store object types defined in the package.
- **galapy.configuration** and **galapy.internal**: are mainly for internal usage, even though some classes and functions of `galapy.internal` might be useful in some parts of the analysis.

³ galapy.readthedocs.io/en/latest/python_doc.

3. Parallelisation strategy

The primary purpose of GalaPy is to derive the parameters that can be inferred from the spectro-photometric properties of galaxies. Our code aims at delivering a high performance serial implementation of parametric SEDs, so that parallelism is not necessary in model generation. In this way, the only bottleneck of the work-flow is parameter-space sampling.

Both `emcee` and `dynesty` allow for passing a pool of workers to the functions running the sampling. In GalaPy we generate pools exploiting the `multiprocessing` package of the Python standard library. Because of the structural limits of Python (i.e. the existence of a Global Interpreter Lock that guarantees parallel threads are not modifying concurrently the reference count in the Python interpreter), allocating a pool of parallel workers, with the intent of speeding-up CPU-bound workflows, requires to generate a copy of the environment. Copying the whole environment, though, results in the necessity of generating deep copies of all the variables that can be referenced in a given scope. This not only means a larger memory usage, but it also reduces the effectiveness of shared memory parallelism, as passing around chunks of memory severely slows down the computation.

In the entry-point provided for fitting SEDs (i.e. `galapy-fit`) the default behaviour tries to reach a compromise between memory usage and parallelism. The variables that require the larger memory budget (e.g. the SSP libraries and the parametric models) are made global and therefore accessible for all the workers in the pool. In the meantime, we spawn as many workers as possible to squeeze all the computing power from the architecture.

As of version `v0.6.0` (see Section 4), the code implements a hybrid parallelisation strategy that combines vectorised execution, shared-memory parallelism, and distributed computing. The general design is constrained by the necessity of maintaining compatibility with external Python-based sampling libraries, which restricts the use of fully compiled C++ parallel backends. Despite these limitations, the structure allows flexible scaling depending on the computational task:

- **Single-object, single-model mode:** In this configuration, all available CPU resources are used to accelerate the computation of a single model instance. Super-scalar execution is achieved through extensive NumPy/SciPy array programming, which automatically exploits optimised BLAS/MKL/OpenMP routines. Shared-memory parallelism further enhances performance via:
 - Native multithreading in NumPy, SciPy, and linked numerical libraries;
 - User-defined `multiprocessing.Pool` objects, which allow concurrent likelihood evaluations within the sampling process.

This setup maximises interoperability and portability but is limited by Python's runtime overhead and the global interpreter lock (GIL). Performance profiling through UNIX system monitors indicates that the effective CPU usage rarely exceeds 20 to 40%, suggesting that the Python layer constitutes the primary bottleneck.

Furthermore, we have measured the strong scaling of sampling via Python `multiprocessing` which is shown in Fig. 2. Strong scaling measures how much faster a program runs when adding more processors for a fixed problem size. Speed-up, $S(p)$, is defined as the ratio between the time taken by execution on a single processor, $p = 1$, and the time taken by executing the same program on p resources (in Fig. 2 this is given by the number of CPUs, N_{CPU}):

$$S(p) = t(1)/t(p). \quad (1)$$

The measurement has been performed on a single compute node of the SISSA Ulysses Cluster (Model: IBM System x(R) iDataPlex(R) dx360 M4 Server), which hosts 2 Intel(R) Xeon(R) E5-2680 v2@2.80 GHz CPUs, with 10 physical cores each. As clearly shown by the picture, the Python implementation reaches a limited speed-up by increasing the number of processors, settling around a plateau at around $S \approx 2.5$, way below the optimal scaling, given by the Amdahl's law (black dashed line).

- **Multiple-object and/or multiple-model mode:** In catalogue applications or multi-model analyses, parallelism is achieved by distributing different objects or model instances across processes or nodes. Two strategies are supported:
 - *Multi-threading*, for shared-memory execution on a single machine;
 - *Message passing*, through MPI interfaces, enabling hybrid schemes where distributed processes (different objects/models) each retain internal shared-memory parallelism as described in the previous case.

