MiMeS: Misalignment Mechanism Solver

Karamitros Dimitrios

School of Physics and Astronomy, The University of Manchester, Manchester M13 9PL, United Kingdom
E-mail: dimitrios.karamitros@manchester.ac.uk

October 26, 2021

Abstract

We introduce a C++ header-only library that is used to solve the axion equation of motion, MiMeS. MiMeS makes no assumptions regarding the cosmology and the mass of the axion, which allows the user to consider various cosmological scenarios and axion-like models. MiMeS also includes a convenient python interface that allows the library to be called without writing any code in C++, with minimal overhead.

Contents

1 Introduction

The axion is a hypothetical particle that was originally introduced in order to solve the strong CP-problem of the standard model (SM) [1, 2, 3]. Furthermore, the axion is assumed to acquire a non-zero vacuum expectation value (VEV), which results in the spontaneous breaking of a global symmetry, called Peccei-Quinn (PQ). That is the axion is a pseudo-Nambu-Goldstone boson. Moreover, it also appears to be a valid dark matter (DM) candidate [4, 5, 6], as it is electrically neutral and long-lived. The axion starts, at very early time, with a VEV close to the PQ breaking scale (usually much higher than 100GeV), but due to the expansion of the Universe it eventually ends oscillating around zero. This oscillation results in an apparent constant number of axion particles (or constant energy of the axion field) today, which can account for the observed [7] DM relic abundance. Apart form the axion, there are other hypothetical particles (for early examples of such particles see [8, 9]; and [10] for a review), called axion-like particles (ALPs), which are not related to the strong CP-problem. That is, these ALPs, interact with the SM in a different way than the axion. However, they can still account for the DM of the Universe. From the point of view of their DM abundance, both axions and ALPs, follow the same dynamics (similar EOM). The only difference between the two is their mass. The mass of the axion is dictated by QCD, while it is model-dependent for other ALPs. Therefore, the cosmological evolution of both axions and ALP is often discussed together(see, for example ref. [11]).

In this article, introduce a header-only C++ library, MiMeS, [1] that solves the axion (or ALP) EOM, where both the mass and the underlying cosmological evolution are treated as user inputs. Therefore, MiMeS can be used to compute the relic abundance of axions or ALPs, in a wide variety of cases. Moreover, one can use MiMeS in python scripts, without having to write any C++ code.

[1] MiMeS is distributed under the MIT license. A copy of this license should be available in the MiMeS root directory. If you have not received one, you can find it at github.com/dkaramit/MiMeS/blob/master/LICENSE.

This paper is organized as follows: In section 2, we introduce the EOM and show how it can be solved approximately. Also, we introduce the notation that `MiMeS` follows, we derive the "adiabatic invariant" of the system, and discuss how it is used. In the next section, we introduce `MiMeS`, by showing how it can be downloaded, compiled, and run for the first time. Also, we explain in detail all the parameters that `MiMeS` as a user input at run-time. In section 4, we discuss the few assumption that `MiMeS` makes, its default compile-time options and user input, and how the user can change them. Moreover, we provide a complete example in both `C++` and `python`.

2 Physics background

Although there are several works in the literature (such as [12, 11]) that can provide a insight on the cosmological evolution of axions, in this section we define, derive, and discuss various quantities we need, in order to understand how `MiMeS` works in detail.

The EOM The axion field is usually expressed in terms of the so-called axion angle, θ, as $A = \theta\ f_a$, with f_a the scale at which the PQ symmetry breaks. [2] The axion angle follows the EOM

$$\left(\frac{d^2}{dt^2} + 3H(t)\ \frac{d}{dt}\right)\theta(t) + \tilde{m}_a{}^2(t)\ \sin\theta(t) = 0\ , \tag{2.1}$$

with $H(t)$ the Hubble parameter (determined by the cosmology), and $\tilde{m}_a(t)$ the time (temperature) dependent mass of the axion. `MiMeS` expects the mass to be of the form

$$\tilde{m}_a{}^2(T) = \frac{\chi(T)}{{f_a}^2}\ , \tag{2.2}$$

and χ a function of the temperature. For the QCD axion, this has been calculated using lattice simulations in [13]. [3]

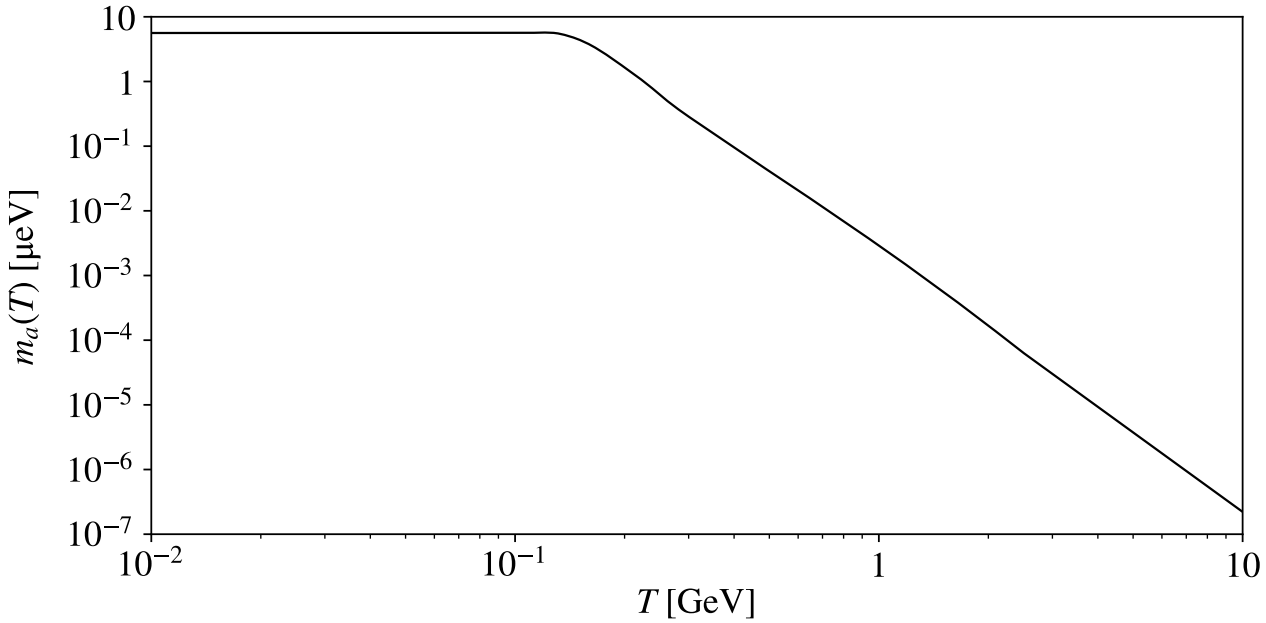


Figure 1: The mass of the axion as a function of the temperature for $f_a = 10^{12}$ GeV, using the data provided in ref. [13].

[2] If in a model under study, there is no f_a, this parameter is still expected by `MiMeS`, but the user can set $f_a = 1$.
[3] The data provided by ref. [13] are used by `MiMeS` out-of-the-box. However, the user is free to change them.

Initial conditions Assuming that the PQ symmetry breaks before inflation, the initial conditions (*i.e.* at some $t = 0$, after inflation) for the EOM is random. However, we note that $\tilde{m}_a \to 0$ (see Fig. (1)) – *i.e.* $\tilde{m}_a \ll H$ – at very early times. Therefore, after inflation, the EOM is simply

$$\left(\frac{d^2}{dt^2} + 3H(t)\ \frac{d}{dt}\right)\theta(t) = 0\ , \tag{2.3}$$

which is solved by $\theta = \theta_{\rm ini} + C\int_0^t dt'\ \left(\frac{a(t'=0)}{a(t')}\right)^3$, with $\theta_{\rm ini}$ a constant and a the scale factor of the Universe. That is, as the Universe expands, and as long as $\tilde{m}_a \ll H$, $\theta \to \theta_{\rm ini}$. Since we would like to calculate the relic abundance of axions, we can integrate eq. (2.1) from a time after inflation (call it $t = t_{\rm ini}$) such that $\dot{\theta}|_{t=t_{\rm ini}} = 0$ and $\theta|_{t=t_{\rm ini}} = \theta_{\rm ini}$. This is the most common case (there are exceptions to this; *e.g.* [14]), and it is what `MiMeS` uses.

2.1 The WKB approximation

In order to solve analytically eq. (2.1), we assume $\theta \ll 1$, which results in the linearised EOM

$$\left(\frac{d^2}{dt^2} + 3H(t)\ \frac{d}{dt} + {\tilde{m}_a}^2(t)\right)\theta(t) = 0\ . \tag{2.4}$$

Using a trial solution $\theta_{\rm trial} = \exp\left[i\int dt\ \Big(\psi(t) + 3/2\ i\ H(t)\Big)\right]$, and defining $\Omega^2 = {\tilde{m}_a}^2 - \frac{9}{4}H^2 - \frac{3}{2}\dot{H}$ we can transform the eq. (2.4) to

$$\psi^2 = \Omega^2 + i\ \dot{\psi}\ , \tag{2.5}$$

which has a formal solution $\psi = \pm\sqrt{\Omega^2 + i\dot{\psi}}$. In the WKB approximation, we assume a slow time-dependence; that is $\dot{\psi} \ll \Omega^2$ and $\dot{\Omega} \ll \Omega^2$. Then we can approximate ψ as

$$\psi \approx \pm\Omega + \frac{i}{2}\frac{d\log\Omega}{dt}\ , \tag{2.6}$$

which results in the general solution of eq. (2.4)

$$\theta \approx \frac{1}{\sqrt{\Omega}}\exp\left(-\frac{3}{2}\int dt\ H\right)\left[A\cos\left(\int dt\ \Omega\right) + B\sin\left(\int dt\ \Omega\right)\right]\ . \tag{2.7}$$

Applying, then, the initial conditions $\dot{\theta}|_{t=t_{\rm ini}} = 0$ and $\theta|_{t=t_{\rm ini}} \approx \theta_{\rm ini}$, we arrive at

$$\theta(t) \approx \theta_{\rm ini}\sqrt{\frac{\Omega_{\rm ini}}{\Omega(t)}}\left(\frac{a}{a_{\rm ini}}\right)^{-3/2}\ \cos\left(\int_{t_{\rm ini}}^t dt'\ \Omega(t')\right)\ . \tag{2.8}$$

In order to further simplify this approximate result, we note that θ deviates from $\theta_{\rm ini}$ close to $t = t_{\rm osc}$ – corresponding to $T = T_{\rm osc}$, the so-called "oscillation temperature" – $\tilde{m}_a|_{t=t_{\rm osc}} = 3H|_{t=t_{\rm osc}}$, which is defined as the point at which the axion begins to oscillate. This observation allows us to set $t_{\rm ini} = t_{\rm osc}$. Moreover, at $t > t_{\rm osc}$, we approximate $\Omega \approx \tilde{m}_a$, as H^2 and $\dot{H}$ become much smaller than ${\tilde{m}_a}^2$ quickly after $t = t_{\rm osc}$. Finally, the axion angle takes the form

$$\theta(t) \approx \theta_{\rm osc}\left(\frac{3}{4}\right)^{1/4}\sqrt{\frac{\tilde{m}_a|_{t=t_{\rm osc}}}{\tilde{m}_a(t)}}\left(\frac{a}{a_{\rm osc}}\right)^{-3/2}\ \cos\left(\int_{t_{\rm osc}}^t dt'\ \tilde{m}_a(t')\right)\ , \tag{2.9}$$

where $\theta_{\rm osc} = \theta|_{t=t_{\rm osc}}$. This equation is further simplified if we assume that $\theta_{\rm osc} \approx \theta_{\rm ini}$, *i.e.*

$$\theta(t) \approx \theta_{\rm ini}\left(\frac{3}{4}\right)^{1/4}\sqrt{\frac{\tilde{m}_a|_{t=t_{\rm osc}}}{\tilde{m}_a(t)}}\left(\frac{a}{a_{\rm osc}}\right)^{-3/2}\ \cos\left(\int_{t_{\rm osc}}^t dt'\ \tilde{m}_a(t')\right)\ . \tag{2.10}$$

It is worth mentioning that the accuracy of this approximation depends, in general, on $T_{\rm osc}$; it determines the difference between $\theta_{\rm ini}$ and $\theta_{\rm osc}$, the deviation of $\dot{\theta}|_{t=t_{\rm osc}}$ from 0, and whether $\dot{\Omega} \ll \Omega^2$.

Axion energy density In the small angle approximation, the energy density of the axion is

$$\rho_a = \frac{1}{2} {f_a}^2 \left[\dot{\theta}^2 + {\tilde{m}_a}^2 \theta^2\right] \ . \tag{2.11}$$

For the relic abundance of axions, we need to calculate their energy density at very late times. That is, $\dot{\tilde{m}}_a = 0$, $\tilde{m}_a \gg H$ and $\dot{H} \ll H^2$. After some algebra, we obtain the approximate form of the energy density as a function of the scale factor (a)

$$\rho_a \approx \frac{m_a}{2} \ {f_a}^2 \ {\theta_{\rm ini}}^2 \ \tilde{m}_a(a_{\rm osc}) \ \left(\frac{a_{\rm osc}}{a}\right)^3 \ , \tag{2.12}$$

which shows that the energy density of axions at late times scales as the energy density of matter; *i.e.* the number of axion particles is conserved. If there is a period of entropy injection to the plasma for $T < T_{\rm osc}$, the axion energy density gets diluted, since

$$a^3 \ s = \gamma \ {a_{\rm osc}}^3 \ s_{\rm osc} \Rightarrow \left(\frac{a_{\rm osc}}{a}\right)^3 = \gamma^{-1} \frac{s}{s_{\rm osc}} \ , \tag{2.13}$$

with γ the amount of entropy injection to the plasma between $t_{\rm osc}$ and t. Therefore, the present (at $T = T_0$) energy density of the axion, becomes

$$\rho_{a,0} = \gamma^{-1} \frac{s_0}{s_{\rm osc}} \ \frac{1}{2} \ {f_a}^2 \ m_a \ \tilde{m}_{a,{\rm osc}} \ {\theta_{\rm ini}}^2 \ , \tag{2.14}$$

with m_a the mass of the axion at $T = T_0$. Notice that the explicit dependence on f_a cancels, since $\tilde{m}_a \sim 1/f_a$. That is, f_a only affects the energy density of the axions through its impact on $T_{\rm osc}$.

