

SymBoltz.jl: a symbolic-numeric, approximation-free and differentiable linear Einstein-Boltzmann solver

Herman Sletmoen

Institute of Theoretical Astrophysics, University of Oslo, P.O.Box 1029 Blindern, N-0315 Oslo, Norway
e-mail: herman.sletmoen@astro.uio.no

Received XX / Accepted XX

ABSTRACT

SymBoltz is a new Julia package that solves the linear Einstein-Boltzmann equations. It features a symbolic-numeric interface for specifying equations, is free of approximation switching schemes and is compatible with automatic differentiation. Cosmological models are built from replaceable physical components in a way that scales well in model space. The modeler should simply write down their equations, and SymBoltz solves them and eliminates much of the friction in the process. SymBoltz enables up to 100× shorter model definitions compared to browsing equivalent files in CLASS. Symbolic knowledge enables powerful automation of tasks, such as separating computational stages like the background and perturbations, generating the Jacobian matrix and its sparsity pattern, and interpolating arbitrary expressions from the solution. Modern implicit solvers integrate the full stiff equations at all times, reducing slowdowns by taking long time steps, reusing the Jacobian and LU-factorizing it over several time steps, and using fast linear system solvers. Automatic differentiation gives exact derivatives of any output with respect to any input, which is important for gradient-based Markov chain Monte Carlo methods in large parameter spaces, training of emulators, Fisher forecasting and sensitivity analysis. These features are useful in their own rights, but also reinforce each other in a synergy. Results agree with established codes like CLASS and CAMB. With more work, SymBoltz can grow into an integrated symbolic-numeric cosmological modeling environment with a large library of models that delivers differentiable output as fast as other codes. SymBoltz is available at <https://github.com/hersle/SymBoltz.jl> with single-command installation and extensive documentation, and welcomes questions, suggestions and contributions.

Key words. Cosmology: theory - Methods: numerical

Cosmology is at a crossroads (Freedman 2017). Despite the enormous success of the Λ CDM model in explaining many observations, the increasing precision of modern observations are revealing tensions between theory and observations. These suggest that the current standard model is not the ultimate truth.

On the theoretical side, this drives a search for the true cosmological model through modifications to Λ CDM (Bull et al. 2016). Efficiently exploring the space of models benefits from numerical tools that are easy to modify. Minimizing friction in the modeling process encourages relaxing model-dependent approximations, creating user-friendly interfaces and structuring codes in modular components that are easy to replace. Some of the most important such tools in the cosmological modeling toolbox are linear Einstein-Boltzmann solvers (“Boltzmann codes”) like CAMB (Lewis et al. 2000) and CLASS (Lesgourgues 2011a).

On the observational frontier, next-generation surveys like the square Kilometer Array (Dewdney et al. 2009), Vera C. Rubin Observatory (LSST Science Collaboration 2009), Dark Energy Spectroscopic Instrument (DESI Collaboration 2016), Simon’s Observatory (The Simons Observatory Collaboration 2019) and Euclid (Euclid Collaboration 2025) promise even more precise data. Setting models apart with upcoming data involves both theoretical model parameters and experimental nuisance parameters that live side-by-side in large $O(100)$ -dimensional spaces (Piras et al. 2024). In high dimensions, modern Markov chain Monte Carlo methods like Hamiltonian Monte Carlo and the No-U-Turn Sampler (Hoffman & Gelman 2011) beat the traditional Metropolis-Hastings algorithm (Hastings 1970). These methods explore parameter space more efficiently, but need both the like-

lihood and its derivatives with respect to parameters. Differentiability is also increasingly important in other applications, such as forward-modeling field-level inference using simulations with initial conditions from Boltzmann solvers (Seljak et al. 2017). Automatic differentiation is a way to compute derivatives that is more accurate and can be faster than finite differences, which is an approximate and brute-force method. A differentiable Boltzmann solver is therefore a cornerstone in the next-generation cosmological modeling toolbox.

SymBoltz is a new Boltzmann solver that aims to fill these gaps, featuring a convenient symbolic-numeric interface, approximation-freeness and differentiability. At its core, a Boltzmann code solves the gravitational (Einstein) equations for a gravitational theory coupled to some particle species described by thermodynamic (Boltzmann) equations up to first perturbative order around a homogeneous and isotropic universe. It can predict cosmic microwave background (CMB), baryon acoustic oscillation (BAO) or supernova (SN) observations; or be used in longer computations, such as to generate initial conditions to non-linear N -body simulations of large-scale structure.

This article is structured as follows. Section 1 revisits the historical development of Boltzmann codes, sketches their structure, reviews methods for computing derivatives and motivates SymBoltz. Section 2 maps out the architecture and main features of SymBoltz and how it differs from other codes. Section 3 shows some example usage. Section 4 discusses synergies and tradeoffs in the design. Section 5 concludes with the current state of the code and future potential. The appendices list equations, implementation details, comparison settings and testing methods.

1. History and motivation

Peebles & Yu (1970) were first to numerically integrate a comprehensive set of linearized Einstein-Boltzmann equations. Their work was refined over several years, and Ma & Bertschinger (1995) established the groundwork for modern Boltzmann solvers with their code COSMICS. Seljak & Zaldarriaga (1996) soon realized that one can integrate photon multipoles by parts to reduce many differential equations to one integral solution and released CMBFAST. Their *line-of-sight integration* method has become a standard technique that greatly speeds up the calculation, but requires truncating the multipoles in the differential equations. This Fortran codebase has since evolved into CAMB¹ written by Lewis et al. (2000), which is one of the two most used and maintained Boltzmann solvers today. Doran (2005a) ported CMBFAST to C++ with the fork CMBEASY, which structured the code in an object-oriented fashion and improved user-friendliness, but this project is abandoned today.

Lesgourgues (2011a); Blas et al. (2011) started the second major family of Boltzmann solvers with the birth of CLASS² in C. It improved performance, user-friendliness, code flexibility, ease of modification and control over precision parameters. It was the first cross-check of CAMB from an independent branch and boosted the scientific accuracy of the Planck mission.

Since then a healthy arms race have fueled refinements to CAMB and CLASS. Both codes have spawned many forks for studying alternative models and performing custom calculations. Today they are very well-made, efficient and reliable tools.

More recently, the market has seen an influx of alternative solvers that target new numerical techniques, GPU parallelization, differentiability, interactivity and symbolic computation. Refregier et al. (2017) published the first symbolic-numeric Boltzmann code PyCosmo³, which automatically generates efficient C++ ODE code from user-provided symbolic equations, performs sparsity optimization on its Jacobian matrix and avoids the use of approximation schemes to speed up and stabilize the integration. Hahn et al. (2024) published the first differentiable Boltzmann solver DISCO-EB⁴ in the JAX framework in Python, and relaxes approximation schemes to avoid complications and overhead when switching ODEs on GPUs. Bolt.jl⁵ by Li et al. (2023) accomplishes much of the same in Julia. SymBoltz.jl⁶ is inspired by some of these developments.

1.1. Structure of traditional Boltzmann solvers

The full Einstein-Boltzmann equations are partial differential equations that linearize to ordinary differential equations.

In principle, the core task of a Boltzmann solver is thus very simple: to solve these ordinary differential equations (ODEs) $\mathbf{du}/d\tau = \mathbf{f}(\tau, \mathbf{u}; \mathbf{p})$ for some initial conditions $\mathbf{u}(\tau_0)$ and parameters $\mathbf{p}$, including several perturbation wavenumbers, and compute desired quantities from their solution, like power spectra.

In practice, however, several properties complicate this task. First, the equations separate into computational stages that benefit substantially from being solved sequentially for performance and stability, such as the background and perturbations stages. It is common to solve each stage with interpolated input from the

previous stage, and joining these can be cumbersome and fragment code. Second, the set of equations is very long and convoluted. Ideally, parts of the equations should be easily replaceable to accommodate different cosmological (sub)models. Organizing a code with a suitable modular structure that scales well in model space is hard. Third, tension between wildly different timescales that coexist in the equations make them extremely stiff and intractable for standard explicit ODE solvers. This stiffness must be massaged away in the equations or dealt with numerically in other ways. Fourth, the size of differential equations is very large and increases for more accurate description of relativistic species. Typical models with accurate treatment of photons and neutrinos need $O(100)$ equations. Fifth, the perturbations must be solved for many different wavenumbers k , and tradeoffs between performance and accuracy must be made.

To overcome these challenges, Boltzmann solvers are typically written in low-level high-performance languages like C, C++ and Fortran. Codes are usually tightly adapted to the pipeline-like *computational* structure of the problem (e.g. input $\rightarrow$ background $\rightarrow$ thermodynamics $\rightarrow$ perturbations $\rightarrow \dots \rightarrow$ output in section 3.3 in Lesgourgues (2011a)). This makes sense for programmers and computers, but not always for modelers.

Traditional codes have nonexistent or only very thin abstraction layers around them. To modify them, users must work directly in the low-level numerical interface and should be prepared to get their hands dirty. For example, to implement a new species, one must generally modify the code in many places: input handling for any new parameters, new background equations, new thermodynamics equations, new perturbation equations, joining between each of these stages, output handling, and possibly more. This leads to fragmented code where the changes related to one species are scattered throughout the code, making it challenging and unintuitive to navigate and decipher.

More importantly, this structure scales poorly in model space. Each module becomes more polluted as more components are added. Even if they are deactivated by if statements at runtime, their mere presence in the source code reduces its readability. In turn, the whole code grows into a complex and mysterious beast.

One can alleviate this problem to some extent by instead forking the code for modified models, so the main code base is not polluted. But this just moves the problem. Forking duplicates the entire code base even though only small parts are modified. Forks are often abandoned and do not receive upstream improvements. They are also incompatible with each other unless one merges them into one code, but then one is back to the first problem.

The two-language problem amplifies the problem, as data analysis usually takes place in slower high-level languages like Python. This shapes Boltzmann solvers to rigid pipelines that must compute everything in one shot and avoid interception at all costs, in order to maximize performance in the low-level language before passing the data back to the high-level language. Some features that are really just post-processing of the ODE solutions are appended to the pipeline even if they are peripheral to the core task of an Einstein-Boltzmann solver – which is to solve the Einstein-Boltzmann equations. This includes non-linear boosting and CMB lensing, for example.

These code smells lead to big monolithic “black boxes” that scale poorly in model space, and whose complexity grows to incorporate features beyond their original core scope. This complexity has even fueled development of specialized AI assistants for CLASS (Casas et al. 2025). While such tools are helpful, we think they are a symptom of unnecessary complexity and poor scaling with the number of models. Of course, there are also some advantages to this structure, as we discuss in section 4.

¹ <https://camb.info/>

² <http://class-code.net>

³ <https://pypi.org/project/PyCosmo/>

⁴ <https://github.com/ohahn/DISCO-EB>

⁵ <https://github.com/xzackli/Bolt.jl>

⁶ <https://github.com/hersle/SymBoltz.jl>

1.2. Boltzmann solver approximation schemes

The Einstein-Boltzmann equations are infamously *stiff*. This is a property of differential equations (3) that means their numerical solution is unstable and requires tiny step sizes with standard explicit Runge-Kutta methods. Stiffness can arise when multiple and very different time scales appear in the same problem. This is very common in cosmology, where particles interact very rapidly in a universe that expands very slowly, particularly in the tightly coupled baryon-photon fluid, for example. Stiff equations are practically impossible to integrate with explicit solvers and require special treatment.

For a long time, Boltzmann solvers have massaged away stiffness with several approximation schemes⁷ in the equations (e.g. Doran (2005b); Blas et al. (2011)):

- tight-coupling approximation (TCA),
- ultra-relativistic fluid approximation (UFA),
- radiation streaming approximation (RSA),
- Saha approximation.

These usually involve switching from one set of equations to another when some control variable that measures the applicability of the approximation crosses a threshold. This can change the unknowns in the ODE and require reinitializing it. The approximations help stability and sometimes performance of the equations and allow the use of explicit solvers.

However, approximations put much more strain on the modeler to derive and validate versions of their equations in different regimes. Furthermore, this process should generally be repeated when further changes are made to the model. The numerics also become more complicated: timeseries from each ODE solution must be stitched together, tolerances are duplicated for each separate ODE system that may respond differently to them, ODE integrators must be reinitialized, and so on.

It is a misunderstanding in the literature that it is “difficult” or “impossible” to solve the Einstein-Boltzmann equations without using approximations, or that they are “unavoidable” (Ma & Bertschinger 1995; Lewis 2025; Lesgourgues 2011a, respectively). This statement must only be understood in the context of explicit solvers! Implicit integrators can solve stiff equations, but seem largely underutilized by Boltzmann solvers, perhaps because they are scarce and harder to implement. CLASS has an implicit solver, but does not permit disabling all approximations Blas et al. (2011). On the contrary, new life has been breathed into this field recently, as development of modern implicit solvers and techniques like automatic and symbolic differentiation make implicit solvers more feasible and powerful.

1.3. Differentiation methods

Derivatives are important in computing. Cosmological applications are no exception. For example, algorithms that optimize likelihoods and Markov chain Monte Carlo (MCMC) samplers for Bayesian parameter inference can take advantage of derivatives of the likelihood with respect to each parameter to intelligently step in a direction where the likelihood or probability increases. In machine learning, the same applies when training neural networks emulators for cosmological observables as functions of cosmological parameters by minimizing scalar loss functions. Some cosmologies are parametrized as boundary-value problems

⁷ Here “approximations” refers to schemes that switch between different equations at different times. It excludes techniques like multipole truncation and line-of-sight integration, which SymBoltz also uses.

with the shooting method, and use nonlinear root solvers like Newton’s method that need Jacobians. Jacobians are also used by implicit ODE solvers to solve for values at the next time step. Fisher forecasting uses the Hessian (double derivative) of the likelihood with respect to parameters to predict how strong parameter constraints that can be placed by data with given errors. Boltzmann solvers typically save computational resources by interpolating spectra computed on coarse grids of k and l to finer grids, and this can be made more precise with knowledge of derivatives with respect to k and l . These are all examples where Boltzmann solvers or applications that use them need derivatives.