This approach offers a flexible compromise between ease of implementation and scalability, though the current Python-based coordination layer still limits full exploitation of the underlying hardware.

To quantify the impact of our implementation strategy for the multiple-model catalogue execution mode, we measure the efficiency of our new MPI implementation. Efficiency, $E(N)$, measures how performance changes when you increase the problem size proportionally with the number of processors. This measure is given in terms of the ratio between the wall-clock time taken for running the program on a problem of size 1 with 1 process and the wall-clock time necessary for running on a problem of size N with N processes:

$$E(N) = t_1(1)/t_N(N). \quad (2)$$

In our case, the problem size is given by the number of either objects or models for which to perform the sampling. Without any parallelisation strategy, this would require to run the sampler N independent times, with N equal to the number of objects or models. Even though this action is possible, it becomes unfeasible when the problem size grows substantially, for instance, in the case of large catalogues of objects (i.e. $N \gtrsim 100$).

To perform this measurement we set as unit worker size of 5 cores bound to a single socket of the same compute nodes used for running the strong scaling measurement described above. This set-up guarantees the possibility of running 4 MPI processes per each of the available compute nodes on a single core, leaving therefore 4 idle cores. The queues available in Ulysses allows for the usage of up to 16 nodes at a time, thus the test has been run for $N_{\text{max}} = 64$. The mean weak scaling efficiency is shown in Fig. 3 as a blue solid line while the shaded blue region marks the $2\text{-}\sigma$ confidence around the mean. We also mark as a black dashed line, the perfect efficiency given by the Gustafson's Law. The parallelisation settles around an efficiency of about 50% (the value is marked by the grey dotted line in Figure) which, even though not optimal, is stable for increasing values of N .

Considering that, as just described, the exposed set-up is overabundant for serial execution, it is also possible to run hybridly, using the idle cores for shared memory parallelism. This is an optimal set-up, as per the strong-scaling speed-up plot of Fig. 2, the performances are still in the increasing part of the blue curve, guaranteeing for a 2x speed-up which compensates for the decrease of the efficiency in weak scaling.

Overall, we highlight acceptable, even though not optimal, performances where. The observed loss is most likely due to the Python sector and algorithmic limitations due to on the one side to the necessity of pickling/un-pickling complex objects which also link to compiled, non-Python code and on the other side, to limitations on the multi-threading

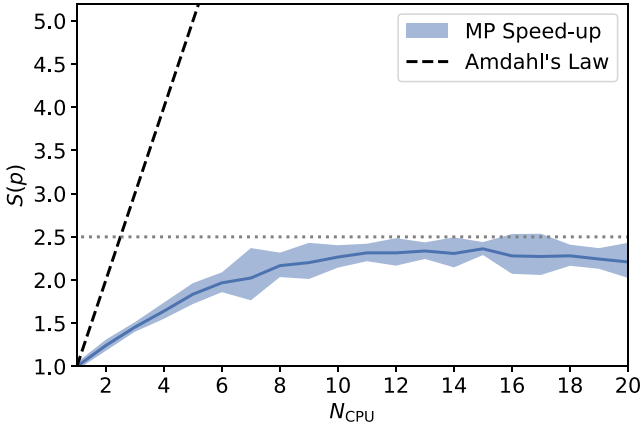


Fig. 2. Strong-scaling of the `galaxy-fit` command, used for automatically sampling the posterior probability with either of the samplers available in the library. The blue line with $3\text{-}\sigma$ shaded region highlights the speed-up compared with perfect scaling (Amdahl's Law marked by a dashed black line). The test has been run on a single IBM System x(R) iDataPlex(R) dx360 M4 Server compute node of SISSA's Ulysses Cluster with hyper-threading disabled. Each of these nodes hosts 2 Intel(R) Xeon(R) E5-2680 v2@2.80 GHz, with 10 physical CPUs each.