2.2 Notation

The EOM (2.1) depends on time, which is not very useful variable in cosmology. Therefore, we introduce

$$u = \log \frac{a}{a_{\rm ini}} \ , \tag{2.15}$$

which results in

$$\begin{aligned} \frac{dF}{dt} &= H \frac{dF}{du} \\ \frac{d^2F}{dt^2} &= H^2 \ \left(\frac{d^2F}{du^2} + \frac{1}{2} \frac{d\log H^2}{du} \frac{dF}{du}\right) \ . \end{aligned} \tag{2.16}$$

The EOM in terms of u, then, becomes

$$\frac{d^2\theta}{du^2} + \left[\frac{1}{2} \frac{d\log H^2}{du} + 3\right] \frac{d\theta}{du} + \ \left(\frac{\tilde{m}_a}{H}\right)^2 \ \sin\theta = 0 \ . \tag{2.17}$$

Notice that in a radiation dominated Universe

$$\frac{d\log H^2}{du} = -\left(\frac{d\log g_{\rm eff}}{d\log T} + 4\right) \delta_h^{-1} \ ,$$

with $\delta_h = 1 + \frac{1}{3} \frac{d\log h_{\rm eff}}{d\log T}$. In a general cosmological setting, if the expansion rate is dominated by an energy density that scales as $\rho \sim a^{-c}$, $\frac{d\log H^2}{du} = -c$. We notice that close to rapid particle annihilations and decays, the evolution of the energy densities change, and $\frac{d\log H^2}{du}$ can only be computed numerically.

Moreover, it is worth mentioning that the EOM 2.17 with the initial condition $\theta(u=0)=\theta_{\rm ini}$ and $d\theta/du(u=0)=0$ can be written as a system of first order ordinary differential equations

$$\begin{aligned} \frac{d\zeta}{du} + \left[\frac{1}{2}\frac{d\log H^2}{du} + 3\right]\zeta + \left(\frac{\tilde{m}_a}{H}\right)^2 \sin\theta = 0 \,. \\ \frac{d\theta}{du} - \zeta = 0 \,. \end{aligned} \tag{2.18}$$

This form of the EOM is what `MiMeS` uses, as it is suitable for integration via Runge-Kutta methods – which are briefly discussed in Appendix A.

2.3 Adiabatic invariant and the anharmonic factor

The small angle approximation does not allow us to solve the EOM 2.1 analytically. Moreover, even for $\theta \ll 1$, the WKB approximation fails to capture the dynamics before the adiabatic conditions are met, and result in an inaccurate axion relic abundance. Therefore, a numerical integration should be preferred. Furthermore, in order to reduce the computation time, the numerical integration needs to stop as soon as the axion begins to evolve adiabatically. After this point, we can correlate its energy density at later times, using an "adiabatic invariant", which can be defined for oscillatory systems with varying period.

Definition of the adiabatic invariant Given a system with Hamiltonian $\mathcal{H}(\theta, p; t)$, the equations of motion are

$$\dot{p} = -\frac{\partial \mathcal{H}}{\partial \theta} \,, \quad \dot{\theta} = \frac{\partial \mathcal{H}}{\partial p} \,. \tag{2.19}$$

Moreover, we note that

$$d\mathcal{H} = \dot{\theta}\; dp - \dot{p}\; d\theta + \frac{\partial \mathcal{H}}{\partial t}\; dt \,. \tag{2.20}$$

If this system exhibits closed orbits (*e.g.* if it oscillates), we define

$$J \equiv C \oint p\; d\theta \,, \tag{2.21}$$

where the integral is over a closed path (*e.g.* a period, T), and C indicates that J can always be rescaled with a constant. This quantity is the adiabatic invariant of the system, if the Hamiltonian varies slowly during a cycle. That is,

$$\frac{dJ}{dt} = C \oint \left(\dot{p}\; d\theta + p\; d\dot{\theta}\right) = C \int_t^{t+T} \frac{\partial \mathcal{H}}{\partial t'}\; dt' \approx T\; \frac{\partial \mathcal{H}(t')}{\partial t'}\Big|_{t'=t} \approx 0 \,.$$

Application to the axion The Hamiltonian that results in the EOM of eq. (2.1) is

$$\mathcal{H} = \frac{1}{2}\frac{p^2}{{f_a}^2\; a^3} + V(\theta)\; a^3 \,, \tag{2.22}$$

with

$$p = {f_a}^2\; a^3\; \dot{\theta} \tag{2.23}$$

$$V(\theta) = {\tilde{m}_a}^2 {f_a}^2 (1 - \cos\theta) \,. \tag{2.24}$$

Notice that the Hamiltonian varies slowly if $\dot{\tilde{m}}_a(T)/\tilde{m}_a \ll \tilde{m}_a$ and $H \ll \tilde{m}_a$, which are the adiabatic conditions. When these conditions are met, the adiabatic invariant for this system becomes

$$J = \frac{\oint p\; d\theta}{\pi {f_a}^2} = \frac{1}{\pi {f_a}^2} \oint \sqrt{2\left(\mathcal{H}(\theta) - V(\theta)\; a^3\right)\; {f_a}^2 a^3}\; d\theta = \frac{2}{\pi {f_a}^2} \int_{-\theta_{\rm peak}}^{\theta_{\rm peak}} \sqrt{2\left(\mathcal{H}(\theta_{\rm peak}) - V(\theta)\; a^3\right)\; {f_a}^2 a^3}\; d\theta \,, \tag{2.25}$$

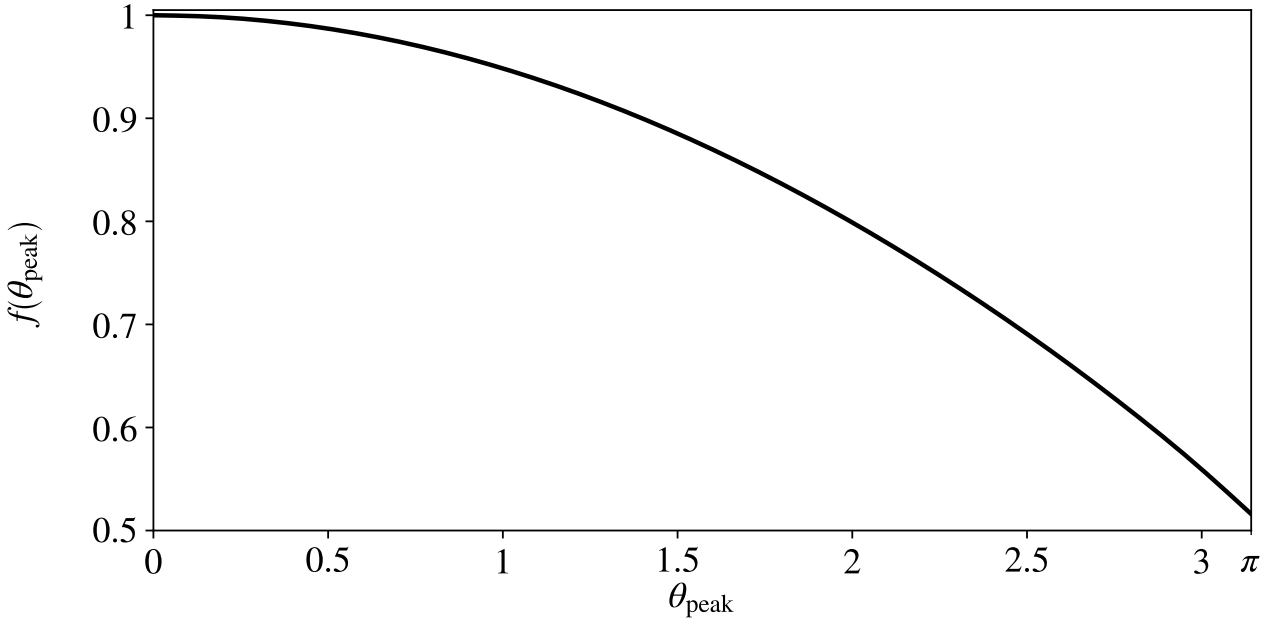


Figure 2: The anharmonic factor for $0 \leq \theta_{\rm peak} < \pi$.

where we note that $\theta_{\rm peak}$ denotes the maximum of θ – the peak of the oscillation, which corresponds to $p = 0$. That is, $\mathcal{H}(\theta_{\rm peak}) = V(\theta_{\rm peak})\ a^3$. Therefore, the adiabatic invariant, takes the form

$$J = \frac{2\sqrt{2}}{\pi f_a} \int_{-\theta_{\rm peak}}^{\theta_{\rm peak}} \sqrt{V(\theta_{\rm peak}) - V(\theta)} a^3 d\theta = \frac{2\sqrt{2}}{\pi}\ \tilde{m}_a\, a^3 \int_{-\theta_{\rm peak}}^{\theta_{\rm peak}} \sqrt{\cos\theta - \cos\theta_{\rm peak}}\ d\theta\ , \quad (2.26)$$

where, for the last equality. we have used the adiabatic conditions, *i.e.* negligible change of $\tilde{m}_a$ and a during one period. Usually, the adiabatic invariant is written as [15, 16]

$$J = a^3\ \tilde{m}_a\ {\theta_{\rm peak}}^2\, f(\theta_{\rm peak})\ , \quad (2.27)$$

where

$$f(\theta_{\rm peak}) = \frac{2\sqrt{2}}{\pi {\theta_{\rm peak}}^2} \int_{-\theta_{\rm peak}}^{\theta_{\rm peak}} d\theta \sqrt{\cos\theta - \cos\theta_{\rm peak}}\ , \quad (2.28)$$

is called the anharmonic factor, with $0.5 \lesssim f(\theta_{\rm peak}) \leq 1$ (see Fig. (2)).

The role of the adiabatic invariant in the axion relic energy density The adiabatic invariant allows us to calculate the maximum value of the angle θ at late times from its corresponding value at some point just after the adiabatic conditions where fulfilled.

In order to do this, we can numerically integrate eq. (2.1), and identify the maxima of θ. [4] Once the adiabatic conditions are fulfilled, we can stop the integration at a peak, $\theta_{{\rm peak},*}$ – which corresponds to $T = T_*$ and $a = a_*$. Then, the value of the maximum angle today ($\theta_{{\rm peak},0} \ll 1$) is related to $\theta_{{\rm peak},*}$ via

$$\theta_{{\rm peak},0}^2 = \left(\frac{a_*}{a_0}\right)^3 \frac{\tilde{m}_{a,*}}{m_a}\ f(\theta_{{\rm peak},*})\ \theta_{{\rm peak},*}^2 = \gamma^{-1}\ \frac{s_0}{s_*}\ \frac{\tilde{m}_{a,*}}{m_a}\ f(\theta_{{\rm peak},*})\ \theta_{{\rm peak},*}^2\ . \quad (2.29)$$

Plugging this into eq. (2.11) (with $\dot{\theta} = 0$, *i.e.* at today's peak), we arrive at the energy density today

$$\rho_{a,0} = \gamma^{-1}\ \frac{s_0}{s_*}\ m_a\ \tilde{m}_{a,*}\ \frac{1}{2}\ {f_a}^2\ \theta_{{\rm peak},*}^2\ \ f(\theta_{{\rm peak},*})\ , \quad (2.30)$$

[4] `MiMeS` identifies the maxima in real time, so it stops as soon as the system becomes adiabatic.

where γ is the entropy injection coefficient between T_* and T_0, defined from

$$a^3(T_0)\ s(T_0) = \gamma\ a^3(T_*)\ s(T_*) \tag{2.31}$$

Notice that eq. (2.30) is similar to the corresponding WKB result (2.14) at $t_{\rm osc} \to t_*$, multiplied by the anharmonic factor $f(\theta_{\rm peak,*})$. So, the numerical integration is needed in order to correctly identify T_* and $\theta_{\rm peak,*}$, which are greatly affected by the underlying cosmology (especially in cases where entropy injection is active close to $T_{\rm osc}$) and whether $\theta_{\rm ini} \gtrsim 1$ [16, 17].

3 MiMeS usage

The latest stable version of MiMeS is available in https://github.com/dkaramit/MiMeS/tree/stable, which can also be obtained by running: [5]

```
git clone -b stable https://github.com/dkaramit/MiMeS.git
```

It is important to note that MiMeS relies on NaBBODES [18] and SimpleSplines [19]. These are two libraries developed independently, and in order to get MiMeS with the latest version of these libraries, one needs to run following commands

```
git clone https://github.com/dkaramit/MiMeS.git
cd MiMeS
git submodule init
git submodule update --remote
```

This downloads the master branch of MiMeS; and NaBBODES [18] and SimpleSplines [19] as submodules. This guaranties that MiMeS uses the most updated version of these libraries, although it may not be stable.