There are at least four ways to compute derivatives. *Manual differentiation* is the exact pen-and-paper method with differentiation rules, but is limited to simple expressions and by human error. *Symbolic differentiation* automates this process with computer algebra systems, but is inherently symbolic and cannot differentiate arbitrary programs nested with control flow through conditional statements and loops that depend on numerical values. *Finite differentiation* approximates the derivative $f'(x) \approx (f(x + \epsilon/2) - f(x - \epsilon/2)) / \epsilon$ with a small $\epsilon > 0$ (here using central differences) by simply evaluating the program several times. This can differentiate arbitrary programs, but is approximate, introduces the step size ϵ as a hyperparameter that must be tuned for both accuracy and stability, and is a brute-force approach that requires $O(n)$ ($2n$ using central differences) evaluations to compute the gradient of an n -variate scalar f . *Automatic differentiation* (e.g. Griewank & Walther 2008) can be understood by viewing any computer program as a (big) composite function

$$f = f_N \circ f_{N-1} \circ \dots \circ f_2 \circ f_1 = f_N(f_{N-1}(\dots f_2(f_1))) \quad (1)$$

of (many) elementary operations $f_i : \mathbb{R}^{a_i} \rightarrow \mathbb{R}^{b_i}$ (think of f_i as the i -th line of code). It then numerically evaluates the chain rule

$$J = J_N \cdot J_{N-1} \dots J_2 \cdot J_1. \quad (2)$$

through the Jacobian $(J_n)_{ij} = \partial f_{n,i} / \partial f_{n-1,j}$ of every elementary operation to accumulate the derivative of the entire program. This is numerically exact, free of precision parameters, usually requires fewer operations than finite differences, but is perhaps less intuitive, harder to implement and needs the source code to interpret the program (1) in the non-standard way (2).

Notably, while the function (1) must be evaluated inside-to-outside (right-to-left), the chain rule (2) is an *associative* matrix product that can be evaluated in any order. This generally changes the number of operations and is a more open-ended computational problem. *Forward-mode* automatic differentiation seeds $J_1 = \mathbf{1}$ (the derivative of the input with respect to itself) and multiplies $J_N(J_{N-1}(\dots (J_2(J_1))))$ by “pushing” every column of J_1 forward through the product in the same evaluation order as f . *Reverse-mode* first computes f in a forward pass, then seeds $J_N = \mathbf{1}$ (the derivative of the output with respect to itself) and multiplies $((((J_N)J_{N-1}) \dots J_2)J_1)$ by “pulling” every row of J_N backwards through the product. This usually makes forward-mode faster when $f : \mathbb{R}^a \rightarrow \mathbb{R}^b$ has more outputs $b \gg a$, and reverse-mode better when there are more inputs $a \gg b$.

In practice, both modes are implemented using techniques like operator overloading or source code transformation. The former overloads every operation to accept a special number type that propagates both values and derivatives, such as *dual numbers* for forward-mode (e.g. Revels et al. 2016). The latter transforms the code for f into another code that computes J . In any case, automatic differentiation requires access to the source code!

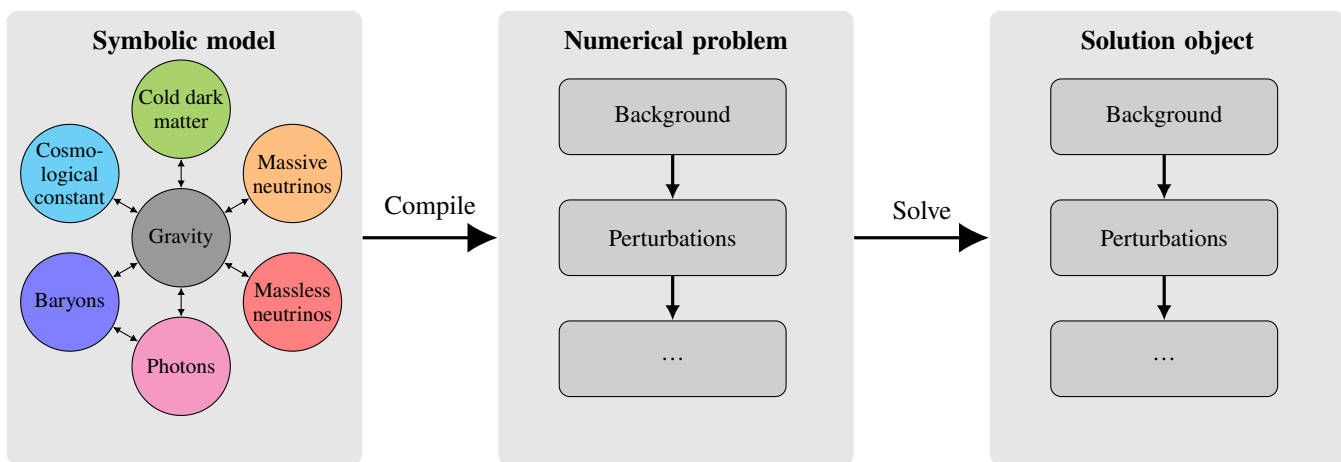


Fig. 1. SymBoltz represents cosmological models with equations in *symbolic* form and grouped by physical components like the spacetime metric, gravity and particle species. This is compiled to a *numerical* problem that splits the model into background and perturbation stages and generates efficient numerical code for ODEs and their Jacobians. The problem is then solved and the result stored in a solution object that gives convenient access to any quantity defined in the symbolic model.

2. Code architecture and main features

SymBoltz is built somewhat differently than other Boltzmann codes. In short, it uses a symbolic-numeric abstraction interface where users enter high-level symbolic equations, and automatically compiles them to fast low-level numerical functions. It cures stiffness with implicit ODE solvers, and uses no approximation schemes to keep models simple, elegant and extensible. It is differentiable, so one can easily get accurate derivatives of any output with respect to any input. This provides rapid model prototyping, helpful abstractions and automation of typical chores when modifying other codes. SymBoltz encourages interactive usage and pursues a modular design that lets users integrate whatever they need from the package into their own applications. The end goal is to prioritize the *modeler*, who should be able to just write down the model equations while SymBoltz takes care of the rest.

SymBoltz is written in the Julia programming language (Bezanson et al. 2017), which attracts development of modern numerical methods and aims to resolve the two-language problem. The symbolic-numeric interface is built on the ModelingToolkit.jl (Ma et al. 2022) and Symbolics.jl (Gowda et al. 2022) packages. The compiled functions are integrated by implicit ODE solvers in DifferentialEquations.jl (Rackauckas & Nie 2017). Everything works with automatic differentiation through ForwardDiff.jl (Revels et al. 2016), for example.

The next subsections present the design of SymBoltz in more detail, revolving around the three key features in the title. Other minor implementation details are given in section A. **This paper is as of SymBoltz version 0.8.1. Please refer to the package documentation for definitive up-to-date information.**

2.1. Symbolic-numeric interface

The core of SymBoltz is built around a symbolic-numeric interface shown in fig. 1. Variables and equations are specified in a high-level user-friendly symbolic modeling language, then ultimately compiled down to efficient low-level numerical functions that are integrated by ODE solvers. This fits the interactive and just-in-time compiled nature of Julia. A key realization is that knowledge of *symbolic* equations makes it possible to analyze their structure programmatically, enabling powerful transformations of the equations and automation of chores.

SymBoltz turns the layout of traditional Boltzmann solvers inside-out. While other codes are built like rigid pipelines following *computational stages* like the background and perturbations, SymBoltz is primarily structured around *physical components* of the Einstein-Boltzmann system. Full cosmological models are built by joining submodels for the metric, gravitational theory, photons, baryons, dark matter, dark energy, neutrinos and other species. Each component is a chunk of related variables and internal equations, and is unaware of other components. Interactions (e.g. Compton scattering or sourcing of gravity) are equations that connect components. This structure is made possible by the symbolic interface and would slow a purely numerical code.

Separation of computational stages is secondary and done automatically (see section 2.1.3). This makes it possible to write everything related to one component *in one place*, instead of scattering it across separate modules for input, background, perturbations and so on. One component can be replaced by another to construct a different cosmological model without polluting a big global model. A large part of SymBoltz is thus simply devoted to building a well-organized library of such physical components.

This modular structure scales better in model space and makes it easier to assemble both extended and reduced models. While many Boltzmann solvers hardcode baryons and photons because they are fundamental to the code, SymBoltz works naturally even with pure models with non-interacting radiation, matter and dark energy. For example, this can help understand how a modified gravity theory responds to different species without complex interactions or thermodynamics cluttering the picture.

Maybe this structure is unfamiliar to some. As a compromise, SymBoltz also provides an “unstructured” version of the Λ CDM model with all equations and variables in one big system. This makes it easy-peasy to change anything in the model, and really shows the power of the symbolic interface: SymBoltz defines the full Λ CDM model (with GR, baryons, RECFast recombination, photons, cosmological constant, massive and massless neutrinos) in just 277 lines of code, while the equivalent code that one must realistically browse in CLASS is spread over 10 files with 27721 lines!⁸ SymBoltz has less features than CLASS, but this is not even remotely close to making up for the 100× simplification.

Let us see which helpful features this interface provides.

⁸ Counting `input.{h,c}`, `background.{h,c}`, `thermodynamics.{h,c}`, `perturbations.{h,c}` and `wrap_recfast.{h,c}` with `wc -l`.

2.1.1. Automatic numerical code generation

SymBoltz automatically compiles symbolic equations to numerical code for ordinary differential equations (ODEs)

$$\frac{d\mathbf{u}}{d\tau} = \mathbf{f}(\tau, \mathbf{u}). \quad (3)$$

The generated code is fast and prevents users unfamiliar with the language or package from writing slow numerical code. If necessary, one can escape the standard code generation and include arbitrary numerical code, but the user is then fully responsible for its performance. This accommodates functions that cannot be written as straightforward equations, for example if one must solve a nonlinear equation for the minimum of a potential or interpolate tabulated data. The code generation deals with chores like dynamically allocating indices for each ODE state u_i , and performs optimizations such as common subexpression elimination. This is helpful as Boltzmann solvers tend to have big $\mathbf{f}$, and particularly with implicit ODE solvers that call $\mathbf{f}$ often.

2.1.2. Automatic handling of observed variables

In general, an ODE (3) admits two types of variables: *unknowns* $\mathbf{u}(\tau)$ are the variables that are integrated with respect to time, and *observed* variables are functions of the unknowns. The Einstein-Boltzmann equations are commonly formulated with many observed variables.

For example, consider the metric and gravity equations in sections A.1 and A.2 sourced by some arbitrary $\rho(\tau)$, $\delta\rho(\tau, k)$ and $\Pi(\tau, k)$. Here $a(\tau)$ and $\Phi(\tau, k)$ are the only unknown (differential) variables that the ODE solver must solve for, while one can “observe” $z(\tau)$, $\mathcal{H}(\tau)$, $H(\tau)$, $\chi(\tau)$ and $\Psi(\tau, k)$ from the unknowns. Of course, it is always possible to eliminate all observeds by explicitly inserting them into the equations for the unknowns. But this kills readability, as observed variables are helpful and natural intermediate definitions that break up the system, and one may want to extract them from the solution as well. Furthermore, modified models can change the sets of unknown and observed variables (e.g. modified gravity can change the constraint equation for Ψ into a differential equation). It is easier to compose models when variables do not have to be hardcoded as either unknown or observed.

SymBoltz reads in an entire system of equations like that defined by section A and automatically figures out which variables are unknown and observed. It then generates numerical code for solving the ODE for the unknowns (see section 2.1.1), and can lazily compute any observed variable or expression thereof from the solution of the unknowns.

The bottom line is that the user can trivially use and obtain any variable anywhere just by referring to it, whether it is unknown or observed. In other solvers one must look up the desired expression for observed variables and recompute them manually. This is tedious, error-prone and can tempt users to neglect parts of expressions (e.g. approximating $\Psi \approx \Phi$ by neglecting anisotropic stress).

2.1.3. Automatic stage separation and splining of unknowns

In principle, one can solve the Einstein-Boltzmann equations by integrating the entire system (i.e. background and perturbations) at the same time. However, the system can be broken down into sequential computational stages that each depend only on those before it. To alleviate stiffness in each stage, separate concerns by solving every equation only once (e.g. avoid recomputing the

background for every perturbation mode) and to improve performance, it is common to solve the system stage-by-stage and spline variables from one stage as input to the next.

To illustrate, again consider the general relativistic equations in section A.2 sourced by some given $\rho(\tau)$, $\delta\rho(\tau, k)$ and $\Pi(\tau, k)$. Clearly $\Phi(\tau, k)$ and $\Psi(\tau, k)$ depend on $a(\tau)$, but $a(\tau)$ does not depend on $\Phi(\tau, k)$ or $\Psi(\tau, k)$ (this just reflects the perturbative nature of the problem). One can first solve for only $a(\tau)$ in the *background*, then spline $a(\tau)$ and use it as input for solving for $\Phi(\tau, k)$ and $\Psi(\tau, k)$ in the *perturbations*, instead of solving for all three together and repeatedly solve for $a(\tau)$ for every k .

SymBoltz uses the same stage separation strategy, but automates and strengthens it with its knowledge of the symbolic equations. First, the full symbolic model is split into background and perturbation systems. Cubic Hermite splines are then constructed for all *unknown* variables (like $a(\tau)$) to pass their solution from one stage to the next. This type of splines is perfect for interpolating ODEs, as cubic Hermite splines increase accuracy by taking both $\mathbf{u}(\tau)$ and $\mathbf{u}'(\tau)$ into account, and $\mathbf{u}'(\tau)$ is known analytically by definition (3). In contrast, *observed* variables (like $\Psi(\tau, k)$) are computed from the (splined) unknowns because their derivatives are not known analytically. Splining them directly would be less accurate. Modelers can simply write down a new background variable and access it in the perturbations for free. They can access any variable anywhere *as if* solving the entire set of equations at once, while still benefiting from splining under the hood.

The separation into background and perturbation stages just reflects the perturbative structure of the linearized Einstein-Boltzmann equations, where each order depends only on those before it, and is always guaranteed. However, they can often be broken further down: most thermodynamics (recombination) models can be separated from the background, and some variables have integral solutions (e.g. the optical depth $\kappa(\tau) = \int_{\tau_0}^{\tau} \kappa'(\bar{\tau}) d\bar{\tau}$ or line-of-sight integration) that can be computed in isolation after solving the differential equations. Over time, SymBoltz aims to extend the automatic background-perturbation separation by automatically incorporating more or *all* stages of the equations at hand.⁹ This feature is a major advantage with access to the symbolic equations.