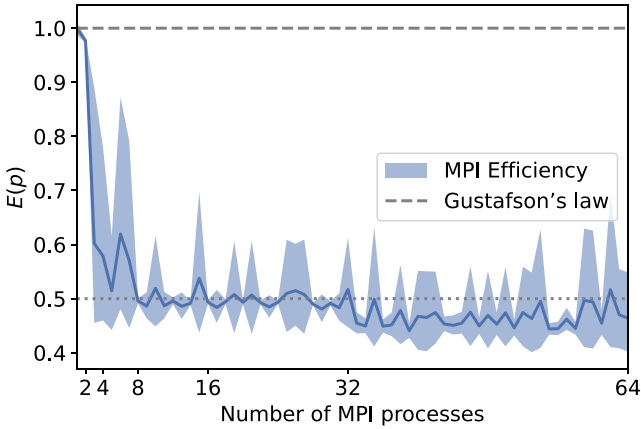


Fig. 3. Weak scaling of the MPI implementation of the new `galaxy-fit` command, available from v0.6.0, run on a catalogue application. The blue line with $2\text{-}\sigma$ shaded region highlight the mean efficiency compared with perfect efficiency (Gustafson's Law, marked by a dashed black line). We have run the test on the same machine and on the same model of compute nodes as per Fig. 2, with the only difference that, instead of increasing the number of processes at fixed problem size, we have reserved 5 cores per each object in our catalogue of 64 simulated galaxies. The resources have been allocated by guaranteeing binding to the same socket, in order to minimise memory movement across the two portion of shared memory available in each node.

inefficiency of the external libraries used for sampling. Future developments will explore tighter C++/Python coupling and more efficient task scheduling to overcome these inherent limitations. We plan to investigate primarily on the potential of a pure Python implementation in solving the pickling/un-pickling shortcomings and, alternatively on extending the C++ sector, by making use of a compiled-language for sampling (e.g. Polychord Handley et al., 2015, or similar libraries). Furthermore, from the sampler side, given the poor strong scaling is a known issue of both the libraries used in GalaPy, we also plan to test usage of the Nautilus sampler (Lange, 2023) which, even though also being a pure-python package, is known to accelerate considerably the sampling, when accurately tuned, thanks to the combination of importance sampling and machine learning assisted prior volume exploration.

4. Latest updates

The following updates to the library⁴ will be presented in a separate work focusing on the scientific and astrophysical aspects, which go beyond the scope of this paper.

The main forthcoming upgrades to GalaPy include:

- **Bayesian evidence-based model selection** (from v0.6.0): in this release, we introduce an automated framework of functions and handlers that allows the user to compare different models and identify the most reliable one based on the Bayes factor computed from the posterior probability distribution. With the functionality supported for posteriors derived through nested sampling (i.e. evidence-based) algorithms, since evidence estimation from standard MCMC methods is generally less robust (Allison and Dunkley, 2013; Speagle, 2019; Higson et al., 2019). With this version, the hybrid parallelisation described in Section 3 is also made public. The new version does not change the user interface to the command-line directives necessary to run a sampling. The parallel run is instead triggered by passing, through the parameter file, arrays of models (e.g. with/without AGN component or with different SFH models) and/or arrays of photometric datasets (e.g. as in a homogeneous catalogue of sources).
- **Panchromatic active galactic nucleus component** (from v0.7.0): we have implemented a fully analytical, unified model of Active Galactic Nucleus (AGN) emission. The necessity for this extension arises from the wish of having an AGN modelling consistent with the analytical modelling-driven implementation of the rest of the emission. In our previous implementation, we made use of the templated models from Fritz et al. (2006), the user had the possibility to choose among a library of 24'000 pre-computed normalised emission shapes, the sampler would then tune just the normalisation of these templates, through a free f_{AGN} given in units of the total infrared transmitted power. Even though functional and exploited in several other libraries, this strategy is not optimal for continuous posterior space sampling, which instead is more efficient when dealing with fully parameterised models. In Fig. 4 we provide a schematic representation of the model and its components:

- Accretion disc emission, allowing for anisotropic geometries;
- Obscuration by a dusty torus and polar dust, adopting a Small Magellanic Cloud-like extinction curve;
- absorption and re-emission from the dusty torus, computed through a time-dependent energy-conservation algorithm;
- X-ray emission from the hot corona, with multiple prescriptions available (bolometric correction, mid-IR to X-ray relations, or α_{OX} -X-ray slope correlations);
- Radio emission from jets, core and lobes, with optional anisotropy and relativistic beaming.