Once everything is downloaded successfully, we can go inside the MiMeS directory, and run "bash configure.sh" and "make". The bash script configure.sh, just writes some paths to some files, formats the data files provided in an acceptable format (in section 4.2 the format is explained), and makes some directories. The makefile is responsible for compiling some examples and checks, as well as the shared libraries that needed for the python interface. If everything runs successfully, there should be two new directories exec and lib. Inside exec, there are several executables that ready to run, in order to ensure that the code runs (*e.g.* no segmentation fault occurs). For example, exec/AxionSolve_check.run, should print the values of the parameters $\theta_{\rm ini}$ and f_a, the oscillation temperature and the corresponding value of θ, the evolution of the axion (*e.g.* temperature, θ, ρ_a, etc.), and the values of various quantities on the peaks of the oscillation. In the directory lib, there are several shared libraries for the python interface.

Although there are various options available at compile-time, we first discuss how MiMeS can be used, in order for the role of these options to be clear.

3.1 First steps

There are several examples in C++ (MiMeS/UserSpace/Cpp) and python (MiMeS/UserSpace/Python), as well as jupyter notebooks (MiMeS/UserSpace/JupyterNotebooks), that show in detail how MiMeS can be used. Here, we discuss the various functions one may use as it will provide some insight for the following discussions.

3.1.1 Using MiMeS in C++

The class that is responsible for the solution of the EOM is mimes::Axion<LD,Solver,Method>, located in MiMeS/src/Axion. However, in order to use it, we have to define the mass of the axion as a function of the temperature and f_a. The axion mass is defined as an instance of the mimes::AxionMass<LD>, which is defined in the header file MiMeS/src/AxionMass/AxionMass.hpp. We should note that LD

[5] Instructions on how git can be installed can be found in https://github.com/git-guides/install-git.

is the numerical type we use (*e.g.* `long double`). The other template arguments are related to the differential equation solver, and their role will be explained in later sections.

Therefore, at the top of the main `.cpp` file, we need to include

```
#include "src/AxionMass/AxionMass.hpp"
#include "src/Axion/AxionSolve.hpp"
```

Notice that if the this `.cpp` file is not in the root directory of `MiMeS`, we need to compile it using the flag `-Ipath-to-root`, "path-to-root" the relative path to the root directory of `MiMeS`; *e.g.* if the `.cpp` is in the `MiMeS/UserSpace/Cpp/Axion` directory, this flag should be `-I../../../`.

The mass of the axion can be defined in two ways; either via a data file or as a user defined function.

Axion mass form data file In many cases, axion mass cannot be written in a closed form. In these cases, the user may provide a data file. Then, the axion mass can be defined via this file as

```
mimes::AxionMass<LD> axionMass(chi_PATH, minT, maxT);
```

The template parameter `LD` can be a numeric type (*e.g.* `double` or `long double`). The argument `chi_PATH` is a (relative or absolute) path to a file with two columns; T (in GeV) and χ (in GeV^4), with increasing T. In this case, the axion mass is interpolated between the temperatures `minT` and `maxT`. These two parameters are just suggestions, and the actual interpolation limits, `TMin` and `TMax`, are chosen as the closest temperatures in the data file. That is, in general, `TMin` $\geq$ `minT` and `TMax` $\leq$ `maxT`. Beyond these limits, the mass is assumed to constant by default. However, one can change this by using

```
axionMass.set_ma2_MAX(ma2_MAX);
axionMass.set_ma2_MIN(ma2_MIN);
```

with `ma2_MAX` and `ma2_MIN` the axion mass squared as functions (or any other callable objects) with signatures `LD ma2_MAX(LD T, LD fa)` and `LD ma2_MIN(LD T, LD fa)`. These functions are called for $T \geq$`TMax` and $T \leq$`TMin`, respectively. Usually, in order to ensure continuity of the axion mass, one needs to know `TMin`, `TMax`, χ(`TMin`), and χ(`TMax`); which can be found by calling `axionMass.getTMin()`, `axionMass.getTMax()`, `axionMass.getChiMin()`, and `axionMass.getChiMax()`, respectively.

Axion mass function In some cases, the dependence of the axion mass on the temperature is known. If this is the case, the user can define the axion mass via

```
mimes::AxionMass<LD> axionMass(ma2);
```

with `ma2` the axion mass squared as a callable object with signature `LD ma2(LD T, LD fa)`.

The EOM solver Once the axion mass is defined, we can declare a variable that will be used to solve the EOM, as

```
mimes::Axion<LD,Solver,Method> ax(theta_i, fa, umax, TSTOP, ratio_ini,
   N_convergence_max, convergence_lim, inputFile, &axionMass, initial_step_size,
   minimum_step_size, maximum_step_size, absolute_tolerance, relative_tolerance,
   beta, fac_max, fac_min, maximum_No_steps);
```

Here, `LD` should be the numeric type to be used; it is recommended to use `long double`, but other choices are also available as we discuss later. Moreover `Solver` and `Method` depend on the type of Runge-Kutta (RK) the user chooses. The available choices are shown in table4. A discussion on RK methods is given in Appendix A.

The various parameters are as follows:

1. `theta_i`: initial angle.
2. `fa`: the PQ scale.

3. `umax` : if $u >$`umax` the integration stops (remember that $u = \log(a/a_i)$). Typically, this should be a large number (~ 1000), in order to avoid stopping the integration before the axion begins to evolve adiabatically.

4. `TSTOP`: if the temperature drops below this, integration stops. In most cases this should be around 10^{-4} GeV, in order to be sure that any entropy injection has stopped before integration stops (since BBN bounds [20, 21] should not be violated).

5. `ratio_ini`: integration starts when $3H/\tilde{m}_a \approx$`ratio_ini` (the exact point depends on the file "`inputFile`", which we will see later).

6. `N_convergence_max` and `convergence_lim`: integration stops when the relative difference between two consecutive peaks is less than `convergence_lim` for `N_convergence_max` consecutive peaks.

7. `inputFile`: relative (or absolute) path to a file that describes the cosmology. the columns should be: u T [GeV] $\log H$, sorted so that u increases. [6] It is important to remember that `MiMeS` assumes that the entropy injection has stopped before the lowest temperature of given in `inputFile`. Since `MiMeS` is unable to guess the cosmology beyond what is given in this file, the user has to make sure that there are data between the initial temperature (which corresponds to `ratio_ini`, and `TSTOP`).

8. `axionMass`: an instance of the `mimes::AxionMass<LD>` class, passed by pointer.

9. `initial_stepsize` (optional): initial step the solver takes.

10. `maximum_stepsize` (optional): This limits the step-size to an upper limit.

11. `minimum_stepsize` (optional): This limits the step-size to a lower limit.

12. `absolute_tolerance` (optional): absolute tolerance of the RK solver

13. `relative_tolerance` (optional): relative tolerance of the RK solver. Generally, both absolute and relative tolerances should be 10^{-8}. In some cases, however, one may need more accurate result (*e.g.* if `f_a` is extremely high, the oscillations happen violently, and the system destabilizes). Whatever the case, if the tolerances are below 10^{-8}, long doubles have be used. `MiMeS` by default uses long double variables, in order to change it see the options available in section 4.2.

14. `beta` (optional): controls how agreesive the adaptation is. Generally, it should be around but less than 1.

15. `fac_max`, `fac_min` (optional): the stepsize does not increase more than fac_max, and less than fac_min. This ensures a better stability. Ideally, `fac_max`$= \infty$ and `fac_min`$= 0$, but in reality one must tweak them in order to avoid instabilities.

16. `maximum_No_steps` (optional): maximum steps the solver can take Quits if this number is reached even if integration is not finished.

In order to understand the role of the optional parameters, some basic techniques of Runge-Kutta methods are discussed in Appendix A.

The EOM (2.17), then can be solved using

```
1  ax.solveAxion();
```

Once the EOM is solved, we can access $T_{\rm osc}$, $\theta_{\rm osc}$, and Ωh^2 via `ax.T_osc`, `ax.theta_osc`, and `ax.relic`. The entire evolution (the points the integrator took) of the axion angle is stored in `ax.points`, which is a two-dimensional `std::vector<LD>`, with the columns corresponding to a, T [GeV], θ, $d\theta/du$, ρ_a.

[6] One can run "`bash MiMeS/src/FormatFile.sh inputFile`" in order to sort it and remove any unwanted duplicates. See Appendix F for details of `MiMeS/src/FormatFile.sh`.

Moreover, the peaks of he oscillation are stored in another two-dimensional `std::vector<LD>`, with the columns corresponding to a, T [GeV], $\theta_{\rm peak}$, $d\theta/du = 0$, ρ_a, J. We should note that the peaks are identified using linear interpolation between integration points, in order to ensure that $d\theta/du = 0$. That is, the values stored in `ax.peaks` do not exist in `ax.points`. Finally, the local errors (see Appendix A) of the integration points of θ and ζ are stored in `ax.dtheta` and `ax.dzeta`.

Changing axion mass definition The axion mass definiton can change at any time by changing the data file (or `ma2_MIN` and `ma2_MAX`) or using using

```
ax.set_ma2(ma2);
```

with `ma2` a callable object with signature `LD ma2(LD T, LD fa)`.

However, since integration starts at a temperature determined by `ratio_ini`, if the mass changes (including the definitions beyond the interpolation limits), we have to remake the interpolation of the underlying cosmology described by the parameter `inputFile`. Thus, if we change the definition of the mass, we need to call

```
ax.restart();
```

This function remakes the interpolations, clears all vector, and sets all variables to 0.

Changing initial condition The final member function is `mimes::Axion::setTheta_i`, which allows the user to set a different $\theta_{\rm ini}$ without generating another instance. [7] This function is used as

```
ax.setTeta_i(new_theta_ini);
```

where `new_theta_ini` is the new value of $\theta_{\rm ini}$. Running this function resets all variables to 0 (except `T_osc` and `a_osc`, since they should not change), and clears all `std::vector<LD>` variables, which allows the user to simply run "`ax.solveAxion();`" as if `ax` was a freshly defined instance.

3.1.2 Using `MiMeS` in `python`

The modules for the `python` interface are located in `MiMeS/src/interfacePy`. Although the usage of the `AxionMass` and `Axion` classes is similar to the `C++` case, it is worth showing explicitly how the `python` interface works. One should keep in mind that the various template arguments discussed in the `C++` case have to be chosen at compile-time. That is, for the `python` interface, one needs to choose the numeric type, and RK method to be used when the shared libraries are compiled. This is done by assigning the relevant variable in `MiMeS`/Definitions.mk before running "`make`". The various options are discussed in section 4.1, and outlined in table 5.

The two relevant classes are defined in the modules `interfacePy.AxionMass` and `interfacePy.Axion`, and can be loaded in a `python` script as

```
from sys import path as sysPath
sysPath.append('path_to_src')
from interfacePy.AxionMass import AxionMass
from interfacePy.Axion import Axion
```

It is important that `'path_to_src'` provides the relative path to the `MiMeS/src` directory. For example, if the script is located in `MiMeS/UserSpace/Python`, `'path_to_src'` should be `'../../src'`.

Axion mass definition via a data file As before, we first need to define the axion mass. Similarly to the previous case, in order to define the axion mass via a file, we use

```
AxionMass axionMass(chi_PATH, minT, maxT)
```

[7] Since the interpolations of the data of `inputFile` are made inside the constructor of the `mimes::Axion<LD,Solver,Method>` class, `mimes::Axion<LD,Solver,Method>::setTheta_i` is a faster choice if ones needs to solve the EOM for a different initial condition.

Here, the constructor requires the same parameters as in C++. Moreover, the axion mass beyond the interpolation limits can be changed via

```
axionMass.set_ma2_MAX(ma2_MAX)
axionMass.set_ma2_MIN(ma2_MIN)
```

Although the naming is the same as in the C++ case, there is an important difference. Namely, ma2_MAX and ma2_MIN have to be functions (that take T and f_a as arguments and return $\tilde{m}_a{}^2$), and cannot be any other callable object. The reason is that MiMeS uses the ctypes module, which only works with objects compatible with C. Moreover, the values of TMin, TMax, χ(TMin), and χ(TMax) can be obtained by axionMass.getTMin(), axionMass.getTMax(), axionMass.getChiMin(), and axionMass.getChiMax(), respectively.

Axion mass definition via a function Again this can be done as

```
AxionMass axionMass(ma2)
```

with ma2 the axion mass squared, which should be a function (not any callable object) of T and f_a.

Importand note Once an AxionMass is no longer required, to call the destructor. In this case, we can run

```
del axionMass
```

The reason is that MiMeS constructs a pointer for every instance of the class, which needs to be deleted manually.

The EOM solver We can define an Axion instance as follows

```
ax=Axion(theta_i, fa, umax, TSTOP, ratio_ini, N_convergence_max, convergence_lim,
   inputFile, axionMass, initial_step_size, minimum_step_size, maximum_step_size,
   absolute_tolerance, relative_tolerance, beta, fac_max, fac_min, maximum_No_steps)
```

Here the input parameters are the same as in the C++ case (the usage of the class can be found by running ?Axion after loading the module). The only slight difference compared to the C++ case is that the axionMass instance is not passed as a pointer; it is done internally using ctypes.

Using the defined variable (ax in this example), we can simply run

```
ax.solveAxion()
```

in order to solve the EOM of the axion. In contrast to the C++ implementation, this only gives us access to $T_{\rm osc}$, $\theta_{\rm osc}$, and Ωh^2; the corresponding variables are ax.T_osc, ax.theta_osc, and ax.relic. In order to get the evolution of the axion field, we need to run

```
ax.getPoints()
```

This will make numpy arrays that contain the scale factor (ax.a), temperature (ax.T), θ (ax.theta), its derivative with respect to u (ax.zeta), and the energy density of the axion (ax.rho_axion).