2.1.4. Automatic solution interpolation

SymBoltz integrates equations in conformal time τ for several given wavenumbers k , and stores the result in an object that wraps independent ODE solutions for the background and perturbations for every k . Conveniently, this solution object can be queried for any variable or symbolic expression

$$x(\tau, k) = x(\mathbf{u}(\tau, k_i)). \quad (4)$$

at any τ and k . The solution object automatically expands x in terms of unknowns $\mathbf{u}$ (if x is observed), and interpolates between solved wavenumbers k_i (if x is perturbative) and in τ using the ODE solver’s built-in interpolation method (or as in section 2.1.3 if x is splined). This provides the modeler easy access to any variable and expression defined in the model. The solution interpolation is also incorporated into convenient plot recipes for trivial visualization of any expressions for any τ and k .

⁹ All stages can in principle be identified from a directed dependency graph between variables (e.g. from the Jacobian matrix). Cycles in the graph correspond to interdependent equations that must be integrated together in one ODE system (e.g. background and perturbations). Leaves with differentiated variables correspond to integral solutions.

2.1.5. Automatic Jacobian generation and sparsity detection

Just like SymBoltz generates code for f in the ODE (3), it can use the same symbolic equations to generate its Jacobian

$$J_{ij} = \frac{\partial f_i}{\partial u_j}. \quad (5)$$

Jacobians are *crucial* for solving stiff ODEs accurately and efficiently with implicit solvers (see section 2.2)! Manually coding them is an extremely tiresome and error-prone undertaking that must be repeated for model modifications. Evaluating them numerically with finite differences is approximate and expensive. Automation of this procedure is a powerful feature that again enables the modeler to focus solely on the main equations.

Another benefit is that analytical knowledge of the Jacobian lets one find its exact sparsity pattern. Numerical approaches to this are less robust and can mistake local zeros for global zeros. This permits constant factorization of sparse Jacobians, which improves performance of implicit solvers (see section 2.2). Full support for sparse Jacobians is still ongoing work, however.

SymBoltz can fall back from symbolic to automatic differentiation for the Jacobian. Finite differences can also be used.

2.1.6. Automatic change of variable (future work)

SymBoltz consistently formulates all differential equations with respect to conformal time τ as the independent variable. This is perhaps the most common parametrization in the literature. It is very natural because the equations are autonomous with respect to τ (i.e. $f(\tau, \mathbf{u}) \rightarrow f(\mathbf{u})$), except some multipole truncation schemes that use $1/(k\tau)$, but these are unphysical).

Some other codes inconsistently use different independent variables in different stages. This requires manual translation into cosmic time t , redshift z , scale factor a or its logarithm $b = \log a$, for example, involving differential transformations like $dt = a(\tau) d\tau$, $dz = -(a'(\tau)/a(\tau)^2) d\tau$, $da = a'(\tau) d\tau$ and $db = (a'(\tau)/a(\tau)) d\tau$. This is another mechanical and error-prone process.

In the future, SymBoltz could automate change to another independent variable (which should be one-to-one with τ) by transforming the symbolic expressions. This can be particularly useful to compute observables as a direct function of redshift, although one can invert $z(\tau)$. This is common in fits to observations.

2.1.7. Automatic unit handling (future work)

Most variables in SymBoltz are defined in internal dimensionless units (see section A). In the future, SymBoltz could use its symbolic pre-processing for more powerful unit features. For example, equations could be checked for dimensional validity to catch modeling mistakes. Input quantities could be automatically transformed from the user's preferred (dimensionful) units (e.g. Mpc) to internal (dimensionless) units, and then back to the user's units for output.

2.1.8. Automatic gauge transformation (future work)

SymBoltz is currently formulated only in the conformal Newtonian gauge. In the future, SymBoltz could conveniently transform the symbolic equations to other gauges automatically, such as the synchronous gauge used in CAMB and CLASS.

2.1.9. Automatic initial conditions (future work)

SymBoltz currently uses adiabatic initial conditions for the perturbations. In the future, SymBoltz could generate initial conditions for arbitrary models from the equations. This typically involves mechanical procedures like perturbation theory or power series expansion that could be automated.

2.1.10. Automatic approximation schemes (future work)

SymBoltz is initially designed to be free of approximation schemes (see section 2.2). In the future, SymBoltz could provide convenience utilities to help derive approximations to the equations. For example, it could automatically expand equations in power series of some smallness parameter. This could accelerate derivation of approximations for modified models to improve their performance with less effort.

2.2. Approximation-freeness

SymBoltz treats stiffness in the Einstein-Boltzmann equations with modern implicit ODE solvers that integrate the full equations at all times. It is therefore free of approximation schemes, such as the TCA, UFA, RSA and Saha approximation. This is much friendlier to the modeler, who now simply has to provide a single set of equations instead of deriving approximations, validating them and dealing with related chores.

However, implicit ODE solvers are more expensive than explicit methods. In particular, at every time step they must solve a nonlinear system of equations for the next unknowns. This is usually done with Newton's method that requires the Jacobian of the nonlinear system, which in turn involves the ODE Jacobian (5). Automatic, accurate and efficient computation of the Jacobian (see section 2.1.5) makes this less of a problem.

As the size of the ODE and its Jacobian increases (e.g. larger $l_{\max}$), solving the large system of equations at every time step becomes a bottleneck. One should then transition from dense to sparse matrix methods (this is still ongoing work) and use performant matrix methods when solving the nonlinear system. By default, SymBoltz uses the Rodas4P (Steinebach 1995) solver for the background (stiff when thermodynamics is included) and KenCarp4 (Kennedy & Carpenter 2003) for the perturbations. These are both implicit solvers from DifferentialEquations.jl. We discuss this aspect further in section 4.

In the long run, SymBoltz could also implement approximation schemes to maximize speed (see section 2.1.10). However, the goal is to keep the code primarily approximation-free and for all approximations to be secondary and optional.

2.3. Differentiability

SymBoltz is compatible with automatic differentiation. One can obtain the derivative of any output quantity, such as all ODE variables and derived spectra, with respect to any combination of input parameters, like the reduced Hubble parameter h or cold dark matter density parameter Ω_{c0} . Automatic differentiation is also used several places internally, such as for computing the Jacobian for implicit ODE solvers (as an alternative to the fully symbolic approach) and in the shooting method (which is a nonlinear equation solver that wraps around the ODE solver).

Currently SymBoltz is only well-tested with forward-mode dual numbers through ForwardDiff.jl. In particular, support for reverse-mode is very attractive for scalar loss applications (e.g. likelihoods), but left for future work (see section 4.3).

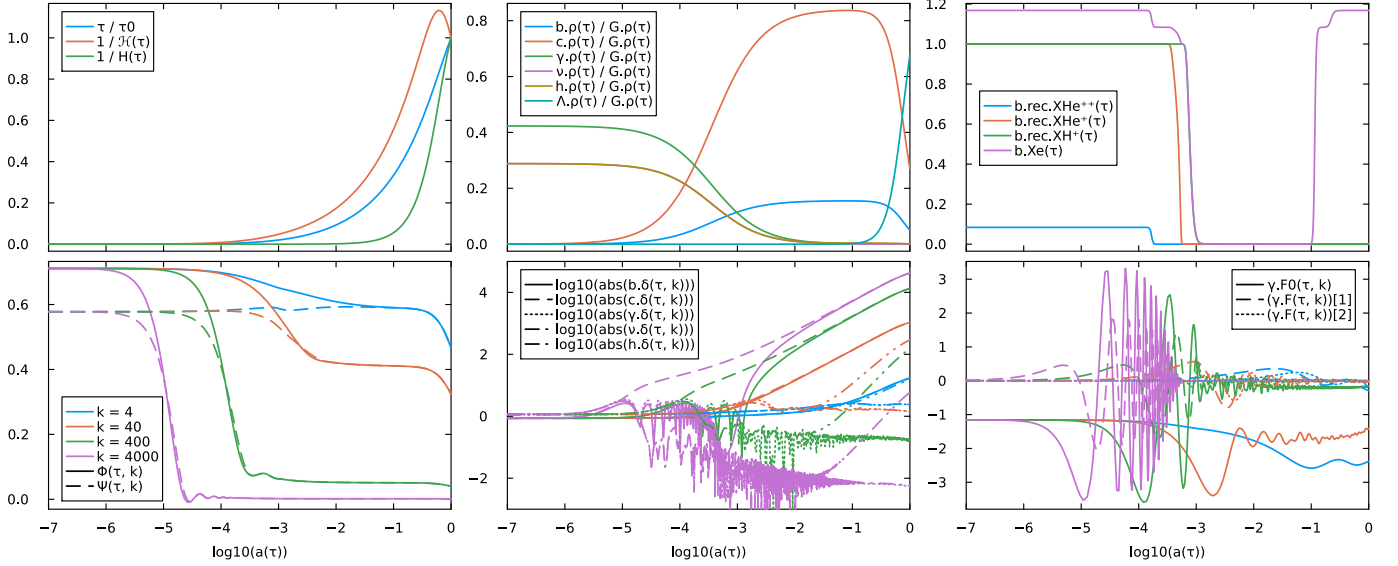


Fig. 2. Included plot recipes in SymBoltz make it trivial to visualize *any* symbolic variable or expression thereof from a solution of the Einstein-Boltzmann equations. This plot was made with one short line of code per subplot. Wavenumbers k are in units of H_0/c .

3. Examples

The best way to illustrate the features in section 2 is perhaps through some examples. The full code is as of SymBoltz version 0.8.1 and available in a notebook in the project repository.

3.1. Basic usage workflow

Most usage follows a model–problem–solution workflow:

```
using SymBoltz
M = ΛCDM(lmax = 10)
pars = Dict{
    M.γ.T₀ => 2.7, M.b.Ω₀ => 0.05, M.b.rec.Υp => 0.25,
    M.v.Neff => 3.0, M.c.Ω₀ => 0.27, M.h.m_eV => 0.06,
    M.I.ln_As1e10 => 3.0, M.I.ns => 0.96, M.g.h => 0.7
}
prob = CosmologyProblem(M, pars)
ks = [4, 40, 400, 4000] # k / (H₀/c)
sol = solve(prob, ks)
```

The model–problem–solution split achieves three distinct goals.

First, a *symbolic* representation of the Λ CDM model M is created. This is a standalone object designed to be interactively inspected and modified independently of numerics (see section 3.2). It contains every variable, parameter and equation of the cosmological model structured as a graph of submodels for each logically distinct (physical) component: the metric g , the gravitational theory G , photons γ , massless neutrinos ν , massive neutrinos h , cold dark matter c , baryons b , the cosmological constant Λ and inflation I . For example, `equations(M)` shows all model equations; `equations(M.G)` shows only the gravitational ones; `M.g.a` gives a handle to the scale factor variable $a(\tau)$ that “belongs” to the metric g ; `M.b.Ω₀` is the density parameter Ω_{b0} in the baryon component b ; and `parameters(M)` lists parameters of the model that the user may set. Everything displays with $\text{\LaTeX}$ -compatibility in notebooks to encourage interactive use.

Second, the symbolic model is compiled to a *numerical* problem `prob` with parameters `pars`. This is an expensive operation where the “magic” in section 2.1 happens: equations are checked for consistency and split into background and perturbation stages;

observed and unknown variables are distinguished; fast ODE code is generated; background unknowns in the perturbations are replaced by splines; the Jacobian matrix is generated in numerical/analytical and dense/sparse form; and initial values are computed. One can customize whether stages should be separated, how the Jacobian should be generated, and so on. This is a separate step because it performs final transformations on the model M once the user has finished modifying and committed to it.

Third, `solve(prob, ks)` solves the background and perturbations for the requested wavenumbers `ks`. Omitting `ks` solves only the background. The resulting solution object `sol` provides convenient access to all model variables. Internally, it stores the values of all ODE *unknowns* at the time steps taken by the (adaptive) solvers and for every requested wavenumber. However, it can be queried with *any* time, wavenumber and symbolic expression, and will automatically compute it from the unknowns and interpolate between stored times and wavenumbers (see sections 2.1.2 to 2.1.4). For example, calling `sol(1.0, 2.0, g.Φ+g.Ψ)` will compute $\Phi + \Psi$ at $k = 1 H_0/c$ and $\tau = 2 H_0^{-1}$ by expanding it in terms of unknowns according to the equations in section A.2. It is also possible to compute arbitrary expressions over grids of k and τ . The convenient solution interface is also integrated into plot recipes for effortless visualization, as shown in fig. 2.

Other Boltzmann solvers save only a hardcoded set of variables, such as only the unknowns. Observed variables must generally be recomputed manually, even though they are already expressed somewhere in the code. This is cumbersome, introduces more potential for user error and adds unnecessary friction in the modeling process. SymBoltz has higher ambitions than simply spitting out a table with some selected variables, and is designed to interactively make modifications to the model with minimal changes and easily inspect its impact on any output.

3.2. Modifying models

Suppose we want to replace the cosmological constant species Λ in the model M with another dark energy species. A well-known example is dynamical $w_0 w_a$ dark energy (Chevallier & Polarski

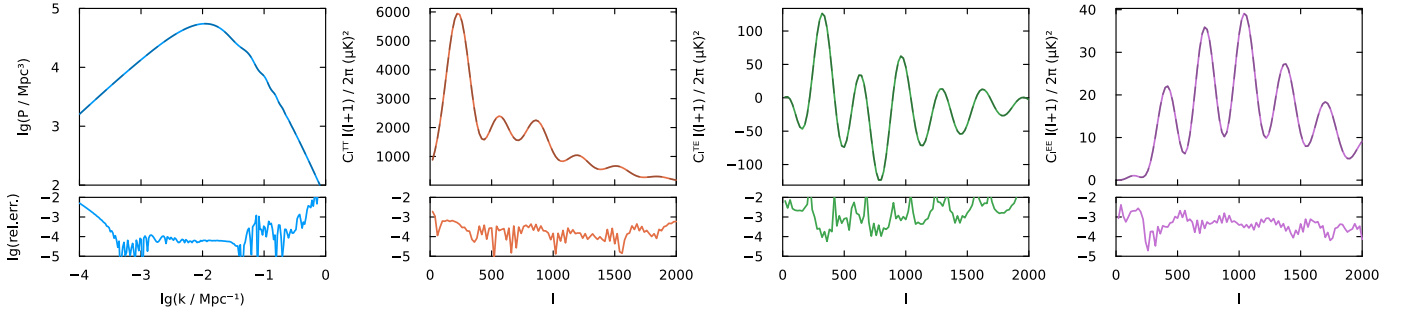


Fig. 3. Matter and CMB (TT, TE and EE) power spectra computed by SymBoltz (SB; colored lines) compared to CLASS (CL; grey dashes) with relative errors $\text{rel.err.} = P_k^{\text{SB}}/P_k^{\text{CL}} - 1$ and $\text{rel.err.} = C_l^{\text{SB}}/C_l^{\text{CL}} - 1$ for the Λ CDM model. CLASS uses the precision parameters in section B.