Each of these components is implemented as an independent Python object (see Fig. 1) and collectively introduces seven additional free parameters to the model framework that are handled by the wrapper class AGN. Even though present, the Object Oriented implementation of the AGN model is not directly exposed to the user, which only interacts with the wrapper class. Additionally, the AGN class regulates the interplay between the free parameters defining the different components of the physical model listed above. Some examples include the total radiated energy coming from the disc and the free-parameters which determine the torus opening angle which both regulate what part

⁴ At the time of writing, the current stable version is v0.5.3.

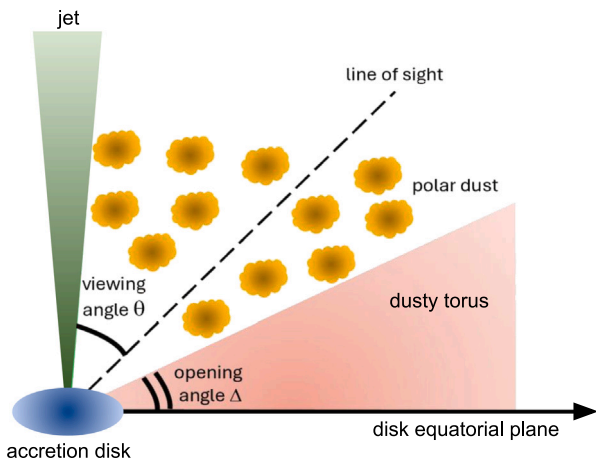


Fig. 4. One quadrant (schematic) of the unified AGN model as implemented in v0.7.0 of the GalaPy library. The diagram highlights the main physical components of an active galactic nucleus (accretion disc, dusty torus, relativistic jets, and hot corona) along with the key orientation angles that determine the classification of the AGN from the observer's perspective.

of the energy is radiated from the optical regions of the spectrum and what part from the Infrared part instead. Lastly, the wrapper class also exposes the possibility to run with either of the two AGN models available in GalaPy: the Panchromatic model summarised here (which will be the focus of a follow up work) or the template-based model from Fritz et al. (2006) which is already available in the current version of the library.

5. Summary and outlook

In this work we have presented the computational structure and design philosophy of GalaPy, a hybrid C++/Python library for the spectral modelling of galaxies. Originally introduced in Ronconi et al. (2024), GalaPy has been developed within the Italian galaxy formation community (SISSA, IFPU) as part of the national HPC programme ICSC – Centro Nazionale di Ricerca in High Performance Computing, Big Data e Quantum Computing. Its implementation reflects the dual need for physical rigour and computational efficiency required by modern SED-fitting analyses in the Exascale era.

The library achieves a balance between performance and flexibility through a modular object-oriented C++ core coupled with a high-level Python interface. The compiled backend performs the most computationally intensive operations, while Python handles interoperability with external packages, data I/O, and user interaction. This architecture ensures portability and accessibility while retaining the ability to exploit low-level optimisations such as vectorisation and CPU cache-aware operations.

Parallel execution in GalaPy is designed to accommodate both shared- and distributed-memory environments (Section 3). Although constrained by the limitations of Python's runtime and the Global Interpreter Lock, the library supports hybrid strategies based on multiprocessing, native OpenMP parallelism, and MPI message passing. Performance profiling shows that typical CPU utilisation remains at 20–40%, indicating that the main bottlenecks arise from the Python layer rather than from the compiled backend. Future development will therefore focus on tighter C++/Python coupling and more efficient task scheduling, aiming at improved scaling on HPC infrastructures.

The most recent releases also introduced significant scientific extensions (Section 4), including Bayesian evidence-based model selection and a fully analytical panchromatic AGN model. These updates expand

the physical and statistical capabilities of the library, while preserving its computational efficiency and modularity.

Overall, GalaPy provides a versatile, transparent, and performant framework for galaxy SED modelling, designed to evolve alongside the rapidly changing computational landscape of modern astrophysics. Its hybrid implementation and open design philosophy make it well suited for integration within larger HPC workflows, supporting the next generation of survey analyses.