Moreover, in order to get the various quantities on the peaks of the oscillation, we can run

```
ax.getPeaks()
```

This makes numpy arrays that contain the scale factor (ax.a_peak), temperature (ax.T_peak), θ (ax.theta_peak), its derivative with respect to u (ax.zeta_peak, which should be equal to 0), the energy density of the axion (ax.rho_axion_peak), and the values of the adiabatic invariant on the peaks (ax.adiabatic_invariant).

Moreover, we can run the following

```
ax.getErrors()
```

in order to store the local errors (see Appendix A) of θ and ζ in ax.dtheta and ax.dzeta, respectively.

Changing axion mass definition We can change the axion mass by changing the file, `ma2_MIN` and `ma2_MAX`, or using

```
ax.set_ma2(ma2)
```

with `ma2` a function that takes T and f_a, and returns the $\tilde{m}_a{}^2$. As in the `C++` case, if the definition of the mass of the axion changes (including the definitions beyond the interpolation limits), we have to call

```
ax.restart()
```

in order to remake the interpolation of cosmological quantities and reset the various variables.

Changing the initial condition The initial condition $\theta_{\rm ini}$ can be changed using

```
ax.setTeta_i(new_theta_ini)
```

which is faster than running the constructor again, since all the interpolations are reused. However, running this function, erases all the arrays, and resets all variables to 0 (except `T_osc` and `a_osc`, as they should not change).

Importand note The `Axion` class constructs a pointer to an instance of the underlying `mimes::Axion` class, which has to be manually deleted. Therefore, once `ax` is used it must be deleted, *i.e.* we need to run

```
del ax
```

We should note that this must run even if we assign another instance to the same variable `ax`, otherwise we risk running out of memory.

4 Assumptions and user input

`MiMeS` only makes a few, fairly general, assumptions. First of all, it is assumed that the axion energy density is always subdominant compared to radiation or any other component of the Universe, and that decays and annihilations of particles have a negligible effect on the axion energy density. Moreover, the initial condition is always assumed to be $\theta_{t=t_{\rm ini}} = \theta_{\rm ini}$ and $\dot{\theta}|_{t=t_{\rm ini}} = 0$. Furthermore, it is also assumed that $3H/\tilde{m}_a$ increases monotonically at high temperatures. Also, it is assumed that the entropy of the plasma is resumes its conserved status at a temperature higher than the minimum temperature of `inputFile` (which is required by the constructor of the `mimes::Axion<LD,Solver,Method>` class). Finally, `MiMeS` does not try to predict anything regarding the cosmology. Therefore, the temperatures in `inputFile` *must* cover the entire region of integration; *i.e.* the maximum temperature has to be larger than the one required to reach `ratio_ini`, while the minimum one should be lower than `TSTOP`.

4.1 Options at Compile-time

The user has a number of options regarding different aspects for the code. If `MiMeS` is used without using the available makefiles, then they must use the correct values for the various template arguments, explained in Appendix B. The various choices we for the shared libraries used by the `python` interface are given in `MiMeS/Definitions.mk` while the corresponding options for the `C++` examples are in the `Definitions.mk` files in the directories inside the subdirectories of `MiMeS/UserSpace/Cpp`. The options correspond to different variables, which are

1. `rootDir`: the relative path of root directory of `MiMeS`.
2. `LONG`: this should be either `long` or omitted. If omitted, the type of the numeric values is `double` (double precision). On the other hand, if `LONG=long`, the type is `long double`. Generally, using `double` should be enough. For the sake of numerical stability, however, it is advised to always use `LONG=long`, as it a safer option. The reason is that the axion angle redshifts, and can become very

small, which introduces "rounding errors". Moreover, if the parameters `absolute_tolerance` or `absolute_tolerance` are chosen to be below $\sim 10^{-8}$, then double precision numbers may not be enough, and `LONG=long` is preferable. This choice comes at the cost of speed; double precision operations are usually preformed much faster. It is important to note that `LONG` defines a macro with the same name (in the `C++` examples), which then defines the macro (again in the `C++` examples) as `#define LD LONG double`. The macro `LD`, then is used as the corresponding template argument in the various classes. We point out again that if one chooses not to use the `makefile` files, the template arguments need to be known at compile-time. So the user has to define them in the code.

3. `LONGpy`: the same as `LONG`, but for the `python` interface. One should keep in mind thtat this cannot be changed inside `python` scripts. It just instructs `ctypes` what numeric type to use. Since the preferred way to compiled the shared libraries is via running "`make`" in the root directory of `MiMeS`, this variable needs to be defined inside `MiMeS/Definitions.mk`. By default, this variable is set to `long`, since this is the most stable choice in general.

4. `SOLVER`: `MiMeS` uses the ordinary differential equation (ODE) integrators of ref. [18]. Currently, there are two available choices; `Rosenbrock` and `RKF`. The former is a general embedded Rosenbrock implementation and it is used if `SOLVER=1`, while the latter is a general explicit embedded Runge-Kutta implementation and can be chosen by using `SOLVER=2` (a brief description of how these algorithms are implemented can be found in Appendix A). By default `MiMeS` uses is `SOLVER=1`, because the axion EOM tends to oscillate rapidly. However, in some cases, a high order explicit method may also work. Note that this variable defines a macro that is then used as the second template argument of the `mimes::Axion<LD,Solver,Method>` class. The preferred way to do it in the shared libraries is via the `MiMeS/Definitions.mk` file, however, the user if free to compile everything in a different way. In this case, the various `Definitions.mk` files, are not being used, and the user must define the relevant arguments in the code where `MiMeS` is used.

5. `METHOD`: Depending on the type of solver is chosen, there are some available methods. [8]

   - For `SOLVER=1`, the available methods are `METHOD=RODASPR2` and `METHOD=ROS34PW2`. The `RODASPR2` choice is a fourth order Rosenbrock-Wanner method (more information can be found in ref. [22]). The ROS34PW2 choice corresponds to a third order Rosenbrock-Wanner method [23].
   - For `SOLVER=2`, the only reliable method available in `NaBBODES` is the Dormand-Prince [24] chosen if `METHOD=DormandPrince`, which is an explicit Runge-Kutta method of seventh order.

   This variable defines a macro (with the same name) that is passed as the third template parameter of `mimes::Axion<LD,Solver,Method>` (*i.e.*`METHOD<LD>` in the place of `Method`). If the compilation is not done via the `makefile` files, the user must define the relevant template arguments in the code.

6. `CC`: the `C++` compiler that one chooses to use. The default option is `CC=g++`, which is the `GNU C++` compiler, and is available for all systems. Another option is to use the `clang` compiler, which is chosen by `CC=clang -lstdc++`. `MiMeS` is mostly tested using `g++`, but `clang` also seems to work (and the resulting executables are sometimes faster), but the user has to make sure that their version of the compiler of choice supports the `C++` 17 standard, otherwise `MiMeS` probably will not work.

7. `OPT`: Optimization level of the compiler. By default, this is `OPT=O3`, which produces executables that are marginally faster than `OPT=O1` and `OPT=O2`, but significantly faster than `OPT=O0`. There

[8] It is worth mentioning that `NaBBODES` is built in order to be a template for all possible Rosenbrock and explicit Runge-Kutta embedded methods, and one can provide their own Butcher tableau if they want to use another method, as shown in Appendix A.

is another choice, `OPT=Ofast`, but it can cause various numerical instabilities, and is generally considered dangerous – although we have not observed any problems when running `MiMeS`.

It is important to note, once again, that the variables that correspond to template arguments must be known at compile time. Thus, if the compilation is done without the help of the various `makefile` files, the template arguments must be given, otherwise compilation will fail. [9] For example, the choice `LONG=long`, `SOLVER=1`, and `METHOD=RODASPR2` will be used to compile the shared libraries (and `C++` example in `MiMeS/UserSpace/Cpp/Axion`) with `mimes::Axion<long double,1,RODASPR2<long double>>`. In order to fully understand the template arguments, the signatures of all classes and functions are given in Appendix B.

4.2 User input

4.2.1 Compile-time input

Files `MiMeS` requires files that provide data for the relativistic degrees of freedom (RDOF) of the plasma, and the anharmonic factor. Although `MiMeS` is shipped with the standard model RDOF found in [25], and a few points for $f(\theta_{\rm peak})$ introduced in eq. (2.28), the user is free change them via the corresponding variables in `Paths.mk` (in the root directory of `MiMeS`). Moreover, there is a set of data for the QCD axion mass as calculated in ref.[13]. The variables pointing to these data files are `cosmoDat`, `axMDat`, and `anFDat`, for the RDOF, axion mass, and the anharmonic factor; respectively.

The format of the files has to be the following:

- The RDOF data must be given in three columns; T (in GeV), $h_{\rm eff}$, and $g_{\rm eff}$.
- The axion mass data must be given in two columns; T (in GeV), χ (in GeV^4). Here, χ is defined as in eq. (2.2). The user can provide a function instead of data for the axion mass, by leaving the `axMDat` variable empty.
- The data for the anharmonic factor must be give in two columns $\theta_{\rm peak}$ $f(\theta_{\rm peak})$; with increasing $\theta_{\rm peak}$.

The paths to these files should be given at compile time. That is, once `Paths.mk` changes, "`bash configure.sh`" and then "`make`" must be ran. The user can change the content of the data files (without changing their paths), in order to use them without compiling `MiMeS` again. However, the user has to make sure that all the files are sorted so that the values of first column increase (with no duplicates or empty lines). In order to ensure this, it is advised to run "`bash FormatFile.sh path-to-file`" (in Appendix F there are some details on `MiMeS/src/FormatFile.sh`), in order to format the file (that should exist in "`path-to-file`") so that it complies with the requirements of `MiMeS`.

These paths are stored as strings in `MiMeS/src/misc_dir/path.hpp` at compile-time (they are defined as `constexpr`), and can be accessed once this header file is included. The corresponding variables are `cosmo_PATH`, `chi_PATH`, and `anharmonic_PATH`, for the path to data file of the plasma quantities, $\chi(T)$, and $f(\theta_{\rm peak})$; respectively. Although, the axion mass data file may be omitted – since the axion mass is defined by the user, the variable `chi_PATH` is still useful if the axion mass is defined via a data file, as it is automatically converted to an absolute path. [10]

4.2.2 Run-time input

The run-time user input is described in sec. 3. The user has to provide at least the parameters that describe the axion evolution, $\theta_{\rm ini}$ and f_a.

Moreover, the maximum allowed value of u and the minimum value of T, allow the user to decide when the integration has to stop even if the axion has not reached its adiabatic evolution. Ideally,

[9] In `C++` the template arguments are part of the definition of a class; if the template arguments are not known, the class is not even constructed.

[10] Absolute paths have the advantage to be accessible from everywhere else in the system. Thus, executables that seek the corresponding files can be called and copied easily.

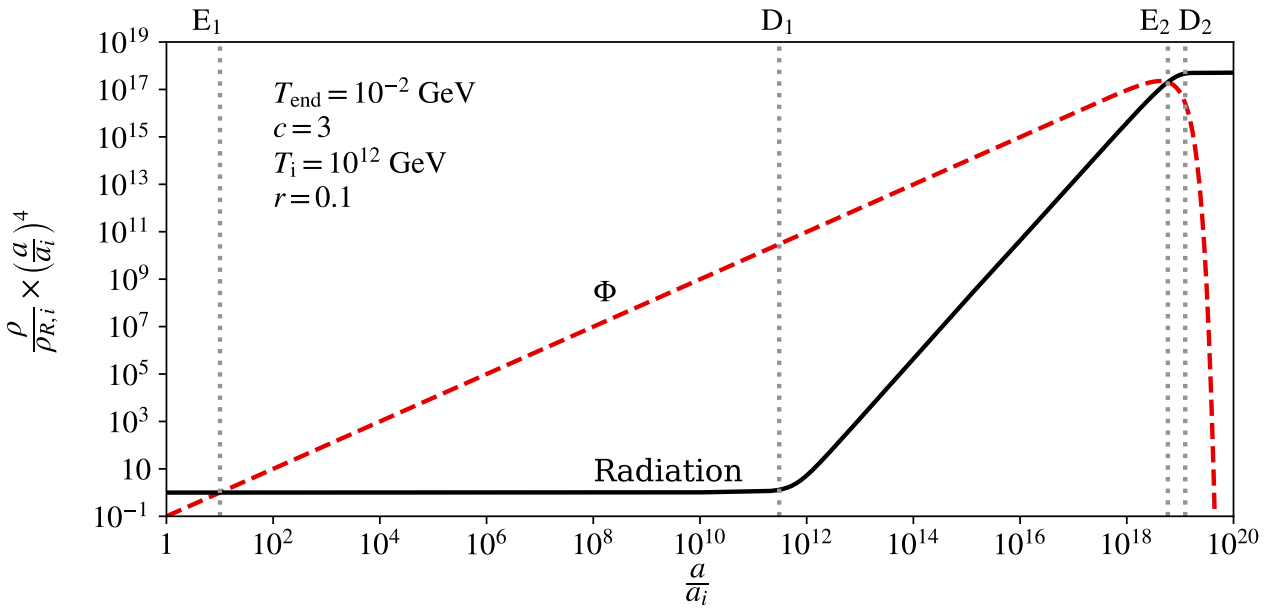


Figure 3: The energy densities of the plasma and a hypothetical decaying fluid. The parameters of the cosmological scenario $T_{\rm END} = 10^{-2}$ GeV, $c = 3$, $T_{\rm ini} = 10^{12}$ GeV, and $r = 0.1$; the parameters follow the definitions in ref. [27].