2001; Linder 2003) with equation of state¹⁰

$$w(\tau) = w_0 + w_a(1 - a(\tau)). \quad (6)$$

In this case, the continuity equation

$$\rho'(\tau) = -3\mathcal{H}(\tau)\rho(\tau)(1 + w(\tau)) \quad (7)$$

admits the analytical solution (by an ansatz of the same form)

$$\rho(\tau) = \rho(\tau_0)a(\tau)^{-3(1+w_0+w_a)} \exp(-3w_a(1 - a(\tau))). \quad (8)$$

To implement this species including perturbations following de Putter et al. (2010), simply write down the symbolic variables, parameters, equations and initial conditions:

```
g, tau, k = M.g, M.tau, M.k
a, H, Phi, Psi = g.a, g.H, g.Phi, g.Psi
D = Differential(tau)
@parameters w0 wa cs^2 Omega0 rho0
@variables rho(tau) P(tau) w(tau) ca^2(tau) delta(tau,k) theta(tau,k) sigma(tau,k)
eqs = [
    w ~ w0 + wa*(1-a)
    rho0 ~ 3*Omega0 / (8*Num(pi))
    rho ~ rho0 * a^(-3*(1+w0+wa)) * exp(-3*wa*(1-a))
    P ~ w * rho
    ca^2 ~ w - 1/(3H) * D(w)/(1+w)
    D(delta) ~ 3H*(w-cs^2)*delta - (1+w) * (
        (1+9(H/k)^2*(cs^2-ca^2))*theta + 3*D(Phi)
    )
    D(theta) ~ (3cs^2-1)*H*theta + k^2*cs^2*delta/(1+w) + k^2*Psi
    sigma ~ 0
]
initialization_eqs = [
    delta ~ -3//2 * (1+w) * Psi
    theta ~ 1//2 * k^2*tau * Psi
]
X = System(eqs, tau; initialization_eqs, name = :X)
```

Everything is packed down into the w_0w_a component X . Note that SymBoltz encourages Unicode symbols to maximize similarity between equations and code (e.g. Ω_0 over Omega_0) and to easily display L^AT_EX in compatible environments (e.g. notebooks).

This is all the user has to do! Variables are automatically split into and carried across the background and perturbation stages, hooks for setting input parameters and getting arbitrary output variables are automatically available, and so on. The modification simply consists of writing down the equations verbatim. It could not have been more compact and to the point.

¹⁰ SymBoltz already includes w_0w_a as an available species.

A similar modification to CLASS is more involved, for example. It would require *at least*: reading new parameters in `input.c`; declaring new background and perturbation variables in `background.h` and `perturbations.h`; solving background equations, storing desired output and coupling them to gravity in `background.c`; and recomputing or looking up background variables, solving perturbation equations and storing desired output and coupling them to gravity in `perturbations.c`. The changes should be if-guarded correctly to ensure the code works both with and without w_0w_a .

A full w_0w_a CDM model and problem can now be built by replacing the cosmological constant species Λ in Λ CDM by the w_0w_a species X :

```
M = LambdaCDM(Lambda = X, name = :w0waCDM)
pars[M.X.w0] = -0.9
pars[M.X.wa] = 0.2
pars[M.X.cs^2] = 1.0
prob = CosmologyProblem(M, pars)
```

We proceed with this new model and problem.

3.3. Computing power spectra

SymBoltz can compute the matter power spectrum $P(k)$ and the angular CMB power spectra C_l^{TT} , C_l^{TE} and C_l^{EE} :

```
using Unitful, UnitfulAstro # for Mpc unit
ks = 10 .^ range(-4, 1, length=200) / u"Mpc"
Ps = spectrum_matter(prob, ks)
ls = 10:10:2000
Cls = spectrum_cmb([:TT, :TE, :EE], prob, ls)
```

Calling `spectrum_matter` and `spectrum_cmb` with `prob` will automatically select a grid of k to solve the perturbations for, and interpolate between them when computing the spectra. This is a common interpolation trick in Boltzmann solvers to integrate fewer perturbation modes. The functions can be called with the solution `sol` instead, but will then interpolate only between the k that `sol` was solved for, leaving it to the user to ensure sufficient sampling density. Figure 3 shows that the spectra agree with CLASS to 1‰-1% or better for the chosen parameters.

3.4. Differentiable Fisher forecasting

Fisher forecasting is a powerful technique for predicting how strong parameter constraints that can be placed by data with given errors. It requires accurate derivatives and is a nice application for automatic differentiation.

Near a peak θ_0 , where derivatives vanish, a log-likelihood function of parameters θ is approximated by the Taylor series

$$\log L(\theta) \approx \log L(\theta_0) - \sum_{i,j} F_{ij}(\theta_i - \bar{\theta}_i)(\theta_j - \bar{\theta}_j), \quad (9)$$

where F is the Fisher (information) matrix with elements

$$F_{ij} = -\frac{1}{2} \frac{\partial^2 \log L}{\partial \theta_i \partial \theta_j} \Big|_{\theta=\theta_0}. \quad (10)$$

Intuitively, F measures how sharp the peak is, or how sensitive $\log L$ is to changes in different directions in parameter space. Fisher forecasting is powered by the *Cramér-Rao bound*, which guarantees that F_{ij}^{-1} is a lower bound $|C_{ij}| \geq |F_{ij}^{-1}|$ for the covariance C_{ij} between model parameters θ_i and θ_j . The inequality is more saturated the better the likelihood approximation (9) is (i.e. the “more Gaussian” the probability distribution is, for which the expansion is exact). Thus, computing F_{ij} at a peak and inverting it gives the tightest parameter constraints one can hope for.

Since F_{ij} depends on derivatives of $\log L$, Fisher forecasting traditionally involves error-prone finite differences and step size tuning. This problem is avoided with automatic differentiation.

To demonstrate differentiable Fisher forecasting with SymBoltz, we make the best possible CMB (TT) measurement: one of $\bar{C}_l$ over the full sky with errors only due to cosmic variance

$$\sigma_l = \sqrt{\frac{2}{2l+1}} \bar{C}_l. \quad (11)$$

Assuming the measurements of each $\bar{C}_l$ are normally distributed and uncorrelated, the log-likelihood for this experiment is χ^2 :

$$\log L(\theta) = -\frac{1}{2} \sum_l \left(\frac{C_l(\theta) - \bar{C}_l}{\sigma_l} \right)^2. \quad (12)$$

In this case, the Fisher matrix (10) becomes

$$F_{ij} = \sum_l \frac{\partial C_l}{\partial \theta_i} \frac{1}{\sigma_l^2} \frac{\partial C_l}{\partial \theta_j} \Big|_{\theta=\theta_0}, \quad (13)$$

evaluated in some fiducial cosmology $\theta = \theta_0$.

The derivatives $\partial C_l / \partial \theta_i$ are hard to compute and perfect candidates for automatic differentiation with ForwardDiff.jl:

```
using ForwardDiff: jacobian
vary = [
    M.g.h, M.c.Ω₀, M.b.Ω₀, M.b.YHe, M.v.Neff,
    M.h.m_eV, M.X.w₀, M.X.wₐ, M.I.ln_As1e10, M.I.ns,
]
genprob = parameter_updater(prob, vary)
ls, ls' = 100:1:1000, 100:25:1000
Cl(θ) = spectrum_cmb(:TT, genprob(θ), ls, ls')
θ₀ = map(par -> pars[par], vary)
dCl_dθ = jacobian(Cl, θ₀)
```

Here `vary` orders the subset of parameters to differentiate with respect to, and `genprob` is a *function* that generates a new problem with updated parameters θ . Then `Cl` specifies a function that computes $C_l(\theta)$ for `ls`, interpolating from a coarser grid `ls'`. This machinery just converts from SymBoltz' parameter-to-value mapping to a function of an array of parameter values, as `jacobian` requires a pure mathematical function $f: \mathbb{R}^a \rightarrow \mathbb{R}^b$ to differentiate. Finally $\partial C_l / \partial \theta_i$ is computed with forward-mode automatic differentiation and stored in `dCl_dθ`, as shown in fig. 4.

It is now trivial to compute the Fisher matrix (13). Inverting it forecasts the constraints in fig. 5. They are in excellent agreement with finite difference results from CLASS, but these required significant tuning of precision parameters and step sizes.

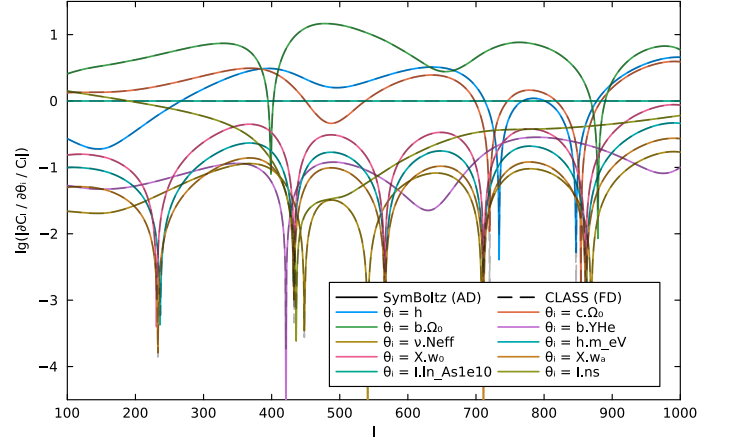


Fig. 4. Normalized derivatives $(\partial C_l / \partial \theta_i) / C_l$ of a CMB TT power spectrum with respect to cosmological parameters θ_i from SymBoltz and automatic differentiation (AD; colored lines) versus CLASS and central finite differences (FD; gray dashes). CLASS uses the precision parameters in section B and finite differences with 5% relative step sizes.

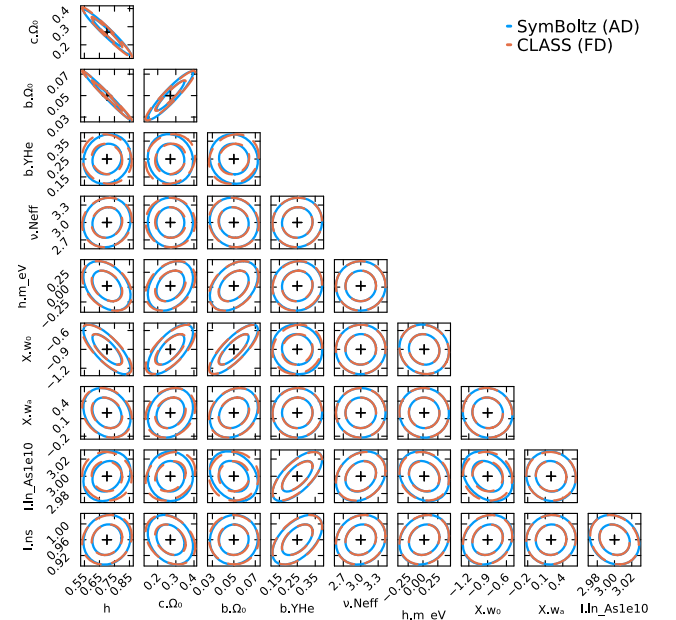


Fig. 5. Marginalized 68% and 95% 2D confidence ellipses for parameter constraints from a Fisher forecast on a cosmic variance-dominated CMB TT-only survey using the derivatives in fig. 4.

3.5. Differentiable MCMC sampling with supernova data

Finally, we show that SymBoltz can output differentiable results for gradient-based Markov chain Monte Carlo (MCMC) samplers, like the No-U-Turn Sampler (NUTS). This state-of-the-art method uses likelihood gradients to move more intelligently in parameter space than the “blind” Metropolis-Hastings algorithm.

We predict apparent magnitudes $m(z)$ of Type Ia supernovae as a function of redshift z and fit them to 1048 observations (z_i, m_i) from the Pantheon dataset (Scolnic et al. 2018). Such supernovae are standard candles with constant absolute magnitude $M \approx -19.3$. Their apparent magnitudes are $m(z) = M + \mu(z)$, where $\mu(z) = 5 \lg(d_L(z)/10 \text{ pc})$ is the distance modulus from

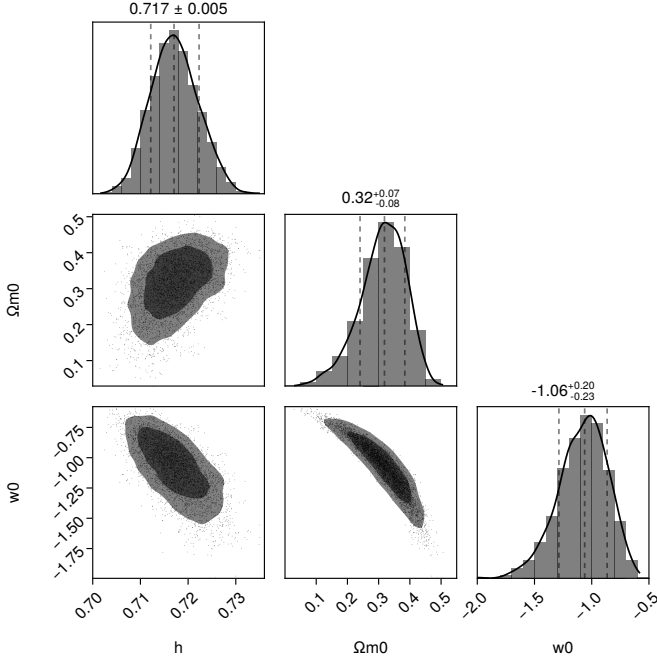


Fig. 6. Parameter inference on 1048 Type Ia supernovae from the full Pantheon supernova dataset, using 5000 MCMC samples with the gradient-based NUTS sampler in Turing.jl and differentiable theory predictions from SymBoltz.jl.

the background-derived luminosity distance¹¹

$$d_L(z) = \frac{c}{H_0} \frac{\chi(z)}{a(z)} \operatorname{sinc} \left(H_0 \sqrt{-\Omega_{k0}} \chi(z) \right). \quad (14)$$

We define the likelihood by a multivariate normal distribution of the 1048 supernovae with (constant) covariance matrix C given in detail by Scolnic et al. (2018). To compute it, we interface SymBoltz with the probabilistic programming framework Turing.jl (Fjelde et al. 2025), which allows a high-level description of the probabilistic model equivalent to the log-likelihood

$$\log L = -\frac{1}{2} \sum_{i,j} (m(z_i) - m_i) C_{ij}^{-1} (m(z_j) - m_j). \quad (15)$$

We use the NUTS sampler in Turing. The code for this example can be found in the paper’s notebook. For the flat w_0 CDM model ($\Omega_{k0} = 0$, $w_a = 0$, fixed Ω_{r0}) it gives the constraints in fig. 6. The results are not interesting, but show that differentiable parameter inference *works* with SymBoltz. Performance improvements are needed to make this viable also for perturbation-derived spectra.