CRediT authorship contribution statement

Tommaso Ronconi: Writing – review & editing, Writing – original draft, Validation, Software, Project administration, Methodology, Investigation, Formal analysis, Conceptualization. **Andrea Lapi:** Writing – review & editing, Supervision, Project administration, Methodology, Conceptualization.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgements

During the preparation of this work the authors used ChatGPTv5 by OpenAI in order to review the language, as none of them is a native English speaker. After using this tool/service, the authors reviewed and edited the content as needed and take full responsibility for the content of the published article. This work made use of the C++ (Stroustrup, 2013) and Python (Van Rossum et al., 2007) programming languages, and of the following software: Astropy (The Astropy Collaboration et al., 2013), NumPy (Harris et al., 2020), SciPy (Virtanen et al., 2020), Matplotlib (Hunter, 2007), Emcee (Foreman-Mackey et al., 2013), dynesty (Speagle, 2020), GetDist (Lewis, 2019), pybind11 (Jakob et al., 2017), HDF5 (Folk et al., 2011), h5py (Collette et al., 2021).

This work is founded by the Italian Research Center on High Performance Computing Big Data and Quantum Computing (ICSC), project funded by European Union - NextGenerationEU - and National Recovery and Resilience Plan (NRRP) - Mission 4 Component 2 within the activities of Spoke 3 (Astrophysics and Cosmos Observations).

References

- Allison, R., Dunkley, J., 2013. Comparison of sampling techniques for Bayesian parameter estimation. *Mon. Not. R. Astron. Soc.* 437 (4), 3918–3928. doi:10.1093/mnras/stt2190.
- Battisti, A.J., da Cunha, E., Grasha, K., Salvato, M., Daddi, E., Davies, L., Jin, S., Liu, D., Schinnerer, E., Vaccari, M., Collaboration, C., 2019. MAGPHYS+photo-z: Constraining the physical properties of galaxies with unknown redshifts. *Astrophys. J.* 882 (1), 61. doi:10.3847/1538-4357/ab345d.
- Boquien, Burgarella, D., Roehlly, Y., Buat, V., Ciesla, L., Corre, D., Inoue, A.K., Salas, H., 2019. CIGALE: a python code investigating GALaxy emission. *Astron. Astrophys.* 622, A103. doi:10.1051/0004-6361/201834156.
- Collette, A., Kluyver, T., Caswell, T.A., Tocknell, J., Kieffer, J., Scopatz, A., Dale, D., Jelenak, A., VINCENT, T., Sciarrelli, P., et al., 2021. h5py/h5py: 3.1.0. Zenodo.
- Da Cunha, E., Charlot, S., Elbaz, D., 2008. A simple model to interpret the ultraviolet, optical and infrared emission from galaxies. *Mon. Not. R. Astron. Soc.* 388 (4), 1595–1617. doi:10.1111/j.1365-2966.2008.13535.x.
- Folk, M., Heber, G., Koziol, Q., Pourmal, E., Robinson, D., 2011. An overview of the HDF5 technology suite and its applications. In: *Proceedings of the EDBT/ICDT 2011 Workshop on Array Databases*. pp. 36–47.
- Foreman-Mackey, D., Hogg, D.W., Lang, D., Goodman, J., 2013. emcee: The MCMC Hammer, vol. 125, Publications of the Astronomical Society of the Pacific, p. 306. doi:10.1086/670067, arXiv:1202.3665.
- Fritz, J., Franceschini, A., Hatziminaoglou, E., 2006. Revisiting the infrared spectra of active galactic nuclei with a new torus emission model. *Mon. Not. R. Astron. Soc.* 366 (3), 767–786. doi:10.1111/j.1365-2966.2006.09866.x.