`umax`$= \infty$ and `TSTOP`$= 0$, but `MiMeS` is designed to be as general as possible and there may be cases where one needs to stop the integration abruptly. [11]

Furthermore, `ratio_ini` allows the user to choose a desired point at which the interpolation of the data in `inputFile` begins. This can save valuable computation time as well as memory, as only the necessary data are stored and searched. Generally, for `ratio_ini`$> 10^3$, the relic abundance becomes independent from `ratio_ini`, but one has to choose it carefully, in order to find a balance between accuracy and computation time.

Finally, the convergence conditions – *i.e.* `N_convergence_max` and `convergence_lim` – allow the user to decide when the adiabatic evolution of the axion begins. Generally, the relic abundance does not have a strong dependence on these parameters as long as `N_convergence_max`> 3 and `convergence_lim`$< 10^{-1}$ (*i.e.* the adiabatic invariant does not vary more that 10% for three consecutive peaks of the oscillation). However, we should note that greedy choices (*e.g.* `N_convergence_max`$=$ 100 and `convergence_lim`$= 10^{-5}$) are dangerous, as θ tends to oscillate rapidly and destabilize the differential equation solver. Therefore, these parameters should be chosen carefully, in order to ensure that integration stops when axion has reached its adiabatic evolution, without destabilising the EOM.

4.3 Complete Examples

Although one can modify the examples provided in `MiMeS/UserSpace`, in this section we show a complete example in both `C++` and `python`. The underlying cosmology is assumed to be an EMD scenario. [12]. and the evolution of the energy densities of the plasma and a fluid Φ is shown in Fig. (3). The various regime changes are indicated as $T_{\rm E_{1,2}}$ – corresponding to the first and second time where $\rho_\Phi = \rho_{\rm R}$, and $T_{\rm D_{1,2}}$ – which correspond to the start and end of entropy injection (for more precise definition see [27]). Regarding the axion, this cosmological scenario alters its evolution, since both the Hubble parameter and the entropy of the plasma change significantly. This results in a shift in the oscillation temperature, and the dilution of the axion energy density.

Moreover, we will use the QCD axion mass of ref. [13]. For the axion mass beyond the minimum

[11] These two variables are not optional, because the user must be aware of them, in order to choose them according to their needs.

[12] In terms of the parametrisation introduced in ref. [26, 27], the early matter dominated case is defined via $T_{\rm end} = 10^{-2}$ GeV, $c = 3$, $T_{\rm ini} = 10^{12}$ GeV, and $r = 0.1$.

and maximum temperatures, $T_{\min,\max}$, in the corresponding data file, we will take

$$\tilde{m}_a = \begin{cases} \dfrac{\chi(T_{\min})}{{f_a}^2} & \text{for } T < T_{\min} \\ \dfrac{\chi(T_{\max})}{{f_a}^2}\left(\dfrac{T}{T_{\max}}\right)^{-8.16} & \text{for } T > T_{\max} \end{cases} . \tag{4.1}$$

4.3.1 complete example in C++

In order to write a C++ program that uses MiMeS in order to solve the EOM 2.19, we must include the header files src/AxionMass/AxionMss.hpp and src/Axion/AxionSolve.hpp. In the example at hand, the main function should contain the definition of the axion mass

```
// use chi_PATH to interpolate the axion mass.
mimes::AxionMass<LD> axionMass(chi_PATH,0,mimes::Cosmo<LD>::mP);

/* This is the axion mass squared beyond the interpolation limits
for the current data. If you don't specify them, the axion mass
is taken to be constant beyond these limits */

/*set ma2 for T>TMax*/
long double TMax=axionMass.getTMax();
long double chiMax=axionMass.getChiMax();
axionMass.set_ma2_MAX(
   [&chiMax,&TMax](long double T, long double fa){
      return chiMax/fa/fa*std::pow(T/TMax,-8.16);
   }
);

/*set ma2 for T<TMin*/
long double TMin=axionMass.getTMin();
long double chiMin=axionMass.getChiMin();
axionMass.set_ma2_MIN(
   [&chiMin,&TMin](long double T, long double fa){
      return chiMin/fa/fa;
   }
);
```

It should be noted that, since we have used chi_PATH, we need to include the path.hpp header file from MiMeS/src/misc_dir. Moreover, we note that we have used as parameter mimes::Cosmo<LD>::mP, which is the Planck mass, as a static member of the Cosmo class (described in Appendix B), in order to interpolate the mass up to $T = m_P$, *i.e.* the entire possible mass range.

Following, the axion EOM can be solve as

```
std::string inputFile = std::string(rootDir)+
          std::string("/UserSpace/InputExamples/MatterInput.dat");

mimes::Axion<long double, 1, RODASPR2<long double> >
ax(0.1, 1e16, 500, 1e-4, 1e3, 15, 1e-2, inputFile, &axionMAss,
1e-1, 1e-8, 1e-1, 1e-11, 1e-11, 0.9, 1.2, 0.8, int(1e7));

ax.solveAxion();
```

This program should solve the axion EOM. However, one needs to print anything that they need. For example, the relic abundance is printed by adding std::cout<<ax.relic<<"\n"; after ax.solveAxion();. It should be noted that, in order to make sure that the code is compiled successfully, one should add at the top of the file any other header they need. In particular, in this case, we need to add #include<iostream>, in order to be able to use std::cout.

Parameter choice It should be noted that the string that is assigned to the variable `inputFile`, is the path of file that exists in `MiMeS/UserSpace/InputExamples`. Also, `rootDir` is a constant static (`char[]`) variable that is defined in `MiMeS/src/path.hpp`, and it is automatically generated when `bash configure.sh` is executed. Moreover, the other parameters used in the constructor are:

- `theta_i`= 0.1
- `fa`= 10^{16} GeV
- `umax` = 500
- `TSTOP`= 10^{-4} GeV
- `ratio_ini`= 10^{3}
- `N_convergence_max`= 15
- `convergence_lim`= 10^{-2}
- `initial_stepsize`= 10^{-1}
- `maximum_stepsize`= 10^{-1}
- `minimum_stepsize`= 10^{-8}
- `absolute_tolerance`= 10^{-11}
- `relative_tolerance`= 10^{-11}
- `beta`= 0.9
- `fac_max`= 1.2
- `fac_min`= 0.8
- `maximum_No_steps`= 10^{7}

Template arguments Furthermore, The first template parameter is `long double`, which means that all the numeric types (apart from integers like `maximum_No_steps`) have "long double" precision. which is useful when considering low tolerances as in this example. The second and third template parameters are responsible for the Runge-Kutta method we use. That is, n this case, the second template parameter – 1 – means that we use a Rosenbrock method, with the method being `RODASPR2`. Notice that the method also needs a template parameter that is used to declare all member variables of the `RODASPR2` class as `long double`.

Compilation Before we compile, we have to make sure that we have already executed `bash configure.sh`. Assuming that we name the file that contains the code `axionExample.cpp`, and it is located in `MiMeS/UserSpace`, an executable can be produced as

```
g++ -O3 -std=c++17 -lm -I../ -o axion axionExample.cpp
```

or

```
clang -lstdc++ -O3 -std=c++17 -lm -I../ -o axion axionExample.cpp
```

Both of these commands should create an executable that solves the axion EOM. That is, assuming we have a terminal open in `MiMeS/UserSpace`, we can run `./axion`, and get the results we chose. It should be noted that all paths that `MiMeS` uses by default are written as absolute paths when we run bash ./configure.sh. Therefore, if the `inputFile` variable is also an absolute path, the executable can be copied and used in any other directory of the same system. However, it should be preferred that executables are kept under the `MiMeS` directory, in order to be able to compile them with different data file paths if needed.

The entire code This example consists of only a few lines of code, which, including the change in the axion mass beyond the interpolation limit, is

```
#include<iostream>
//include everything you need from MiMeS
#include"src/Axion/AxionSolve.hpp"
#include"src/AxionMass/AxionMass.hpp"
#include"src/Cosmo/Cosmo.hpp"
#include"src/misc_dir/path.hpp"

int main(){
   // use chi_PATH to interpolate the axion mass.
   mimes::AxionMass<LD> axionMass(chi_PATH,0,mimes::Cosmo<LD>::mP);

   /*set m̃_a^2 for T ≥ T_max*/
   long double TMax=axionMass.getTMax();
   long double chiMax=axionMass.getChiMax();
   axionMass.set_ma2_MAX(
      [&chiMax,&TMax](long double T, long double fa){
```

```
17            return chiMax/fa/fa*std::pow(T/TMax,-8.16);
18         }
19      );
20
21      /*set m̃_a^2 for T ≤ T_min*/
22      long double TMin=axionMass.getTMin();
23      long double chiMin=axionMass.getChiMin();
24      axionMass.set_ma2_MIN(
25         [&chiMin,&TMin](long double T, long double fa){
26            return chiMin/fa/fa;
27         }
28      );
29
30
31      std::string inputFile = std::string(rootDir)+
32         std::string("/UserSpace/InputExamples/MatterInput.dat");
33
34      mimes::Axion<long double, 1, RODASPR2<long double> >
35      ax(0.1, 1e16, 500, 1e-4, 1e3, 15, 1e-2, inputFile, &axionMass,
36      1e-1, 1e-8, 1e-1, 1e-11, 1e-11, 0.9, 1.2, 0.8, int(1e7));
37
38      ax.solveAxion();
39
40      std::cout<<ax.relic<<"\n";
41
42      return 0;
43   }
```

We should point out that another example is given in `MiMeS/UserSpace/Cpp/Axion`, where all parameters are taken as command-line inputs, the various compilation-time options can be given `MiMeS/UserSpace/Cpp/Axion/Definitions.mk`, and then compiled using `make`. Therefore, the user can just modify this code in order to meet their needs. That is, using this example, one only needs to add their preferred definition of `ma2_MAX` and `ma2_MIN` – or change the `axionMass` variable (in `MiMeS/src/static.hpp`) using a function as mentioned in section 4.2.1 without writing the entire program themselves.

Alternative axion mass definition For this particular example, we could have used the approximate definition of the axion mass

$$\tilde{m_a}^2 \approx {m_a}^2 \times \begin{cases} \left(\dfrac{T_{\rm QCD}}{T}\right)^{8.16} & \text{for } T \geq T_{\rm QCD} \\ 1 & \text{for } T \leq T_{\rm QCD} \end{cases}, \tag{4.2}$$

where $T_{\rm QCD} \approx 150$ MeV and $m_a = \dfrac{3.2\times 10^{-5}\ \text{GeV}^4}{{f_a}^2}$. This can be done by substituting the `axionMass` definition (lines 9-28) with

```
1   auto ma2 = [](long double T,long double fa){
2       long double TQCD=150*1e-3;
3       long double ma20=3.1575e-05/fa/fa;
4       if(T<=TQCD){return ma20;}
5       else{return ma20*std::pow((TQCD/T),8.16);}
6   };
7   mimes::AxionMass<LD> axionMass(ma2);
```

4.3.2 complete example in `python`

In order to be able use the `AxionMass` and `Axion` classes in `python`, we need to import the corresponding modules from `MiMeS/src/interfacePy`. That is, assuming that the script from which we intend to im-

port `MiMeS/src/interfacePy/Axion/Axion.py` and `MiMeS/src/interfacePy/Axion/AxionMass.py` is in `MiMeS/UserSpace`, we can import the classes by writing the following

```
#add the relative path for MiMeS/src
from sys import path as sysPath
sysPath.append('../src')

from interfacePy.AxionMass import AxionMass #import the AxionMass class
from interfacePy.Axion import Axion #import the Axion class
from interfacePy.Cosmo import mP #import the Planck mass
```

Once everything we need is imported, we can simply follow the steps outlined in section 3.1.2. For the example at hand, we can create an instance of the `AxionMass` class as

```
# AxionMass instance
axionMass = AxionMass(r'../src/data/chi.dat',0,mP)

# This is the axion mass squared beyond the interpolation limits for the current data.
# If you don't specify them, the axion mass is taken to be constant beyond these limits
TMax=axionMass.getTMax()
chiMax=axionMass.getChiMax()
TMin=axionMass.getTMin()
chiMin=axionMass.getChiMin()

axionMass.set_ma2_MAX( lambda T,fa: chiMax/fa/fa*pow(T/TMax,-8.16) )
axionMass.set_ma2_MIN( lambda T,fa: chiMin/fa/fa )
```

Then we can simply create an instance of the `Axion` class as

```
#in python it is more convenient to use relative paths
inputFile = "./InputExamples/MatterInput.dat"

ax = Axion(0.1, 1e16, 500, 1e-4, 1e3, 15, 1e-2, inputFile, axionMass,
1e-1, 1e-8, 1e-1, 1e-11, 1e-11, 0.9, 1.2, 0.8, int(1e7))
```

Again, the parameters for the constructor are the same as in the `C++` example. The axion EOM, then, is solved using

```
ax.solveAxion()
```

In contrast to the `C++` usage of this function, this only stores the $\theta_{\rm ini}$, f_a, $\theta_{\rm osc}$, $T_{\rm osc}$, and Ωh^2 in the variables `ax.theta_i`, `ax.fa`, `ax.theta_osc`, `ax.T_osc`, and `ax.relic`; respectively. Therefore, we can print the relic abundance by calling `print(ax.relic)`. In order to get the integration points (*i.e.* the evolution of the angle and the other quantities), the quantities at the peaks, and the local integration errors, we need to call

```
#this gives you all the points of integration
ax.getPoints()

#this gives you the peaks of the oscillation
ax.getPeaks()

#this gives you local integration errors
ax.getErrors()
```

The documentation of any `python` function can be read directly inside the script using `?` as a prefix. For example, in order to see what the functionality and usage of the `getPeaks` function, we can call "`?ax.getPeaks`", and its documentation will be printed.