4. Discussion of design synergies and tradeoffs

A symbolic-numeric interface, lack of approximations and differentiability are interesting features in their own rights. They also constitute a self-reinforcing synergy explained in fig. 7. Like any Boltzmann solver, SymBoltz makes some tradeoffs in its design.

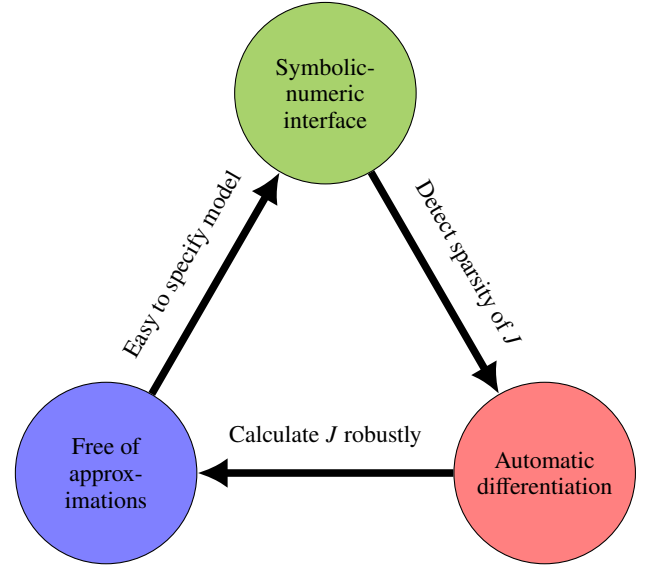


Fig. 7. A symbolic-numeric interface, approximation-freeness and differentiability are three important features of SymBoltz that form a synergy. For example: automatic differentiation helps to calculate the ODE Jacobian robustly; which is needed by implicit solvers to integrate stiff ODEs without approximations; which makes it easier to write down models in a simple symbolic form; which can be used to detect the sparsity pattern of the Jacobian; which in turn increases the efficiency of the implicit ODE solver. This synergy creates a self-reinforcing design.

4.1. Symbolic vs. numeric interface

As explained in sections 1.1 and 2.1, modifications to most Boltzmann solvers are made by editing the low-level numerical source code directly, while changes to SymBoltz are made in a high-level symbolic interface that abstracts away internal details.

We think three properties make the linearized Einstein-Boltzmann equations particularly attractive for symbolic abstraction. First, on the “outside”, they are fundamentally *one* (large) system of equations with a predictable general structure. Second, on the “inside”, they are complicated to solve in multiple stages due to their structure, stiffness and need for speed. Third, they are heavily subject to modifications as the true cosmological model is unknown. To simplify the modeling process, these aspects make it valuable to abstract the “inside” from the “outside”.

Purely numerical codes generally put the *programmer* first. They are adapted to the *computational* structure of the problem. Users have full freedom to tweak problem-specific details, such as implementing complex approximation schemes. Without abstraction layers, the code says exactly what it does. This can be worthwhile for custom-tailoring codes to “workhorse” cosmological models like Λ CDM for maximum performance and demanding MCMC analyses. However, it can result in an overwhelming monolithic design that is hard to read and modify.

The symbolic interface in SymBoltz prioritizes the *modeler*. Its design matches the *physical* structure of the model. Manual chores are automated with compile-time logic on the equations. Users can more easily change equations without understanding internals *provided that* they can be written in a straightforward form that is compatible with the symbolic interface. This is not a major restriction for the Einstein-Boltzmann equations, as they have a plain and predictable structure when linearized and free of approximations. Nevertheless, to bridge this gap, SymBoltz lets users call arbitrary numerical functions that escape the symbolic-to-numeric compilation, if necessary.

¹¹ This expression is valid for any Ω_{k0} with complex $\operatorname{sinc}(x) = \sin(x)/x$, but can be split into branches for positive, negative and zero Ω_{k0} .

This abstraction layer works only if it fits the structure of the underlying equations. A key to envision the design was to look past the *computational* pipeline structure of Boltzmann solvers (e.g. background $\rightarrow$ perturbations $\rightarrow$...), and rather organize the code primarily by *physical* components like the metric, gravitational theory and particle species. Components are self-contained and joined into full cosmological models. As more models are added, the complexity of this modular design stays constant, while monolithic “all-in-one” codes become complicated.

This structure can be unfamiliar to those used to the layout of other codes. The modular structure scales best with a growing number of submodels for individual physical components. But SymBoltz also provides a full unstructured Λ CDM model where everything is merged into one large system, which is easier to work with if one wants to modify arbitrary small parts of the equations. This leaves the choice to the user, who can select between the best of both worlds.

Mixing symbolic and numerical computing can be susceptible to performance problems. Numerics should be fast and symbolics is usually slow, so “perceived performance” suffers if the two are interleaved. But SymBoltz disentangles the expensive symbolic operations in section 2.1 from numerics, so they are performed only before and after performance-critical tasks.

We hope the design of SymBoltz provides a platform for easy exploration of alternative models, regardless of which sector of the equations one is interested in modifying.

4.2. Approximation-freeness vs. performance

As seen in section 1.2, traditional codes reduce stiffness by approximating the equations for explicit integrators, while SymBoltz solves the full stiff system with implicit methods.

Without approximations, SymBoltz needs only one set of equations to solve. These equations are easy to write down in pure symbolic form, which pairs nicely with SymBoltz’ high-level symbolic interface. With approximations, traditional solvers need more complicated infrastructure to switch between versions of the same equations. This fits well to a low-level and fully numerical code that maximizes implementation freedom, like CLASS.

Approximation schemes do not only alleviate stiffness, but also improve performance by reducing the ODE size. However, Lesgourgues & Tram (2011); Hahn et al. (2024) found that the TCA and UFA provide only marginal speedups compared to use of implicit solvers. On the other hand, the RSA can provide significant speedups with high $l_{\max}$ (Lesgourgues & Tram 2011; Moser et al. 2022). However, more load is put on the modeler to derive, implement and validate multiple approximations. This process must generally be repeated for modified models that can easily reintroduce stiffness or invalidate the approximations.

Implicit solvers take more expensive steps than explicit ones. They solve a nonlinear system for the next unknowns at every time step, which gets costly as the ODE size grows (e.g. higher $l_{\max}$). This is often done with Newton’s method, which iterates over linear matrix solutions $Ax = b$, where A involves the ODE Jacobian J . But several factors mitigate this slowdown.

First and foremost, implicit solvers take longer and fewer steps due to better stability properties. This is highly solver-dependent, however: two different implicit solvers can perform very differently on one stiff problem. DifferentialEquations.jl lets SymBoltz easily switch between a large suite of implicit solvers. The background (with stiff thermodynamics) is solved robustly with Rodas4P/Rodas5P, and the perturbations efficiently with TRBDF2/KenCarp4/Rodas5P for low/medium/high precision. Notably, CLASS includes and defaults to a custom implicit solver

ndf15 and performs great (Lesgourgues & Tram 2011). In contrast, SymBoltz can choose between a range of compatible solvers to fulfill different performance and precision requirements.

Second, a key optimization for implicit solvers is to compute J efficiently, reuse it over time steps and LU-factorize A to speed up successive linear system solutions until Newton’s method no longer converges due to outdated Jacobians. The TRBDF2 and KenCarp4 solvers do this. When needed, J is updated with the method in section 2.2, which is cheaper than finite differences. The linear matrix solver used in Newton’s method also matters, as they are among the most thoroughly optimized numerical algorithms. DifferentialEquations.jl interoperates with LinearSolve.jl, which lets SymBoltz easily swap the linear solver in the ODE solver. On our hardware we find that performance improves by 5 $\times$ with a recursive LU factorization algorithm from RecursiveFactorization.jl or Intel’s (proprietary) Math Kernel Library MTK.jl over Julia’s default OpenBLAS backend.

Third, sparse matrix methods can speed up the linear solver as the system grows in size and the fraction of zeros increases. SymBoltz can find the exact sparsity pattern from the symbolic Jacobian in section 2.1.5, and LinearSolve.jl provides several sparse matrix methods. Support for this is still ongoing work.

Although SymBoltz sacrifices some performance to get rid of approximations, these countermeasures make the impact less severe than one might fear. In return, the approximation-free structure is a major simplification and pairs well with the high-level interface for simple symbolic equations. There is still room to improve performance without resorting to approximations by tuning precision tolerances, supporting sparse matrix methods, exploring implicit-explicit (IMEX) solvers (e.g. Kennedy & Carpenter (2003)) and smarter sampling of k and l , for example.

Further performance optimizations and comparisons are left for future work. SymBoltz is fast enough for single runs and interactive usage, but not yet for MCMC analyses with perturbation-derived quantities. Of course, SymBoltz can also implement approximation schemes in the future, but its symbolic nature can make this harder than in other codes. As a counterweight to other codes, SymBoltz’ primary mission will be to solve full stiff equations and keep approximations optional and secondary.

4.3. Forward-mode vs. reverse-mode automatic differentiation

At this time SymBoltz is only well-tested with forward-mode automatic differentiation. However, we saw in section 1.3 that reverse-mode is more attractive for applications with more inputs than outputs. This is the case for popular applications with scalar likelihood or loss functions, such as MCMC parameter inference and training neural network emulators. Reverse-mode could be particularly powerful for parameter inference with next-generation surveys, where experimental nuisance parameters must also be sampled in large $O(100)$ -dimensional parameter spaces, even if eventually marginalized over. However, forward-mode has better characteristics when performing sensitivity analyses. For example, computing $\partial P(k; \theta)/\partial \theta_i$ or $\partial C_l(\theta)/\partial \theta_i$ (e.g. Fisher forecasting in section 3.4) is faster with forward-mode, since one typically wants $O(100-1000)$ k or l , but only has $O(10)$ parameters.

Of course, automatic differentiation is just an additional feature. One can run the code with or without automatic differentiation, or use finite differences instead. It does not pose any trade-off, but robust support for both forward-mode and reverse-mode remains a future goal that would make SymBoltz more powerful for different applications. This situation will improve as the differentiable Julia ecosystem continues to evolve.

5. Conclusion and future potential

SymBoltz is a fresh Julia package for solving the linearized Einstein-Boltzmann equations. It relaxes all approximation switching schemes found in other codes and solves a single set of stiff equations at all times with implicit integrators, and combats the performance loss with modern and efficient implicit ODE solvers, optimized linear system solvers and (soon) sparse matrix methods. It is differentiable, so one can get accurate derivatives of any output quantity with respect to any input parameter. This enables modelers to rapidly prototype new models by straightforwardly writing down new variables and equations.

Version 0.8.1 features the metric in the conformal Newtonian gauge, General Relativity, cold dark matter, photons, baryons and RECFast recombination, massless and massive neutrinos, the cosmological constant and $w_0 w_a$ dark energy, and computes luminosity distances and matter and CMB spectra. It also has some rudimentary models for Brans-Dicke gravity, quintessence dark energy and curved geometry, but these are not complete yet. We think the modular design makes it very easy to add new models and compute other quantities.

Forward-mode automatic differentiation is well-tested, while support for reverse-mode is a future goal. This would make SymBoltz very powerful for scalar loss applications like MCMCs.

With some more work, SymBoltz could grow into a fully integrated symbolic-numeric and differentiable cosmological modeling environment. For example, numerical code generation could be extended from linear equations to non-linear N -body simulations in a consistent and unified framework. This could alleviate the N -language problem ($N \geq 2$) prevalent in cosmological modeling, which glues together convenient high-level languages like Python with performant low-level languages like C and Fortran, and sometimes symbolic work in a CAS like Mathematica.

SymBoltz shows that it is possible to create a symbolic-numeric, approximation-free and differentiable Boltzmann solver with modern numerical techniques. It excels at model simplicity and ease of modification. More work is needed to make it as feature-complete and fast as CAMB and CLASS that have been refined over many years. A major goal is to compute values and derivatives of perturbation-derived spectra fast enough for use with gradient-based MCMC samplers. Implicit ODE solvers, nonlinear and linear equation solvers, sparse matrix methods and differentiability are all demanding numerical techniques. Making them work robustly and efficiently together is challenging, but the current state shows promise. SymBoltz is built on (and has contributed to) several evolving Julia packages, and will continue to grow both on its own and with improvements to its dependencies.

SymBoltz is easy to install from <https://github.com/hersle/SymBoltz.jl>. Documentation is available there, and the code is tested with continuous integration to ensure it remains correct and up-to-date (see section C). **Anyone is encouraged to ask questions, give feedback, open issues and contribute pull requests in the repository!** We hope SymBoltz offers valuable competition and new ideas on the Boltzmann solver market.

Acknowledgments

I thank Aayush Sabharwal and Christopher Rackauckas for developing and maintaining ModelingToolkit.jl and DifferentialEquations.jl and answering questions. I thank Hans A. Winther for helpful suggestions, testing the code and giving feedback on it and this manuscript. I thank Julien Lesgourgues, Thomas Tram and others for developing CLASS, which has inspired SymBoltz.