- Handley, W.J., Hobson, M.P., Lasenby, A.N., 2015. Polychord: nested sampling for cosmology. *Mon. Not. R. Astron. Soc.: Lett.* 450 (1), L61–L65. doi:10.1093/mnras/slv047.
- Harris, C.R., Millman, K.J., Van Der Walt, S.J., Gommers, R., Virtanen, P., Cournapeau, D., Wieser, E., Taylor, J., Berg, S., Smith, N.J., et al., 2020. Array programming with NumPy. *Nature* 585 (7825), 357–362.
- Higson, E., Handley, W., Hobson, M., et al., 2019. Dynamic nested sampling: an improved algorithm for parameter estimation and evidence calculation. *Stat. Comput.* 29, 891–913. doi:10.1007/s11222-018-9844-0.
- Hunter, J.D., 2007. Matplotlib: A 2D graphics environment. *Comput. Sci. Eng.* 9 (3), 90–95. doi:10.1109/MCSE.2007.55.
- Jakob, W., Rhinelanders, J., Moldovan, D., 2017. Pybind11 – seamless operability between c++11 and python. <https://github.com/pybind/pybind11>.
- Johnson, B.D., Leja, J., Conroy, C., Speagle, J.S., 2021. Stellar population inference with prospector. *Astrophys. J. Suppl. Ser.* 254 (2), 22. doi:10.3847/1538-4365/abef67, arXiv:2012.01426.
- Lange, J.U., 2023. nautilus: boosting Bayesian importance nested sampling with deep learning. *Mon. Not. R. Astron. Soc.* 525 (2), 3181–3194. doi:10.1093/mnras/stad2441, arXiv:https://academic.oup.com/mnras/article-pdf/525/2/3181/51331635/stad2441.pdf.
- Lewis, A., 2019. GetDist: a python package for analysing Monte Carlo samples. arXiv preprint arXiv:1910.13970.
- Ronconi, T., Lapi, A., Torsello, M., Bressan, A., Donevski, D., Pantoni, L., Behiri, M., Boco, L., Cimatti, A., D’Amato, Q., Danese, L., Giulietti, M., Perrotta, F., Silva, L., Talia, M., Massardi, M., 2024. GalaPy: A highly optimised C₊₊/Python spectral modelling tool for galaxies. I. Library presentation and photometric fitting. *Astron. Astrophys.* 685, A161. doi:10.1051/0004-6361/202346978, arXiv:2402.12427.
- Speagle, J.S., 2019. A conceptual introduction to Markov chain Monte Carlo methods. doi:10.48550/arXiv.1909.12313, arXiv e-Prints, arXiv:1909.12313. arXiv:1909.12313.
- Speagle, J.S., 2020. DYNESTY: a dynamic nested sampling package for estimating Bayesian posteriors and evidences. *Mon. Notices Royal Astron. Soc.* 493 (3), 3132–3158. doi:10.1093/mnras/staa278, arXiv:1904.02180.
- Stroustrup, B., 2013. *The C++ Programming Language*. Pearson Education.
- The Astropy Collaboration, Robitaille, T.P., Tollerud, E.J., Greenfield, P., Droettboom, M., Bray, E., Aldcroft, T., Davis, M., Ginsburg, A., Price-Whelan, A.M., Kerzendorf, W.E., Conley, A., Crighton, N., Barbary, K., Muna, D., Ferguson, H., Grollier, F., Parikh, M.M., Nair, P.H., Günther, H.M., Deil, C., Woillez, J., Conseil, S., Kramer, R., Turner, J.E.H., Singer, L., Fox, R., Weaver, B.A., Zabalza, V., Edwards, Z.I., Azalee Bostroem, K., Burke, D.J., Casey, A.R., Crawford, S.M., Dencheva, N., Ely, J., Jenness, T., Labrie, K., Lim, P.L., Pierfederici, F., Pontzen, A., Ptak, A., Refsdal, B., Servillat, M., Streicher, O., 2013. Astropy: A community python package for astronomy. *Astron. Astrophys.* 558, A33. doi:10.1051/0004-6361/201322068.
- Van Rossum, G., et al., 2007. *Python programming language*.
- Virtanen, P., Gommers, R., Oliphant, T.E., Haberland, M., Reddy, T., Cournapeau, D., Burovski, E., Peterson, P., Weckesser, W., Bright, J., et al., 2020. Scipy 1.0: fundamental algorithms for scientific computing in Python. *Nature Methods* 17 (3), 261–272.