As already mentioned, it is important to always delete any instance of the `AxionMass` and `Axion` classes once they are not needed. In this case this is done by calling

```
del ax
del axionMass
```

Compilation of the shared library As described in section 4.2.1, we may need to change the default data file paths, or the various compilation options. This is done through the variables in `MiMeS/Definitions.mk` and `MiMeS/Paths.mk` described in section 4.1.

Once we have chosen everything according to our needs, the library can be created by opening a terminal inside the root directory of `MiMeS` and running

```
bash configure.sh
make lib/Axion_py.so
```

The entire code As in the `C++` example, this example consists of only a few lines of code. The script we described here is

```
#add the relative path for MiMeS/src
from sys import path as sysPath
sysPath.append('../src')

from interfacePy.AxionMass import AxionMass #import the AxionMass class
from interfacePy.Axion import Axion #import the Axion class
from interfacePy.Cosmo import mP #import the Planck mass

# AxionMass instance
axionMass = AxionMass(r'../src/data/chi.dat',0,mP)

# define \tilde{m}_a^2 for T <= T_min
TMin=axionMass.getTMin()
chiMin=axionMass.getChiMin()
axionMass.set_ma2_MIN( lambda T,fa: chiMin/fa/fa )

# define \tilde{m}_a^2 for T >= T_max
TMax=axionMass.getTMax()
chiMax=axionMass.getChiMax()
axionMass.set_ma2_MAX( lambda T,fa: chiMax/fa/fa*pow(TMax/T),8.16) )

#in python it is more convenient to use relative paths
inputFile = inputFile="./InputExamples/MatterInput.dat"

ax = Axion(0.1, 1e16, 500, 1e-4, 1e3, 15, 1e-2, inputFile, axionMass,
1e-1, 1e-8, 1e-1, 1e-11, 1e-11, 0.9, 1.2, 0.8, int(1e7))

ax.solveAxion()

print(ax.relic)

#once we are done we should run the destructor
del ax
del axionMass
```

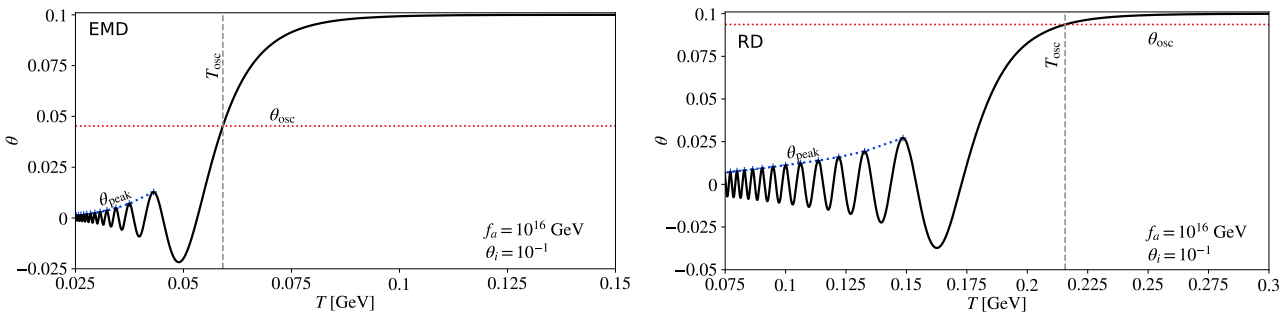
One can find a complete example, including the option to create several plots, in the script `MiMeS/UserSpace/Python/Axion.py`. Also, the same example can be used interactively, in jupyter notebook environment, that can be found in `MiMeS/UserSpace/JupyterNotebooks/Axion.ipynb`. One can read the comments, and change all different parameters, in order to examine how the results are affected.

Alternative axion mass definition In order to use the approximate axion mass as defined in eq. (4.2), we could replace the definition of the `axionMass` variable (lines 10-21) with

```
def ma2(T,fa):
   TQCD=150*1e-3
   ma20=3.1575e-05/fa/fa
```

```
    if T<=TQCD:
       return ma20;
    return ma20*(TQCD/T)**8.16
axionMass = AxionMass(ma2)
```

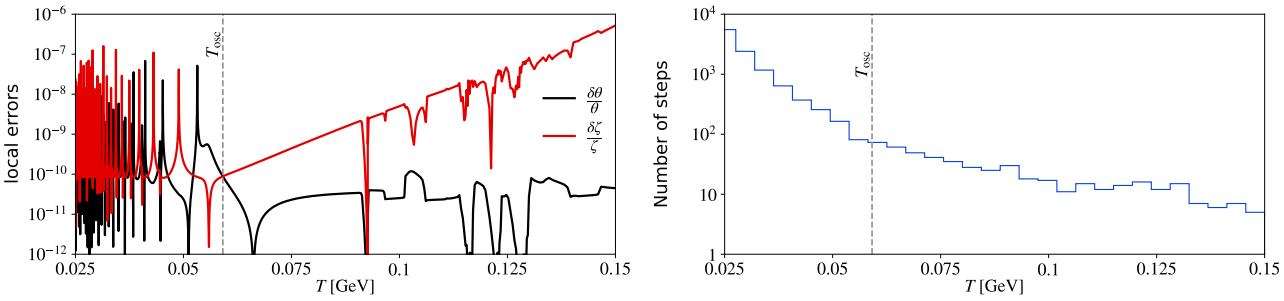
4.3.3 Results



(a) (b)

Figure 4: The evolution of the axion angle, θ, with the temperature in early matter dominated (a) and radiation dominated (b) cases. For the EMD case, the $\theta_{\rm ini}$ is chosen so that $\Omega h^2 = 0.12$.

In Fig. (4a) we show the evolution of θ for temperatures $T \in [0.025, 0.15]$ GeV, where the vertical line indicates $T_{\rm osc}$, while the horizontal one the corresponding value of θ, $\theta_{\rm osc}$. Also, the blue curve connects the peaks of the oscillation. For comparison, in Fig. (4b) we also show the evolution of the angle in a radiation dominated Universe with constant entropy (*i.e.* standard cosmological scenario). From these figures, we can see that the effect of an early matter domination reduces the amplitude of oscillation – due to the injection of entropy – as well the oscillation temperature – due to the increase of the Hubble parameter. Furthermore, the angle at $T = T_{\rm osc}$ in the EMD scenario is much smaller that the corresponding value in the standard cosmological case, since the entropy injection causes the scale factor to be larger compared to the scale factor at the same temperature. Moreover, in Fig. (5), we show the relative local error of integration as well as a histogram of the number of integration steps for 0.025 GeV $< T <$ 0.15 GeV. The local relative errors, defined in Appendix A, are shown



(a) (b)

Figure 5: (a) Local relative errors of θ and ζ for the temperature range 0.025GeV $\leq T \leq$ 0.15GeV. (b) Histogram of number of points in the same temperature range.

in Fig. (5a). The black and red lines correspond to $\delta\theta/\theta$ and $\delta\zeta/\zeta$, respectively. This figure indicates that the local errors are relatively well behaved for $T > T_{\rm osc}$, and they only start to oscillate violently once the oscillations start. However, the adaptation of the integration step seems to work, as the errors are kept below $\sim 10^{-7}$ (the relative error of *zeta* is large because $\zeta \approx 0$ before the $T_{\rm osc}$). In order to examine how the step-size is adapted to the difficulty of the problem, we also show a histogram which show how many integration steps are taken for fixed temperature intervals. [13] In this figure, we see that

[13] More precisely, the histogram is made by dividing the the temperature range 0.025 GeV $\lesssim T \lesssim$ 0.15 GeV to 30 bins

the number of integration steps increases rapidly for temperatures below the oscillation temperature. This is expected, since integration becomes more difficult as the frequency of the oscillation increases – $\tilde{m}_a/H$ increases rapidly for $T < T_{\rm osc}$. That is, the local integration error tends to increase. Thus, in order to reduce the local error, the embedded RK method we employ, reduces the step-size. This means that for $T \lesssim T_{\rm osc}$ the number of integration steps increase drastically. This is the general picture of how adaptation happens. However, one has to experiment with all the available parameters, in order to solve the axion EOM as accurately and fast as possible.

5 Acknowledgements

The author acknowledges support by the Lancaster–Manchester–Sheffield Consortium for Fundamental Physics, under STFC research grant ST/T001038/1.

6 Summary

We have introduced `MiMeS`; a header-only library written `C++` that is used to compute the axion (or ALP) relic abundance, by solving the corresponding EOM, in a user defined underlying cosmology. `MiMeS` makes only a few assumptions, which allows the user to explore a wide range ALP and cosmological scenarios.

In this manuscript, we have provided a detailed explanation on how to use `MiMeS`, by showing examples in both `C++` and `python` (paragraphs (3.1) and (4.3)). We have described the user input that is expected, and the various options available (sections (3) and (4)). Moreover, in the Appendix, we provide a detailed review of all the internal components that comprise `MiMeS`. We briefly discuss the Runge-Kutta methods that `MiMeS` uses, and show how the user can implement their own. We explain the functionality of all the classes, modules, and utilities. Also, we provide a detailed input and option guide.

In the future, `MiMeS` will be extended in several ways. First, we should implement new functionality that will allow the user to automatically compare against various experimental data (although, there is already an available module [28] that can be used for this goal). This is going to be helpful, as the user will only need to use one program (or script) to compute what is needed. We also aim to supplement `MiMeS` with the option to produce ALPs via interactions of the plasma (*e.g.* freeze-out/in), which may be useful in certain cases. Moreover, a later version of `MiMeS` may allow the user to define different initial condition for ζ as well, since there are cases where this is needed (*e.g.* [14]). Finally, `MiMeS` will continue to improve by correcting mistakes, or implementing suggestions by the community.

Appendix

A Basics of embedded Runge-Kutta Mehtods

Runge-Kutta (RK) methods are employed in order to solve an ordinary differential equation (ODE), or a system of ODEs of first order. [14] Although there are some very insightful sources in the literature (*e.g.* [29, 30, 31]) we give a brief overview of them in order to help the user to make appropriate decisions when using `MiMeS`.

The general form of a system of first order of ODEs is

$$\frac{d\vec{y}}{dt} = \vec{f}(\vec{y}, t) \ , \tag{A.1}$$

with given initial condition $\vec{y}(0)$. Also, the components of $\vec{y}$ denote the unknown functions. Note that we can always shift t to start at 0, which simplifies the notation. In order to solve the the system of

of equal size.

[14] Boundary value problems, and higher order differential equations are expressed as first order ODEs, and then solved. Similarly to eq. (2.19).

eq. (A.1), an RK method uses an iteration of the form

$$\vec{y}_{n+1} = \vec{y}_n + h \sum_{i=1}^{s} b_i \, \vec{k}_i \ , \tag{A.2}$$

with n denoting the iteration number, h the "step-size" that is used to progress t; $t_{n+1} = t_n + h$. Moreover, s, b_i and $\vec{k}_i$ define the corresponding RK method. For example, the classic Euler method is an RK method with $s = 1$, $b = 1$, and $\vec{k}_1 = \vec{f}(y_n, t_n)$. Methods with $\vec{k}$ that depends on previous step (*i.e.* $\vec{k} = \vec{k}\,[y_n, t_n]$), are called explicit, while the ones that try to also predict next step (*i.e.* $\vec{k} = \vec{k}\,[y_n, t_n; y_{n+1}, t_{n+1}]$) are called implicit. [15]

A.1 Embedded RK methods

A large category of RK methods are the so-called embedded RK methods. These methods make two estimates for the same step simultaneously – without evaluating $\vec{k}$ many times within the same iteration. Therefore, together with the iteration of eq. (A.2), a second estimate is given by

$$\vec{y}^{*}_{n+1} = \vec{y}_n + h \sum_{i=1}^{s} b^{*}_i \vec{k}_i \ , \tag{A.3}$$

with b^*_i is an extra parameter that characterise the "embedded" method of different order (typically, one order higher that the estimate A.2). The local (for the step t_{n+1}) error divided by the scale of the solution, then, estimated as

$$\Delta \equiv \sqrt{\frac{1}{N} \sum_{d=1}^{N} \left(\frac{y_{n+1,d} - y^{*}_{n+1,d}}{\Lambda_d} \right)^2} \ , \tag{A.4}$$

where n the iteration number, N the number of ODEs, d the component of $\vec{y}$, and Λ_d defined as

$$\Lambda_d = \texttt{Atol} + \max\left(|y_{n+1,d}|, |y^{*}_{n+1,d}|\right) \texttt{ Rtol} \ , \tag{A.5}$$

with `Atol` and `Rtol` the absolute and relative tolerances that characterise the desirable accuracy we want to achieve; user defined values, typically `Atol`=`Rtol`$\ll 1$. With these definitions, the desirable error is reached when $\Delta \lesssim 1$.

Step-control The definition A.4, allows us to adjust the step-size h in such way that $\Delta \approx 1$. That is, we take trial steps, and h is adapted until $\Delta \approx 1$. A simple adaptive strategy adjusts step-size, using

$$h \to \beta \, h \, \max\left[f_{\min}, \min\left(\Delta^{-\frac{1}{p+1}}, f_{\max} \right) \right] \ , \tag{A.6}$$

with p the order of the RK method ($p+1$ is the order of the embedded one), β a bias factor of the adaptive strategy (typically β is close but below 1), used to adjust the tendency of h to be somewhat smaller than what the step-control predicts. Also, f_{min} and f_{max} are the minimum and maximum allowed factors, respectively, that can multiply $\beta\ h$, used in order to avoid large fluctuations that can destabilise the process. All these parameters are chosen by the user, in order to make the step-control process as aggressive or safe as needed.