References

- Bezanson, J., Edelman, A., Karpinski, S., & Shah, V. B. 2017, SIAM Rev., 59, 65 [arXiv:1411.1607]
- Blas, D., Lesgourgues, J., & Tram, T. 2011, JCAP, 07, 034 [arXiv:1104.2933]
- Bull, P., Akrami, Y., Adamek, J., et al. 2016, Physics of the Dark Universe, 12, 56 [arXiv:1512.05356]
- Casas, S., Fidler, C., Bolliet, B., Villaescusa-Navarro, F., & Lesgourgues, J. 2025, ArXiv [arXiv:2508.05728]
- Chevallier, M., & Polarski, D. 2001, Int. J. Mod. Phys. D, 10, 213 [arXiv:gr-qc/0009008]
- de Putter, R., Huterer, D., & Linder, E. V. 2010, Phys. Rev. D, 81, 103513 [arXiv:1002.1311]
- DESI Collaboration. 2016, ArXiv [arXiv:1611.00036]
- Dewdney, P. E., Hall, P. J., Schilizzi, R. T., & Lazio, T. J. L. W. 2009, Proceedings of the IEEE, 97, 1482 [DOI:10.1109/JPROC.2009.2021005]
- Doran, M. 2005a, JCAP, 10, 011 [arXiv:astro-ph/0302138]
- Doran, M. 2005b, JCAP, 06, 011 [arXiv:astro-ph/0503277]
- Euclid Collaboration. 2025, A&A, 697, A1 [arXiv:2405.13491]
- Fjelde, T. E., Xu, K., Widmann, D., et al. 2025, ACM Trans. Probab. Mach. Learn., 1, 1 [DOI:10.1145/3711897]
- Freedman, W. L. 2017, Nat Astron, 1, 0121 [arXiv:1706.02739]
- Gowda, S., Ma, Y., Cheli, A., et al. 2022, ArXiv [arXiv:2105.03949]
- Griewank, A. & Walther, A. 2008, Evaluating Derivatives, 2nd edn., Other Titles in Applied Mathematics (Society for Industrial and Applied Mathematics)
- Hahn, O., List, F., & Porqueres, N. 2024, JCAP, 06, 063 [arXiv:2311.03291]
- Hastings, W. K. 1970, Biometrika, 57, 97 [DOI:10.1093/biomet/57.1.97]
- Hoffman, M. D. & Gelman, A. 2011, ArXiv [arXiv:1111.4246]
- Kennedy, C. A. & Carpenter, M. H. 2003, Applied Numerical Mathematics, 44, 139 [DOI:10.1016/S0168-9274(02)00138-1]
- Lesgourgues, J. 2011a, ArXiv [arXiv:1104.2932]
- Lesgourgues, J. 2011b, ArXiv [arXiv:1104.2934]
- Lesgourgues, J. & Tram, T. 2011, JCAP, 09, 032 [arXiv:1104.2935]
- Lewis, A. 2025, CAMB Notes [https://cosmologist.info/notes/CAMB.pdf]
- Lewis, A., Challinor, A., & Lasenby, A. 2000, ApJ, 538, 473 [arXiv:astro-ph/9911177]
- Li, Z., Sullivan, J., & Millea, M. 2023 [DOI:10.5281/zenodo.10065126]
- Linder, E. V. 2003, Phys. Rev. Lett., 90, 091301 [arXiv:astro-ph/0208512]
- LSST Science Collaboration. 2009, ArXiv [arXiv:0912.0201]
- Ma, C.-P. & Bertschinger, E. 1995, ApJ, 455, 7 [arXiv:astro-ph/9506072]
- Ma, Y., Gowda, S., Anantharaman, R., et al. 2022, ArXiv [arXiv:2103.05244]
- Moser, B., Lorenz, C. S., Schmitt, U., et al. 2022, Astronomy and Computing, 40, 100603 [arXiv:2112.08395]
- Peebles, P. J. E. & Yu, J. T. 1970, ApJ, 162, 815 [DOI:10.1086/150713]
- Piras, D., Polanska, A., Mancini, A. S., Price, M. A., & McEwen, J. D. 2024, The Open Journal of Astrophysics, 7 [arXiv:2405.12965]
- Rackauckas, C. & Nie, Q. 2017, Journal of Open Research Software, 5 [DOI:10.5334/jors.151]
- Refregier, A., Gamper, L., Amara, A., & Heisenberg, L. 2017, ArXiv [arXiv:1708.05177]
- Revels, J., Lubin, M., & Papamarkou, T. 2016, ArXiv [arXiv:1607.07892]
- Scolnic, D. M., Jones, D. O., Rest, A., et al. 2018, ApJ, 859, 101 [arXiv:1710.00845]
- Scott, D. & Moss, A. 2009, MNRAS, 397, 445 [arXiv:0902.3438]
- Seager, S., Sasselov, D. D., & Scott, D. 1999, ApJ, 523, L1 [arXiv:astro-ph/9909275]
- Seager, S., Sasselov, D. D., & Scott, D. 2000, ApJS, 128, 407 [arXiv:astro-ph/9912182]
- Seljak, U., Aslanyan, G., Feng, Y., & Modi, C. 2017, JCAP, 12, 009 [arXiv:1706.06645]
- Seljak, U. & Zaldarriaga, M. 1996, ApJ, 469, 437 [arXiv:astro-ph/9603033]
- Steinebach, G. 1995, Technische Hochschule Darmstadt [https://pub.h-brs.de/frontdoor/index/index/docId/1548]
- The Simons Observatory Collaboration. 2019, JCAP, 02, 056 [arXiv:1808.07445]
- Wong, W. Y., Moss, A., & Scott, D. 2008, MNRAS, 386, 1023 [arXiv:0711.1357]

Appendix A: List of equations and practical implementation details

This appendix summarizes the equations that define the standard Λ CDM model in SymBoltz and comments on their practical implementation. We hope this can be a useful reference for others. The list is structured like SymBoltz with one component per subsection. Variables are in units where “ $G = c = H_0 = 1$ ” and temperatures are in K, unless otherwise is stated. These units are chosen because G , c and H_0 can be divided out from the Einstein equations as natural units, but this requires some conversion in the recombination equations that depend explicitly on H_0 . In other words, times are in units of $1/H_0$, distances in c/H_0 and masses in $c^3/H_0 G$. When G , c or H_0 appear explicitly in the equations, they are in SI units and used only to convert from SI units into the dimensionless units. The equations closely follow the conventions in the seminal paper by [Ma & Bertschinger \(1995\)](#) and very closely matches the source code of SymBoltz with Unicode characters. It is far outside the scope of this paper to derive the equations and explain the meaning of every variable.

The independent variable is conformal time τ , and all derivatives $' = d/d\tau$ are with respect to it (in units of $1/H_0$). By default, integration starts from the early time $\tau = \tau_i = 10^{-6}$, and is terminated when the scale factor crosses $a = 1$, and the corresponding time $\tau = \tau_0$ is labeled today.

A.1. Metric and spacetime (g)

SymBoltz is currently only formulated in the conformal Newtonian gauge, with this metric and related quantities:

$$g_{0i} = g_{i0} = -a^2(1 + 2\Psi)\delta_{0i}, \quad g_{ij} = a^2(1 - 2\Phi)\delta_{ij}, \quad z = \frac{1}{a} - 1, \quad \mathcal{H} = \frac{a'}{a}, \quad H = \frac{\mathcal{H}}{a}, \quad \chi = \tau_0 - \tau.$$

Here $\mathcal{H}$ and H are the conformal and cosmic Hubble factors (in units where $\mathcal{H}_0 = H_0 = 1$). The scale factor a is related to redshift z , and χ is the lookback time from today that appears in some integral solutions. SymBoltz is currently restricted to a flat spacetime.

A.2. General relativity (G)

The gravitational theory of the Λ CDM model is General Relativity governed by the Einstein field equations $G_{\mu\nu} = 8\pi T_{\mu\nu}$. By default, SymBoltz solves for the metric variables a , Φ and Ψ with $(\mu, \nu) = (0, 0)$ in the background (1st Friedmann equation) and $(\mu, \nu) = \{(0, 0), (i, j)\}$ in the perturbations:

$$a' = \sqrt{\frac{8\pi}{3}} \rho a^2, \quad \Phi' = -\mathcal{H}\Psi - \frac{k^2}{3\mathcal{H}}\Phi - \frac{4\pi}{3} \frac{a^2}{\mathcal{H}} \delta\rho, \quad \Psi = -\Phi - 12\pi \left(\frac{a}{k}\right)^2 \Pi.$$

Note that it is also possible to evolve other redundant combinations of the Einstein equations (such as the acceleration equation). The equations are coupled to total densities ρ , $\delta\rho$, pressures P and anisotropic stresses Π for an *arbitrary* set of species s that are present in the cosmological model:

$$\rho = \sum_s \rho_s, \quad P = \sum_s P_s, \quad \delta\rho = \sum_s \delta\rho_s = \sum_s \delta_s \rho_s, \quad \delta P = \sum_s \delta P_s = \sum_s \delta_s P_s c_{s,s}^2, \quad \Pi = \sum_s \Pi_s = \sum_s (\rho_s + P_s) \sigma_s.$$

We emphasize that the gravity component is completely unaware of all particle species and makes no assumptions about them. It only reacts to total stress-energy components. All species *must* therefore define ρ , P , $\delta\rho$, δP and Π explicitly, even if zero. This requirement is somewhat pedantic, but helps isolate components from each other for greater reuse when composing models.

The scale factor a is initialized as the nonlinear solution of the Friedmann equation constrained to $\mathcal{H} = 1/\tau$ (motivated by its radiation-dominated solution $a = \sqrt{\Omega_{r0}} \tau$). The constraint potential is initialized to $\Psi = 20C/(15 + 4f_\nu)$ with the (arbitrary) integration constant $C = 1/2$ and initial energy density fraction $f_\nu = (\rho_\nu + \rho_h)/(\rho_\nu + \rho_h + \rho_\gamma)$ of all (massless and massive) neutrino species relative to all species that are radiative at early times. The evolved potential Φ is initialized accordingly from the constraint equation (the found solution is close to $\Phi = (1 + 2f_\nu/5)\Psi$, but providing both Φ and Ψ explicitly leads to overdetermined initialization equations due to the constraint equation).

The next sections present the Λ CDM section of the “library of species” that are available in SymBoltz.

A.3. Cold dark matter (c)

Cold dark matter is a non-relativistic and non-interacting species that follows very simple equations:

$$w = 0, \quad c_s^2 = w, \quad P = 0, \quad \rho = \frac{\rho_0}{a^3}, \quad \delta' = -\theta + 3\Phi', \quad \theta' = -\mathcal{H}\theta + k^2\Psi, \quad u = \frac{\theta}{k}, \quad \sigma = 0.$$

Initial conditions are adiabatic with $\delta/(1 + w) = -3\Psi/2$ and $\theta = k^2\tau\Psi/2$. The species is parametrized by the reduced density $\Omega_0 = \frac{8\pi}{3} \rho_0$ today.

A.4. Baryons (b)

Baryons are also non-relativistic, but interact with photons through Compton scattering and are subject to recombination physics. This significantly complicates their behavior. SymBoltz currently implements equations from RECFAST¹² version 1.5.2 (Seager et al. 1999, 2000; Wong et al. 2008; Scott & Moss 2009):

$$\begin{aligned}
w &= 0, & P &= 0, & \rho &= \frac{\rho_0}{a^3}, & f_{\text{He}} &= \frac{Y_{\text{He}}}{\frac{m_{\text{He}}}{m_{\text{H}}}(1 - Y_{\text{He}})}, & n_{\text{H}} &= \frac{(1 - Y_{\text{He}})\rho}{m_{\text{H}}}, & n_{\text{He}} &= f_{\text{He}}n_{\text{H}}, & c_s^2 &= \frac{k_B}{\mu c^2} \left(T_b - \frac{T'_b}{3\mathcal{H}} \right), \\
\beta &= \frac{1}{k_B T_b}, & T'_b &= -2\mathcal{H}T_b - \frac{a}{H_0} \frac{8}{3} \frac{T_\gamma^4 X_e}{1 + f_{\text{He}} + X_e} (T_b - T_\gamma), & \mu &= \frac{m_{\text{H}}}{1 + \left(\frac{m_{\text{H}}}{m_{\text{He}}} - 1 \right) Y_{\text{He}} + (1 - Y_{\text{He}}) X_e}, & \kappa' &= -\frac{a}{H_0} n_e \sigma_T c, \\
v &= -\kappa' e^{-\kappa}, & n_e &= X_e n_{\text{H}}, & X_e &= X_{\text{H}}^+ + X_{\text{He}}^{++} + f_{\text{He}} X_{\text{He}}^+ + X_{\text{e}}^{\text{re}_1} + X_{\text{e}}^{\text{re}_2}, & X_{\text{H}}^+ &= -\frac{a}{H_0} C_{\text{H}} \left(\alpha_{\text{H}} n_e X_{\text{H}}^+ - \beta_{\text{H}} e^{-\beta_b E_{\text{H}}^{2s,1s}} (1 - X_{\text{H}}^+) \right), \\
X_{\text{He}_1}^+ &= -\frac{a}{H_0} C_{\text{He}_1} \left(\alpha_{\text{He}_1} n_e X_{\text{He}_1}^+ - \beta_{\text{He}_1} e^{-\beta_b E_{\text{He}_1}^{2s,1s}} (1 - X_{\text{He}_1}^+) \right), & X_{\text{He}_3}^+ &= -\frac{a}{H_0} C_{\text{He}_3} \left(n_e \alpha_{\text{He}_3} X_{\text{He}_3}^+ - 3\beta_{\text{He}_3} e^{-\beta_b E_{\text{He}_3}^{2s,1s}} (1 - X_{\text{He}_3}^+) \right), \\
X_{\text{He}}^+ &= X_{\text{He}_1}^+ + X_{\text{He}_3}^+, & X_{\text{He}}^{++} &= \frac{2f_{\text{He}} R_{\text{He}}^+}{\left(1 + f_{\text{He}} + R_{\text{He}}^+ \right) \left(1 + \sqrt{1 + \frac{4f_{\text{He}} R_{\text{He}}^+}{(1 + f_{\text{He}} + R_{\text{He}}^+)^2}} \right)}, & R_{\text{He}^+} &= \frac{\exp(-\beta E_{\text{He}^+}^{\infty,1s})}{n_{\text{H}} \lambda_{\text{e}}^3}, & \lambda_{\text{e}} &= \frac{h}{\sqrt{2\pi m_e \beta}}, \\
X_{\text{e}}^{\text{re}_1} &= \frac{1 + f_{\text{He}}}{2} + \frac{1 + f_{\text{He}}}{2} \tanh \left(\frac{4}{3} \frac{(1 + z^{\text{re}_1})^{3/2} - (1 + z)^{3/2}}{(1 + z^{\text{re}_1})^{1/2}} \right), & X_{\text{e}}^{\text{re}_2} &= \frac{f_{\text{He}}}{2} + \frac{f_{\text{He}}}{2} \tanh \left(\frac{4}{3} \frac{(1 + z^{\text{re}_2})^{3/2} - (1 + z)^{3/2}}{(1 + z^{\text{re}_2})^{1/2}} \right), \\
\delta' &= -\theta - 3\mathcal{H}c_s^2 \delta + 3\Phi', & \theta' &= -\mathcal{H}\theta + k^2 c_s^2 \delta + k^2 \Psi - \frac{4}{3} \kappa' \frac{\rho_\gamma}{\rho_b} (\theta_\gamma - \theta_b), & u &= \frac{\theta}{k}, & \sigma &= 0.
\end{aligned}$$

Transition rates and coefficients related to recombination of Hydrogen include fitting functions that emulate the results of more accurate and expensive computations (here $\ln(a)$ is the logarithm of the scale factor, while (Fa) is an unrelated fudge factor):

$$\begin{aligned}
\alpha_{\text{H}} &= 10^{-19} (Fa) \frac{\left(\frac{T_b}{T_0} \right)^b}{1 + c \left(\frac{T_b}{T_0} \right)^d}, & \beta_{\text{H}} &= \frac{\alpha_{\text{H}}}{\lambda_{\text{e}}^3} \exp(-\beta E_{\text{H}}^{\infty,2s}), \\
K_{\text{H}} &= \left(1 + A_1 \exp \left(-\left(\frac{\ln(a) - \ln(a_1)}{w_1} \right)^2 \right) + A_2 \exp \left(-\left(\frac{\ln(a) - \ln(a_2)}{w_2} \right)^2 \right) \right) \frac{(\lambda_{\text{H}}^{2s,1s})^3}{8\pi H}, & C_{\text{H}} &= \frac{1 + K_{\text{H}} \Lambda_{\text{H}} n_{\text{H}} (1 - X_{\text{H}}^+)}{1 + K_{\text{H}} (\Lambda_{\text{H}} + \beta_{\text{H}}) n_{\text{H}} (1 - X_{\text{H}}^+)}.
\end{aligned}$$

Helium rates and coefficients are even more complicated. First, Helium includes contributions from singlet states (He_1):

$$\begin{aligned}
\alpha_{\text{He}_1} &= \frac{q_1}{\sqrt{\frac{T_b}{T_2}} \left(1 + \sqrt{\frac{T_b}{T_2}} \right)^{1-p_1} \left(1 + \sqrt{\frac{T_b}{T_1}} \right)^{1+p_1}}, & \beta_{\text{He}_1} &= 4 \frac{\alpha_{\text{He}_1}}{\lambda_{\text{e}}^3} \exp(-E_{\text{He}_1}^{\infty,2s}), & K_{\text{He}_1} &= \frac{1}{K_{\text{He}_1^0}^{-1} + K_{\text{He}_1^1}^{-1} + K_{\text{He}_1^2}^{-1}}, \\
K_{\text{He}_1^0}^{-1} &= \frac{8\pi H}{(\lambda_{\text{He}_1}^{2p,1s})^3}, & K_{\text{He}_1^1}^{-1} &= -\exp(-\tau_{\text{He}_1}) K_{\text{He}_1^0}^{-1}, & K_{\text{He}_1^2}^{-1} &= \frac{A_{2p_1}}{3(1 + 0.36 \gamma_{2p_1}^{0.86}) n_{\text{He}} (1 - X_{\text{He}}^+)}, & \tau_{\text{He}_1} &= \frac{3A_{2p_1} n_{\text{He}} (1 - X_{\text{He}}^+)}{K_{\text{He}_1^0}^{-1}}, \\
\gamma_{2p_1} &= \frac{3A_{2p_1} f_{\text{He}} c^2 (1 - X_{\text{He}}^+)}{8\pi \sigma_{\text{He}_1} \sqrt{\frac{2\pi}{\beta m_{\text{He}} c^2}} (f_{\text{He}_1}^{2p,1s})^3 (1 - X_{\text{H}}^+)}, & C_{\text{He}_1} &= \frac{\exp(-\beta E_{\text{He}_1}^{2p,2s}) + K_{\text{He}_1} \Lambda_{\text{He}_1} n_{\text{He}} (1 - X_{\text{He}}^+)}{\exp(-\beta E_{\text{He}_1}^{2p,2s}) + K_{\text{He}_1} (\Lambda_{\text{He}_1} + \beta_{\text{He}_1}) n_{\text{He}} (1 - X_{\text{He}}^+)}.
\end{aligned}$$

Second, Helium also includes contributions from triplet states (He_3):

$$\begin{aligned}
\alpha_{\text{He}_3} &= \frac{q_3}{\sqrt{\frac{T_b}{T_2}} \left(1 + \sqrt{\frac{T_b}{T_2}} \right)^{1-p_3} \left(1 + \sqrt{\frac{T_b}{T_1}} \right)^{1+p_3}}, & \beta_{\text{He}_3} &= \frac{4}{3} \frac{\alpha_{\text{He}_3}}{\lambda_{\text{e}}^3} \exp(-\beta E_{\text{He}_3}^{\infty,2s}), & \tau_{\text{He}_3} &= \frac{3A_{2p_3} n_{\text{He}} (1 - X_{\text{He}}^+) (\lambda_{\text{He}_3}^{2p,1s})^3}{8\pi H}, \\
\gamma_{2p_3} &= \frac{3A_{2p_3} f_{\text{He}} c^2 (1 - X_{\text{He}}^+)}{8\pi \sigma_{\text{He}_3} \sqrt{\frac{2\pi}{\beta m_{\text{He}} c^2}} (f_{\text{He}_3}^{2p,1s})^3 (1 - X_{\text{H}}^+)}, & C_{\text{He}_3} &= \frac{A_{2p_3} \left(\frac{1 - \exp(-\tau_{\text{He}_3})}{\tau_{\text{He}_3}} + \frac{1}{3(1 + 0.66 \gamma_{2p_3}^{0.9})} \right) \exp(-\beta E_{\text{He}_3}^{2p,2s})}{A_{2p_3} \left(\frac{1 - \exp(-\tau_{\text{He}_3})}{\tau_{\text{He}_3}} + \frac{1}{3(1 + 0.66 \gamma_{2p_3}^{0.9})} \right) \exp(-\beta E_{\text{He}_3}^{2p,2s}) + \beta_{\text{He}_3}}.
\end{aligned}$$

Every variable that does not occur on the left side of an equation is either a constant or a parameter. This includes Y_{He} , fudge factors and wavenumbers, frequencies and energies for atomic transitions. Some important variables are the baryon temperature T_b , photon

¹² <https://www.astro.ubc.ca/people/scott/recfast.html>

temperature T_γ , mean molecular weight μ , baryon sound speed c_s^2 , optical depth κ , visibility function v and the free electron fraction X_e (conventionally relative to Hydrogen, so $X_e > 1$ in presence of Helium). Please consult the code and RECFAST references cited above for more details.

Unlike other RECFAST implementations, SymBoltz does not approximate the stiff Peebles equations at early times by Saha approximations (although X_{He}^{++} is given by a Saha equation at *all* times). This is not necessary with a good implicit ODE solver. SymBoltz sets $C_{\text{H}} = C_{\text{He}} = 1$ when $X_e \gtrsim 0.99$ to avoid numerical instability at early times. Atomic calculations are done in SI units and converted to SymBoltz' dimensionless units by factors of H_0 in SI units. The differential equation for T'_b is very stiff and sensitive to $T_b - T_\gamma$, but $T_b \approx T_\gamma$ in the early universe, so we rewrite it to a more stable differential equation for $\Delta T' = T'_b - T'_\gamma$ instead, initialize $\Delta T = 0$ and observe $T_b = \Delta T + T_\gamma$. The optical depth $\kappa(\tau) = \int_{\tau_0}^{\tau} \kappa'(\tau') d\tau'$ is really a line-of-sine integral into the past, but is integrated together with the background ODEs by initializing $\kappa(\tau_i) = 0$ to an arbitrary value, integrating the differential equation for κ' and subtracting the final value of $\kappa(\tau_0)$ (i.e. $\int_{\tau_0}^{\tau} = \int_{\tau_0}^{\tau_i} + \int_{\tau_i}^{\tau} = \int_{\tau_i}^{\tau} - \int_{\tau_i}^{\tau_0}$). There is no tight-coupling approximation.

Initial conditions are full ionization $X_{\text{H}}^+ = X_{\text{He}}^+ = 1$, thermal equilibrium $T_b = T_\gamma$ ($\Delta T = 0$), the arbitrary $\kappa = 0$, and adiabatic perturbations $\delta/(1+w) = -3\Psi/2$ and $\theta = k^2\tau\Psi/2$. The baryon species is parametrized by the reduced density $\Omega_0 = \frac{8\pi}{3}\rho_0$ today and the primordial Helium mass fraction Y_{He} .

SymBoltz solves thermodynamics equations together with the background equations, while some other codes treat these as separate stages. There is no meaningful performance improvement from doing this, as the size of the background (and thermodynamics) ODEs is so small. This makes a clear distinction between the background with all 0th order equations of motion, and the perturbations with all 1st order equations. It also makes it easy to create exotic models where the thermodynamics couple to the background, for example.

Note that RECFAST uses fitting functions to emulate the results of more physically accurate and expensive simulations. These are tuned to work for the Λ CDM model. SymBoltz would therefore benefit from including more physically accurate recombination models for safer use with modified models.

A.5. Photons (γ)

Photons are massless and therefore ultra-relativistic. Unlike non-relativistic particles, one must account for the direction $\cos\theta = \mathbf{p} \cdot \mathbf{k}/|\mathbf{p}||\mathbf{k}|$ of their momenta $\mathbf{p}$ relative to the Fourier wavenumber $\mathbf{k}$. This results in a theoretically infinite hierarchy of equations for Legendre multipoles l , which in practice must be truncated at some maximum multipole l_{max} :

$$\begin{aligned} T &= \frac{T_0}{a}, & w &= \frac{1}{3}, & c_s^2 &= w, & P &= \frac{\rho}{3}, & \rho &= \frac{\rho_0}{a^4}, & \delta &= F_0, & \theta &= \frac{3}{4}kF_1, & u &= \frac{\theta}{k}, & \sigma &= \frac{F_2}{2}, \\ F'_0 &= -kF_1 + 4\Phi', & F'_1 &= \frac{k}{3}(F_0 - 2F_2 + 4\Psi) + \frac{4}{3}\frac{\kappa'}{k}(\theta_\gamma - \theta_b), \\ F'_l &= \frac{k}{2l+1}(lF_{l-1} - (l+1)F_{l+1}) + F_l\kappa' - \delta_{l,2}\frac{\kappa'}{10}\Pi, & F'_{l_{\text{max}}} &= kF_{l_{\text{max}}-1} - \frac{l_{\text{max}}+1}{\tau}F_{l_{\text{max}}} + \kappa'F_{l_{\text{max}}}, \\ G'_0 &= -kG_1 + \kappa'G_0 - \frac{\kappa'}{2}\Pi, & G'_l &= \frac{k}{2l+1}(lG_{l-1} - (l+1)G_{l+1}) + \kappa'G_l - \delta_{l,2}\frac{\kappa'}{10}\Pi, \\ G'_{l_{\text{max}}} &= kG_{l_{\text{max}}-1} - \frac{l_{\text{max}}+1}{\tau}G_{l_{\text{max}}} + \kappa'G_{l_{\text{max}}}, & \Pi &= F_2 + G_0 + G_2, & \Theta_l &= \frac{F_l}{4}. \end{aligned}$$

The equations for F'_l and G'_l apply for $2 \leq l < l_{\text{max}}$. There are no tight-coupling, radiation-streaming or ultra-relativistic fluid approximations. Initial conditions are adiabatic with $F_0 = -2\Psi$ (i.e. $\delta/(1+w) = -\frac{3}{2}\Psi$), $F_1 = \frac{2}{3}k\tau\Psi$ (i.e. $\theta = \frac{1}{2}k^2\tau\Psi$), $F_2 = -\frac{8}{15}\frac{k}{\kappa'}F_1$, $G_0 = \frac{5}{16}F_2$, $G_1 = -\frac{1}{16}\frac{k}{\kappa'}F_2$, $G_2 = \frac{1}{16}F_2$, and $F_l = -\frac{l}{2l+1}\frac{k}{\kappa'}F_{l-1}$ and $G_l = -\frac{l}{2l+1}\frac{k}{\kappa'}G_{l-1}$ for $3 \leq l \leq l_{\text{max}}$. The species is parametrized by its temperature T_0 today, which in turn sets the density parameters $\Omega_0 = \frac{\pi^2}{15}\frac{(k_B T_0)^4}{(\hbar c)^3}\frac{8\pi G}{3H_0^2}$ and $\rho_0 = \frac{8\pi}{3}\Omega_0$ today.

A.6. Massless neutrinos (ν)

Massless neutrinos behave similarly to photons, but decouple from interactions with other species in the very early universe. One must only account for this interaction in initial conditions, while their evolution equations are a simpler case of the photons':

$$\begin{aligned} T &= \frac{T_0}{a}, & w &= \frac{1}{3}, & c_s^2 &= \frac{1}{3}, & P &= \frac{\rho}{3}, & \rho &= \frac{\rho_0}{a^4}, & \delta &= F_0, & \theta &= \frac{3}{4}kF_1, & \sigma &= \frac{F_2}{2}, \\ F'_0 &= -kF_1 + 4\Phi', & F'_1 &= \frac{k}{3}(F_0 - 2F_2 + 4\Psi), & F'_l &= \frac{k}{2l+1}(lF_{l-1} - (l+1)F_{l+1}), & F'_{l_{\text{max}}} &= kF_{l_{\text{max}}-1} - \frac{l_{\text{max}}+1}{\tau}F_{l_{\text{max}}}. \end{aligned}$$

The equations for F'_l apply for $2 \leq l < l_{\text{max}}$. There is no ultra-relativistic fluid approximation. Initial conditions are adiabatic with $F_0 = -2\Psi$ (i.e. $\delta/(1+w) = -\frac{3}{2}\Psi$), $F_1 = \frac{2}{3}k\tau\Psi$ (i.e. $\theta = \frac{1}{2}k^2\tau\Psi$), $F_2 = \frac{2}{15}(k\tau)^2\Psi$ and $F_l = \frac{l}{2l+1}k\tau F_{l-1}$. The species is parametrized by the effective number N_{eff} , the reduced density $\Omega_0 = \frac{8\pi}{3}\rho_0$ and temperature T_0 today. If photons are present, they default to $T_{\nu 0} = (\frac{4}{11})^{1/3}T_{\gamma 0}$ and $\Omega_{\nu 0} = N_{\text{eff}}\frac{7}{8}(\frac{4}{11})^{4/3}\Omega_{\gamma 0}$.

A.7. Massive neutrinos (h)

Massive neutrinos are the most complicated species in the Λ CDM model (alongside baryon recombination). In essence, the species we have looked at so far have Boltzmann equations where the momenta of their distribution function can be integrated out *analytically* in non-relativistic and ultra-relativistic limits. This means that their stress-energy components are linked by trivial equations of state and sound speeds, for example, and their effect can be parametrized by a simple density parameter Ω_0 .