Correspondence between `MiMeS` parameters and RK ones The various parameters that `MiMeS` are described in section 4.2. The correspondence between them and the RK parameters is given in table 1.

[15] Generally, by substituting implicit methods y_{n+1} as in eq. (A.2), we end up with a system of equations that need to be solved in order to compute $\vec{k}$.

MiMeS	Runge-Kutta
absolute_tolerance	Atol
relative_tolerance	Rtol
b	β
fac_min	$f_{\min}$
fac_max	$f_{\max}$

Table 1: Correspondence between the user MiMeS user input and the RK parameters described in the text.

A.2 Explicit embedded RK methods

Explicit methods use only the information of the previous step in order to compute $\vec{k}_i$ from

$$\vec{k}_i = \vec{f}\left(\vec{y}_n + h\Big(\sum_{j=1}^{i-1} a_{ij}\vec{k}_j\Big), t_n + c_i\, h\right) , \quad \forall i = 1, 2, \ldots, s \, , \tag{A.7}$$

with a_{ij}, c_i, together with b_i and b_i^*, consist the so-called Butcher tableau of the corresponding method. For explicit methods this is usually presented as

$$\begin{array}{c|cccccc} 0 & 0 & 0 & 0 & \ldots & 0 & 0 \\ c_2 & a_{21} & 0 & 0 & \ldots & 0 & 0 \\ c_3 & a_{31} & a_{32} & 0 & \ldots & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \ddots & \vdots \\ c_s & a_{s1} & a_{s2} & a_{s3} & \ldots & a_{ss-1} & 0 \\ \hline p & b_1 & b_2 & b_3 & \ldots & b_{s-1} & b_s \\ p+1 & b_1^\star & b_2^\star & b_3^\star & \ldots & b_{s-1}^\star & b_s^\star \end{array} \tag{A.8}$$

It should be noted that $c_i = \sum_{j=1}^{s} a_{ij}$.

A.3 Rosenbrock methods

Explicit RK methods, encounter instabilities when a system is "stiff" [16]; *e.g.* when it oscillates rapidly, or has different elements at different scales. These problems are somewhat resolved by trying to predict the next step inside $\vec{k}$; *i.e.* in implicit methods. However, then one has to solve a non-linear set of equations in order to compute $\vec{y}_{n+1}$, which is generally a slow process, as the Newton method (or some variation) needs to be applied. However, there exist another way, a compromise between explicit and implicit methods. Linearly implicit RK methods, usually called Rosenbrock methods – with popular improvements as Rosenbrock-Wanner methods, introduce parameters in the diagonal of the Butcher tableau A.8 and linearise the system of non-linear equations (for details, see [30]). In these methods, $\vec{k}$ is determined by

$$\Big(\hat{I} - \gamma h\hat{J}\Big) \cdot \vec{k}_i = h\vec{f}\Big(\vec{y}_n + \sum_{j=1}^{i-1} a_{ij}\vec{k}_j, t_n + c_i h\Big) + h^2 \left(\gamma + \sum_{j=1}^{i-1}\gamma_{ij}\right) \frac{\partial \vec{f}}{\partial t} + h\hat{J} \cdot \sum_{j=1}^{i-1}\gamma_{ij}\vec{k}_j \, , \tag{A.9}$$

which is written in such a way that everything is evaluated at $t = t_n$. In eq. (A.9), $\hat{J} = \dfrac{\partial \vec{f}}{\partial \vec{y}}$ the Jacobian of the system of ODEs, $\hat{I}$ the unit matrix with dimension equal to the number of ODEs. Moreover, γ and γ_{ij} are parameters that characterise the method (along with a_{ij}, c_i, b_i, and b_i^*).

[16] A definition of stiffness can be found in [30, 29].

Implementing a new Butcher tableau in NaBBODES As already mentioned, MiMeS uses NaBBODES, which supports the implementation of new Butcher tableaux. This is done by adding a new class (or struct) inside the header file METHOD.hpp that can be found in MiMeS/src/NaBBODES/RKF for the explicit RK and MiMeS/src/NaBBODES/Rosenbrock for the Rosenbrock embedded methods. All the new parameters must be public, constexpr static variables, of a type that is the template parameter of the method. For example, the Heun-Euler method can be implemented by adding the following code in MiMeS/src/NaBBODES/RKF/METHOD.hpp

```
/*-----Implementation of the Heun-Euler method-----*/
/*-----It shouldn't be used for MiMeS, since it is likely to fail-----*/

//LD is the numeric type (\ie double, or long double).
template<class LD>
//use struct because its variables are public by default.
struct HeunEuler{
   static constexpr unsigned int s=2; // HeunEuler is a 2-stage RK
   static constexpr unsigned int p=1; // first order, with embedded p+1=2

   //these are aliases for the arrays
   using arr=std::array<LD,s>;
   using arr2=std::array<std::array<LD,s>,s>;

   static constexpr arr b={0.5,0.5}; // this is b_i
   static constexpr arr bstar={0.5,0.5}; // this is b_i^*
   static constexpr arr c={0,1}; // remember that c_i = \sum_{j=1}^{s} a_{ij}

   // this is a_{ij}
   static constexpr arr2 a={
      arr{0,0},
      arr{1,0}
   };
};
```

In order to implement the ROS3w [23] method, one can add the following code in the header file MiMeS/src/NaBBODES/Rosenbrock/METHOD.hpp

```
/*-----Implementation of the ROS3w method-----*/
/*-----It shouldn't be used, since it is likely to fail to produce a result-----*/


//LD is the numeric type (i.e. double, or long double).
template<class LD>
struct ROS3w{
   static constexpr unsigned int s=3; // 3-stage
   static constexpr unsigned int p=2; // second order (embedded order is p+1)

   //aliases for the arrays
   using arr=std::array<LD,s>;
   using arr2=std::array<std::array<LD,s>,s>;

   static constexpr arr b={0.25,0.25,0.5 }; // this is b_i
   // this is b_i^*
   static constexpr arr bstar={ 0.746704703274,0.1144064078371,0.1388888888888};

   static constexpr arr c={0,2/3.,4/3.}; // remember that c_i = \sum_{j=1}^{s} a_{ij}
   static constexpr LD gamma=0.4358665215084; // this is \gamma

   // this is a_{ij}
```

```
      static constexpr arr2 a={
         arr{0,0,0},
         arr{2/3.,0,0},
         arr{2/3.,2/3.,0}
      };

      // this is γ_ij
      static constexpr arr2 g={
         arr{0,0,0},
         arr{0.3635068368900,0,0},
         arr{-0.8996866791992,-0.1537997822626,0}
      };
   };
```

How to compile `MiMeS` in order to use the newly implemented method Once a new Butcher tableau is implemented, the `mimes::Axion` class can use it. This class, just needs the name of method assigned to the corresponding template argument; `Method`. A convenient way to do this, is to define a macro using the `-D` flag of the compiler, and use macro as the corresponding template parameter of the `mimes::Axion` class. Alternatively, if one uses the `makefile` files, a method can be chosen by adding it in the corresponding `Definitions.mk` file; *e.g.* as `METHOD=ROS3w` for the `ROW3` method. [17]

B `C++` classes

`MiMeS` is designed as an object-oriented header-only library. That is, all the basic components of the library are defined as classes inside header files. All the classes relevant to the use of `MiMeS` are under the namespace `mimes`.

B.1 `Cosmo` class

The `mimes::Cosmo<LD>` class is responsible for interpolation of the various quantities of the plasma. Its header file is `MiMeS.src/Cosmo/Cosmo.hpp`, and needs to be included in order to use this class. The template parameter `LD` is the numeric type that will be used, *e.g.* `double`. The constructor of this class is

```
template<class LD>
mimes::Cosmo<LD>(std::string cosmo_PATH, LD minT=0, LD maxT=mimes::Cosmo<LD>::mP)
```

The argument `cosmo_PATH` is the path of the data file that contains T (in GeV), $h_{\rm eff}$, $g_{\rm eff}$, with increasing T. The parameters `minT` and `maxT` are minimum and maximum interpolation temperatures. These temperatures are just limits, and the action interpolation is done between the closest temperatures in the data file. Moreover, beyond the interpolation temperatures, both $h_{\rm eff}$ and $g_{\rm eff}$ are assumed to be constants.

Interpolation of the RDOF, allows us to define various quantities related to the plasma; *e.g.* the entropy density is defined as $s=\frac{2\pi^2}{45}h_{\rm eff}T^3$. These quantities are given as the member functions:

- `template<class LD> LD mimes::Cosmo<LD>::heff(LD T)`: $h_{\rm eff}$ as a function of T.
- `template<class LD> LD mimes::Cosmo<LD>::geff(LD T)`: $g_{\rm eff}$ as a function of T.
- `template<class LD> LD mimes::Cosmo<LD>::dheffdT(LD T)`: $dh_{\rm eff}/dT$ as a function of T.
- `template<class LD> LD mimes::Cosmo<LD>::dgeffdT(LD T)`: $dg_{\rm eff}/dT$ as a function of T.
- `template<class LD> LD mimes::Cosmo<LD>::dh(LD T)`: $\delta_h=1+\frac{1}{3}\frac{d\log h_{\rm eff}}{d\log T}$ as a function of T.

[17] One needs to make sure to use the correct `Solver` template argument (or `SOLVER` variable in the `Definitions.mk` files), otherwise compilation will fail.

- `template<class LD> LD mimes::Cosmo<LD>::s(LD T)`: The entropy density of the plasma as a function of T.
- `template<class LD> LD mimes::Cosmo<LD>::rhoR(LD T)`: The energy density of the plasma as a function of T.
- `template<class LD> LD mimes::Cosmo<LD>::Hubble(LD T)`: The Hubble parameter assuming radiation dominated expansion as a function of T.

Moreover, there are several cosmological quantities are given as members variables:

- `template<class LD> constexpr static LD mimes::Cosmo<LD>::T0`: CMB temperature today [32] in GeV.
- `template<class LD> constexpr static LD mimes::Cosmo<LD>::h_hub`: Dimensionless Hubble constant [32].
- `template<class LD> constexpr static LD mimes::Cosmo<LD>::rho_crit`: Critical density [32] in GeV^3.
- `template<class LD> constexpr static LD mimes::Cosmo<LD>::relicDM_obs`: Central value of the measured DM relic abundance [7].
- `template<class LD> constexpr static LD mimes::Cosmo<LD>::mP`: Planck mass [32] in GeV.

B.2 `AnharmonicFactor` class

The class `mimes::AnharmonicFactor<LD>` is responsible interpolating the anharmonic factor as defined in eq. (2.28). The corresponding header file is `MiMeS/src/AnharmonicFactor/AnharmonicFactor.hpp`.

The constructor of this class is

```
1  template<class LD>
2  mimes::AnharmonicFactor<LD>(std::string anharmonic_PATH)
```

Again, the template argument `LD` is a numeric type, and the `anharmonic_PATH` string is the path of the data file with data for θ_{peak} (which should be in increasing order) and $f(\theta_{\text{peak}})$.

The member function that `MiMeS` uses is the overloaded call operator

```
1  template<class LD> LD mimes::AnharmonicFactor<LD>::operator()(LD theta_peak)
```

This function returns the value of the anharmonic factor at θ_{peak} =`theta_peak`. Although, there is no need to call this function beyond the interpolation limits (as long as the data file contains $0 \leq \theta_{\text{peak}} \leq \pi$), it is important to note that the anharmonic factor is taken to be constant beyond these limits.

B.3 `AxionMass` class

The `mimes::AxionMass<LD>` class is responsible for the definition of the axion mass. The header file of this class is `MiMeS/src/AxionMass/AxionMass.hpp`. Its usage and member functions are described in the examples given in sections (3.1) and (4.3). However, it would be helpful to outline them here.

The class has two constructors. The first one is

```
1  template<class LD>
2  mimes::AxionMass<LD>(std::string chi_PATH, LD minT=0, LD maxT=mimes::Cosmo::mP)
```

The first argument, `chi_PATH`, is the path to a data file that contains two columns; T (in GeV) and χ (in GeV^4), with increasing T. The arguments `minT` and `maxT` are the interpolation limits. These limits are used in order to stop the interpolation in the closest temperatures that exist in the data file. That is the actual interpolation limits are $T_{\text{min}} \geq$`minT` and $T_{\text{max}} \leq$`maxT`. Beyond these limits, by default, the axion mass is assumed to be constant. However, this can be changed by using the member functions

```
void set_ma2_MIN(std::function<LD(LD,LD)> ma2_MIN)
void set_ma2_MAX(std::function<LD(LD,LD)> ma2_MAX)
```

Here, `ma2_MIN` and `ma2_MAX` are functors that define the axion mass squared beyond the interpolation limits. In order to ensure that the axion mass is continuous, usually we need $T_{\rm min}$, $T_{\rm max}$, $\chi(T_{\rm min})$, and $\chi(T_{\rm max})$. These values can be obtained using the member functions

- `template<class LD> LD mimes::AxionMass<LD>::getTMin()`: This function returns the minimum interpolation temperature, $T_{\rm min}$.
- `template<class LD> LD mimes::AxionMass<LD>::getTMax()`: This function returns the maximum interpolation temperature, $T_{\rm max}$.
- `template<class LD> LD mimes::AxionMass<LD>::getChiMin()`: This function returns $\chi(T_{\rm min})$.
- `template<class LD> LD mimes::AxionMass<LD>::getChiMax()`: This function returns $\chi(T_{\rm max})$.

An alternative way to define the axion mass is via the constructor

```
template<class LD>
mimes::AxionMass<LD>(std::function<LD(LD,LD)> ma2)
```

Here, the only argument is the axion mass squared, $\tilde{m}_a$, defined as a callable object.