On the other hand, massive neutrinos have intermediate masses that fall between the non-relativistic and ultra-relativistic limits. Integrals over their distribution function must be computed *numerically*. This is very expensive if done naively, and it is extremely important to choose a quadrature scheme that minimizes the number of sampled points. Fortunately, the momentum integrals have a structure that can be exploited: they are all in the form weighted form $I[g(x)] = \int_0^\infty dx x^2 f(x) g(x)$, where $f(x) = 1/(e^x + 1)$ is the equilibrium distribution function and $g(x)$ is an arbitrary function of the dimensionless momentum $x = pc/k_B T$ (see [Ma & Bertschinger \(1995\)](#) for more details). One can generally approximate $I[g(x)] \approx \sum_i W_i g(x_i)$ with a weighted quadrature scheme with points x_i and weights W_i (more on this after the equations). In other words, the integral operator $\int_0^\infty dx x^2 f(x)$ is effectively replaced by the discrete summation operator $\sum_i W_i$ for some weights W_i .

On top of this, perturbations are also expanded in Legendre multipoles l up to a cutoff $l_{\max}$:

$$\begin{aligned} T &= \frac{T_0}{a}, & x &= \frac{pc}{k_B T}, & y &= \frac{mc^2}{k_B T}, & E_i &= \sqrt{x_i^2 + y^2}, & f &= \frac{1}{1 + e^x}, & \frac{d \ln f}{d \ln x} &= -\frac{x}{1 + e^{-x}}, \\ I_\rho &= \sum_i W_i E_i, & I_P &= \sum_i W_i \frac{x_i^2}{E_i}, & \rho &= \frac{N}{\pi^2} \frac{(k_B T)^4}{(\hbar c)^3} \frac{G}{(H_0 c)^2} I_\rho, & P &= \frac{N}{3\pi^2} \frac{(k_B T)^4}{(\hbar c)^3} \frac{G}{(H_0 c)^2} I_P, & w &= \frac{P}{\rho}, \\ \psi'_{i,0} &= -k \frac{x_i}{E_i} \psi_{i,1} - \Phi' \left(\frac{d \ln f}{d \ln x} \right)_i, & \psi'_{i,1} &= \frac{k}{3} \frac{x_i}{E_i} (\psi_{i,0} - 2\psi_{i,2}) - \frac{k}{3} \frac{E_i}{x_i} \Psi \left(\frac{d \ln f}{d \ln x} \right)_i, \\ \psi'_{i,l} &= \frac{k}{2l+1} \frac{x_i}{E_i} (l\psi_{i,l-1} - (l+1)\psi_{i,l+1}), & \psi_{i,l_{\max}+1} &= \frac{2l_{\max}+1}{k\tau} \frac{E_i}{x_i} \psi_{i,l_{\max}} - \psi_{i,l_{\max}-1}, \\ I_0 &= \sum_i W_i E_i \psi_{i,0}, & I_1 &= \sum_i W_i x_i \psi_{i,1}, & I_2 &= \sum_i W_i \frac{x_i^2}{E_i} \psi_{i,2}, & \delta &= \frac{I_0}{I_\rho}, & \sigma &= \frac{2I_2}{3I_\rho + I_P}, & u &= \frac{3I_1}{3I_\rho + I_P}, & \theta &= ku. \end{aligned}$$

Initial conditions are $\psi_{i,0} = -\frac{1}{4}(-2\Psi)\left(\frac{d \ln f}{d \ln x}\right)_i$, $\psi_{i,1} = -\frac{1}{3} \frac{E_i}{x_i} \frac{1}{2} k\tau \Psi\left(\frac{d \ln f}{d \ln x}\right)_i$, $\psi_{i,2} = -\frac{1}{2} \frac{1}{15} (k\tau)^2 \Psi\left(\frac{d \ln f}{d \ln x}\right)_i$ and $\psi_{i,l} = 0$. This integrates to adiabatic $\delta/(1+w)$, θ and σ similarly to *massless* neutrinos. Free parameters are the temperature today T_0 , mass m of a single neutrino and degeneracy factor $N = \sum_{i=1}^N m_i/m$ for describing multiple neutrinos with equal mass. The degeneracy factor defaults to $N = 3$, and the temperature to $T_{h0} = \left(\frac{4}{11}\right)^{1/3} T_{\gamma 0}$ if photons are present, as for massless neutrinos.

Here the equation for $\psi'_{i,l}$ applies for $2 \leq l \leq l_{\max}$, and expressions $g_i = g(x_i)$ indexed by i are evaluated with the momentum quadrature point $x = x_i$. The reduction to *dimensionless* momenta $x = pc/k_B T$ (the argument of $\exp$ in f) is deliberate because it makes numerics more well-defined and the quadrature scheme independent of m and all other cosmological parameters.

SymBoltz automatically computes momentum bins x_i and quadrature weights W_i with N -point Gaussian quadrature. First, by default, the following substitution is applied to the momentum integral:

$$\int_0^\infty dx x^2 f(x) g(x) = \int_{u(0)}^{u(\infty)} du x'(u) x(u)^2 f(x(u)) g(x(u)) \quad \text{with} \quad u(x) = \frac{1}{1 + \frac{x}{L}}.$$

This substitution achieves two things: the scaling x/L brings the dominant integral contributions well within $x/L \ll 1$ if L is chosen to be a characteristic decay “length” of the distribution function, and the rational part $1/(1+x/L)$ maps the infinite domain $x \in (0, \infty)$ to the finite domain $u \in (0, 1)$, which can be integrated numerically. The substituted integrand is then passed to an adaptive algorithm in QuadGK.jl¹³ that computes quadrature points u_i and weights W_i by performing weighted integrals against several test functions $g(x)$. Finally, the corresponding momenta $x_i = x(u_i)$ are returned along with the weights W_i , from which one can approximate the integral $I[g(x)] \approx \sum_i W_i g(x_i)$ against any $g(x)$.

SymBoltz tests this numerical quadrature scheme against the analytical result $I[x^{n-2}] = \int_0^\infty dx x^n/(e^x + 1) = (1 - 2^{-n})\zeta(n+1)\Gamma(n+1)$ for $2 \leq n \leq 8$. We assume this to be a reasonable test for the integrals encountered in the equations above. Agreement is excellent with $L = 100$, which yields relative errors below 10^{-6+n-N} for all $2 \leq n \leq 8$ and $1 \leq N \leq 5$. SymBoltz defaults to $N = 4$ momenta, for which this relative error is less than 10^{-6} for $n \leq 4$, for example. It also agrees well with CLASS using default settings.

Note that this momentum quadrature strategy is generic with respect to the distribution function $f(x)$ and substitution $u(x)$, so it can easily be modified for other particle species whose distribution function cannot be integrated out.

CAMB ([Lewis 2025](#)) and CLASS ([Lesgourgues & Tram 2011](#)) apply similar weighted quadrature strategies. They also get away with only a handful of sampled momenta, but the precise details of the computation differ slightly. For reference, here are points and weights computed by SymBoltz for $1 \leq N \leq 8$ momenta:

¹³ <https://github.com/JuliaMath/QuadGK.jl>

N	x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8
1	3.12273							
2	2.07807	5.94834						
3	1.56110	4.22902	8.86258					
4	1.24461	3.30909	6.57536	11.80351				
5	1.02955	2.71805	5.27853	9.03363	14.74043			
6	0.87373	2.30142	4.41479	7.39595	11.55088	17.65818		
7	0.75572	1.99028	3.79064	6.27323	9.60568	14.09716	20.54878	
8	0.66337	1.74848	3.31580	5.44442	8.24194	11.87257	16.65452	23.40806
N	W_1	W_2	W_3	W_4	W_5	W_6	W_7	W_8
1	1.80309							
2	1.30306	0.50002						
3	0.84813	0.88596	0.06899					
4	0.55272	0.99943	0.24384	0.00709				
5	0.36868	0.95311	0.43658	0.04409	0.00063			
6	0.25275	0.84165	0.58496	0.11736	0.00632	0.00005		
7	0.17792	0.71541	0.67227	0.21284	0.02386	0.00079	0.00000	
8	0.12832	0.59663	0.70569	0.31144	0.05685	0.00406	0.00009	0.00000

A.8. Cosmological constant (Λ)

The cosmological constant is equivalent to a very simple species without perturbations:

$$w = -1, \quad \rho = \rho_0, \quad P = -\rho, \quad \delta = 0, \quad \theta = 0, \quad \sigma = 0, \quad u = 0.$$

It is parametrized by the reduced density $\Omega_0 = \frac{8\pi}{3}\rho_0$ today. This is set to $\Omega_{\Lambda 0} = 1 - \sum_{s \neq \Lambda} \Omega_{s0}$ if all species s have a Ω_{s0} parameter and General Relativity is the theory of gravity. This constraint comes from the 1st Friedmann equation today.

A.9. Primordial power spectrum (I)

SymBoltz computes the inflationary primordial power spectrum parametrized by the amplitude A_s and tilt n_s :

$$P_0(k) = \frac{2\pi^2}{k^3} A_s \left(\frac{k}{k_p} \right)^{n_s-1}.$$

A.10. Matter power spectrum

SymBoltz computes the matter power spectrum for some desired set of species s , which are presumably matter-like at late times (e.g. $s \in \{c, b, h\}$):

$$P(k, \tau) = P_0(k) |\Delta(\tau, k)|^2 \quad \text{with} \quad \Delta = \delta + \frac{3\mathcal{H}}{k^2} \theta = \frac{\sum_s \delta \rho_s}{\sum_s \rho_s} + \frac{3\mathcal{H}}{k^2} \frac{\sum_s (\rho_s + P_s) \theta_s}{\sum_s (\rho_s + P_s)}.$$

Here Δ is the total gauge-independent overdensity with total δ and θ computed by summing the components of the energy-momentum tensor that are additive.

A.11. CMB power spectrum and line-of-sight integration

SymBoltz finds photon temperature and polarization multipoles today for any l by computing the line-of-sight integrals

$$\Theta_l^T(\tau_0, k) = \int_{\tau_i}^{\tau_0} S_T(\tau, k) j_l(k(\tau_0 - \tau)) d\tau \quad \text{with} \quad S_T = v \left(\frac{\delta_\gamma}{4} + \Psi + \frac{\Pi_\gamma}{16} \right) + e^{-\kappa} (\Psi + \Phi)' + \frac{(vu_b)'}{k} + \frac{3}{16k^2} (v\Pi_\gamma)'',$$

$$\Theta_l^E(\tau_0, k) = \sqrt{\frac{(l+2)!}{(l-2)!}} \int_{\tau_i}^{\tau_0} S_E(\tau, k) \frac{j_l(k(\tau_0 - \tau))}{(k(\tau_0 - \tau))^2} d\tau \quad \text{with} \quad S_E = \frac{3}{16} v\Pi_\gamma.$$

As first suggested by [Seljak & Zaldarriaga \(1996\)](#), this approach enables cheap computation for any l after integrating the perturbation ODEs with only a few $l \leq l_{\max}$. This drastically speeds up the computation over including all l in an enormous set of coupled perturbation ODEs. SymBoltz performs the integrals with the trapezoid method using the substitution $u(\tau) = \tanh(\tau)$, which adds more points in the early universe when sampled uniformly, using 768 points by default. Here j_l are the spherical Bessel functions

of the first kind. SymBoltz is not yet generalized to non-flat geometries, where they are replaced by hyperspherical functions. The cross-correlated angular spectrum between $A, B \in \{T, E\}$ is then computed from

$$C_l^{AB} = \frac{2\pi}{l(l+1)} D_l^{AB} = \frac{2}{\pi} \int_0^\infty dk k^2 P_0(k) \Theta_l^A(\tau_0, k) \Theta_l^B(\tau_0, k).$$

This integral is also performed with the trapezoid method. The point $(k, \Theta) = (0, 0)$ is included manually, for which the numerical solution to the perturbation ODEs is ill-defined. By default, the Θ_l are sampled on a fine grid of wavenumbers with spacing $\Delta k = 2\pi/2\tau_0$, which interpolates from solved perturbation modes on a coarse grid $\Delta k = 8/\tau_0$. Both grids range between $0.1l_{\min}/\tau_0 \leq k \leq 3l_{\max}/\tau_0$, where $l_{\min}$ and $l_{\max}$ are the angular spectrum's minimum and maximum requested multipoles.

Appendix B: Precision parameters for CLASS

When comparing results to CLASS in section 3, CLASS is configured with the following non-default precision parameters:

```
background_Nloga = 6000
tight_coupling_trigger_tau_c_over_tau_h = 1e-2
tight_coupling_trigger_tau_c_over_tau_k = 1e-3
radiation_streaming_approximation = 3
ur_fluid_approximation = 3
ncdm_fluid_approximation = 3
```

We also set $l_{\max_g}$, $l_{\max_pol_g}$, $l_{\max_ur}$, $l_{\max_ncdm}$ to the same $l_{\max}$ used by SymBoltz' model. These settings disable as many approximations as possible and reduces the impact of the tight-coupling approximation, which cannot be disabled. Oddly, we find that the parameter `background_Nloga` must be *decreased* from the default value 40000 to make the derivatives in fig. 4 stable. This parameter controls the number of points used for splining background functions in the perturbations. The default value of this parameter was changed from 3000 to 40000 in 2023, but we suspect that the increased density in points makes the splines susceptible to oscillations from numerical noise. These settings are important for good agreement between the Fisher forecasts in section 3.4. We used CLASS version 3.3.1.

Appendix C: Testing and comparison to CLASS

SymBoltz' code repository is set up with continuous integration that runs several tests and builds updated documentation pages every time changes to the code are committed. In particular, this compares the solution for Λ CDM with CLASS for many variables solved by the background, thermodynamics and perturbations (using the options `write_background`, `write_thermodynamics` and `k_output_values`). These are the basis for all derived quantities like luminosity distances, matter and CMB power spectra, which are also compared. The checks pass when the quantities agree within a small tolerance. We do not compare directly against more codes like CAMB, but CLASS has already been compared extensively with CAMB with excellent agreement (Lesgourgues 2011b). The comparison takes a lot of space and is not included here, but is found in the documentation linked from SymBoltz' repository.

Another test checks that integration of the background and perturbations equations are stable throughout parameter space. As the equations are very stiff and SymBoltz does not rely on approximations for relieving it, one could imagine that the integration would be stable for some parameter values and unstable for others. The test creates a box in parameter space $\pm 50\%$ around a fiducial set of realistic parameter values, draws several sets of parameter values from that space with Latin hypercube sampling (to efficiently cover parameter space) and integrates the background and perturbations for each such set. All parameter samples are found to integrate successfully without warnings and errors.