Once an instance of the class is defined, we can get ${\tilde{m}_a}^2$ using the member function

```
template<class LD> LD mimes::AxionMass<LD>::ma2(LD T, LD fa)
```

We should note that `ma2` is a public `std::function<LD(LD,LD)>` member variable. Therefore, it can be assigned using the assignment operator. However, in order to change its definition, we can also use the following member function:

```
template<class LD> void mimes::AxionMass<LD>::set_ma2(std::function<LD(LD,LD)> ma2)
```

B.4 `AxionEOM` class

The `mimes::AxionEOM<LD>` class is not useful for the user. However, it is responsible for the interpolation of the underlying cosmology, and the definition of the axion EOM 2.19, which is passed to the ODE solver of `NaBBODES`.

The constructor of the class is

```
template<class LD>
mimes::AxionEOM<LD>(LD fa, LD ratio_ini, std::string inputFile, AxionMass<LD> *axionMass)
```

The role of the arguments are discussed in section (3.1.1) and (4.2.2) as well as in table 2. One the instance is created, the interpolations are constructed by calling the member function

```
template<class LD> void mimes::AxionEOM<LD>::makeInt()
```

Then, the temperature as a function of $u = \log a/a_{\rm ini}$, is given via the member function

```
template<class LD> LD mimes::AxionEOM<LD>::Temperature(LD u)
```

Another useful member function is

```
template<class LD> LD mimes::AxionEOM<LD>::logH2(LD u)
```

This function returns $\log H^2$ as a function of u. Moreover, its derivative, $\frac{d\log H^2}{du}$, is computed using

```
template<class LD> LD mimes::AxionEOM<LD>::dlogH2du(LD u)
```

It should be noted that the highest interpolation temperature is determined by `ratio_ini` while the lower interpolation temperature is the one given in the data file `inputFile`. Beyond these limits, all functions are assumed to be constant. Therefore, one should be careful, and choose an appropriate `ratio_ini`, and provide a lower temperature at which any entropy injection has stopped and the axion has reached its adiabatic evolution.

Finally, the actual EOM is given an overloaded call operator

```
template<class LD>
void mimes::AxionMass<LD>::operator()(std::array<LD,2> &lhs, std::array<LD,2> &y, LD u)
```

Here, the inputs are $u = \log a/a_{\rm ini}$, `y[0]`$= \theta$, and `y[1]`$= \zeta$; which are used to calculate the components of the EOM, with `lhs[0]`$= \frac{d\theta}{du}$ and `lhs[1]`$= \frac{d\zeta}{du}$.

B.5 `Axion` class

The `mimes::Axion<LD,Solver,Method>` class is the class that combines all the others, and actually solves the axion EOM 2.19. Its header file is `MiMeS/src/Axion/AxionSolve.hpp`, and its constructor is

```
template<class LD, const int Solver, class Method>
mimes::Axion< LD, Solver, Method<LD> >(LD theta_i, LD fa, LD umax, LD TSTOP,
LD ratio_ini, unsigned int N_convergence_max, LD convergence_lim,
std::string inputFile, AxionMass<LD> *axionMass, LD initial_step_size=1e-2,
LD minimum_step_size=1e-8, LD maximum_step_size=1e-2, LD absolute_tolerance=1e-8,
LD relative_tolerance=1e-8, LD beta=0.9, LD fac_max=1.2, LD fac_min=0.8,
unsigned int maximum_No_steps=10000000)
```

The various arguments are discussed in section (3.1.1) and (4.2.2); and outlined in table 2.

The member function responsible for solving the EOM is

```
template<class LD, const int Solver, class Method>
void mimes::Axion< LD, Solver, Method<LD> >::solveAxion()
```

Once this function finishes, the results are stored in several member variables.

The quantities $a/a_{\rm ini},\ T,\ \theta,\ \zeta, \rho_a$, at the integration steps are stored in

```
template<class LD, const int Solver, class Method>
std::vector< std::vector<LD> > mimes::Axion< LD, Solver, Method<LD> >::points
```

The quantities $a/a_{\rm ini},\ T,\ \theta,\ \zeta, \rho_a,\ J$, at the peaks of the oscillation are stored in

```
template<class LD, const int Solver, class Method>
std::vector< std::vector<LD> > mimes::Axion< LD, Solver, Method<LD> >::peaks
```

Note that these points are computed using linear interpolation between two integration points with a change in the sign of ζ.

The local integration errors for θ and ζ are stored in

```
template<class LD, const int Solver, class Method>
std::vector<LD> mimes::Axion< LD, Solver, Method<LD> >::dtheta

template<class LD, const int Solver, class Method>
std::vector<LD> mimes::Axion< LD, Solver, Method<LD> >::dzeta
```

Moreover, the oscillation temperature, $T_{\rm osc}$, and the corresponding values of $a/a_{\rm ini}$ and θ are given in

```
template<class LD, const int Solver, class Method>
LD mimes::Axion< LD, Solver, Method<LD> >::T_osc

template<class LD, const int Solver, class Method>
LD mimes::Axion< LD, Solver, Method<LD> >::a_osc

template<class LD, const int Solver, class Method>
LD mimes::Axion< LD, Solver, Method<LD> >::theta_osc
```

Also, the entropy injection between the last peak ($T = T_{\rm peak}$) and today ($T = T_0$), γ (defined as in eq. (2.31)), is given in

```
template<class LD, const int Solver, class Method>
LD mimes::Axion< LD, Solver, Method<LD> >::gamma
```

The relic abundance is stored in the following member variable

```
template<class LD, const int Solver, class Method>
LD mimes::Axion< LD, Solver, Method<LD> >::relic
```

We can set another initial condition, $\theta_{\rm ini}$, using

```
template<class LD, const int Solver, class Method>
void mimes::Axion< LD, Solver, Method<LD> >::setTheta_i(LD theta_i)
```

We should note that running this function all variables are cleared. So we lose all information about the last time `axionSolve()` ran.

In case the mass of the axion is changed, we also need to remake the interpolation (*i.e.* run `mimes::AxionEOM::makeInt()`). This is done using

```
template<class LD, const int Solver, class Method>
void mimes::Axion< LD, Solver, Method<LD> >::restart()
```

Again, this function clears all member variables. So it should be used with caution.

Finally, there is static `mimes::Cosmo<LD>` member variable

```
template<class LD, const int Solver, class Method>
static mimes::Cosmo<LD> mimes::Axion< LD, Solver, Method<LD> >::plasma
```

This variable can be used without an instance of the `mimes::Axion<LD,Solver,Method>` class.

C `python` interface

The various `python` modules, classes, and functions are designed to work exactly in the same way as the ones in `C++`. All the modules are located in `src/interfacePY`, so it is helpful to add the `MiMeS/src` path to the system path at the top of every script that uses `MiMeS`. This is done by adding

```
from sys import path as sysPath
sysPath.append('path_to_src')
```

The available models are `Cosmo`, `AxionMass`, and `Axion`, each defines a class with the same name.

C.1 `Cosmo` class

The `Cosmo` module defines the `Cosmo` class, which contains information about the plasma. The relevant shared library (`lib/libCosmo.so`) is obtained by compiling `MiMeS/src/Cosmo/Cosmo.cpp` using `make lib/libCosmo.so`.

The class can be imported by running

```
from interfacePy.Cosmo import Cosmo
```

Its constructor is

```
Cosmo(cosmo_PATH, minT=0, maxT=mP)
```

The argument `cosmo_PATH` is the path (a string) of a data file that contains T (in GeV), $h_{\rm eff}$, $g_{\rm eff}$, with accenting T. The second and third arguments, `minT` and `maxT`, are minimum and maximum interpolation temperatures, with the interpolation being between the closest temperatures in the data file. Moreover, beyond these limits, both $h_{\rm eff}$ and $g_{\rm eff}$ are assumed to be constants. It is important to note that the class creates a `void` pointer that gets recasted to `mimes::Cosmo<LD>` in order to call the various member functions. This means that once an instance of `Cosmo` is no longer needed, it must be deleted, in order to free the memory that it occupies. An instance, say `cosmo`, is deleted using

```
del cosmo
```

The member functions of this class are:

- `Cosmo.heff(T)`: $h_{\rm eff}$ as a function of T.
- `Cosmo.geff(T)`: $g_{\rm eff}$ as a function of T.
- `Cosmo.dheffdT(T)`: $dh_{\rm eff}/dT$ as a function of T.
- `Cosmo.dgeffdT(T)`: $dg_{\rm eff}/dT$ as a function of T.
- `Cosmo.dh(T)`: $\delta_h = 1 + \frac{1}{3}\frac{d\log h_{\rm eff}}{d\log T}$ as a function of T.
- `Cosmo.s(T)`: The entropy density of the plasma as a function of T.
- `Cosmo.rhoR(T)`: The energy density of the plasma as a function of T.
- `Cosmo.Hubble(T)`: The Hubble parameter assuming radiation dominated expansion as a function of T.

The several cosmological quantities are given as members variables:

- `Cosmo.T0`: CMB temperature today [32] in GeV.
- `Cosmo.h_hub`: Dimensionless Hubble constant [32].
- `Cosmo.rho_crit`: Critical density [32] in GeV^3.
- `Cosmo.relicDM_obs`: Central value of the measured DM relic abundance [7].
- `Cosmo.mP`: Planck mass [32] in GeV.

Note that these values can be directly imported from the module, without declaring an instance of the class, as

```
from interfacePy.Cosmo import T0, h_hub, rho_crit, relicDM_obs, mP
```

C.2 `AxionMass` class

The `AxionMass` class is defined in the module with the same name that can be found in the directory `MiMeS/src/interfacePy/AxionMass`. This class is responsible for the definition of the axion mass. This module loads the corresponding shared library from `MiMeS/lib/libma.so`, which is created by compiling `MiMeS/src/AxionMass/AxionMass.cpp` using "`make lib/libma.so`". Its usage is described in the examples given in sections (3.1) and (4.3). Moreover, this class is used in the same way as `mimes::AxionMass<LD>`. However, we should append in this section the definition of its member functions in `python`.

The class is imported using

```
from interfacePy.AxionMass import AxionMass
```

The constructor is

```
AxionMass(*args)
```

and can be used in two different ways.

First, one can pass three arguments, *i.e.*

```
AxionMass(chi_PAT, minT, maxT)
```

The first argument is the path to a data file that contains two columns; T (in GeV) and *chi* (in GeV^4), with increasing T. The arguments `minT` and `maxT` are the interpolation limits. These limits are used in order to stop the interpolation in the closest temperatures in `chi_PATH`. That is the actual interpolation limits are $T_{\rm min} \geq$`minT` and $T_{\rm max} \leq$`maxT`. Beyond these limits, by default, the axion mass is assumed to be constant. However, this can be changed by using the member functions

```
AxionMass.set_ma2_MIN(ma2_MIN)
AxionMass.set_ma2_MAX(ma2_MAX)
```

Here, `ma2_MIN(T,fa)` and `ma2_MAX(T,fa)`, are functions (not any callable object), should take as arguments `T` and `fa`, and return the axion mass squared beyond the interpolation limits. In order to ensure that the axion mass is continuous, usually we need $T_{\rm min}$, $T_{\rm max}$, $\chi(T_{\rm min})$, and $\chi(T_{\rm max})$. These values can be obtained using the member functions

- `AxionMass.getTMin()`: This function returns the minimum interpolation temperature, $T_{\rm min}$.
- `AxionMass.getTMax()`: This function returns the maximum interpolation temperature, $T_{\rm max}$.
- `AxionMass.getChiMin()`: This function returns $\chi(T_{\rm min})$.
- `AxionMass.getChiMax()`: This function returns $\chi(T_{\rm max})$.

An alternative way to define the axion mass is via the constructor

```
AxionMass(ma2)
```

Here, `ma2(T,fa)` is a function (not any callable object) that takes T (in GeV) and f_a, and returns $m_a^2(T)$ (in GeV). As in the other `python` classes, once the instances of this class are no longer needed, they must be deleted using the destructor, `del`.

The member function that returns $\tilde{m}_a{}^2$ is

```
AxionMassma2(T,fa)
```

We should note that another `ma2` can be changed using the following member function:

```
AxionMass.set_ma2(ma2)
```

Again, `ma2(T,fa)` is a function (not any callable object) that takes T (in GeV) and f_a, and returns $m_a^2(T)$ (in GeV).

C.3 `Axion` class

The class `Axion`, solves the axion EOM 2.19. This class is defined in `MiMeS/interfacePy/Axion/Axion.py`, and imports the corresponding shared library. This library is compiled by running `make lib/Axion_py.so`, and its source file is `MiMeS/src/Axion/Axion-py.cpp`. As in the previous classes, its usage is similar to the `C++` version.

Its constructor is

```
Axion(theta_i, fa, umax, TSTOP,
ratio_ini, unsigned int N_convergence_max, convergence_lim,
inputFile, axionMass, initial_step_size=1e-2,
minimum_step_size=1e-8, maximum_step_size=1e-2, absolute_tolerance=1e-8,
relative_tolerance=1e-8, beta=0.9, fac_max=1.2, fac_min=0.8,
unsigned int maximum_No_steps=10000000)
```

Again, the various arguments are discussed in section (3.1.1) and (4.2.2); and can also be found in table 2. Notice one important difference between this and the `C++` verison of this class; the instance of `AxionMass`, `axionMass`, is passed by value (there are no pointers in `python`). However, internally, the constructor converts this instance to a pointer, which is then passed to the underlying `C` function responsible for creating the relevant instance.

The member function responsible for solving the EOM is

```
Axion.solveAxion()
```

Once this function is finished, the following member funcions are available

- `Axion.T_osc`: the oscillation temperature, $T_{\rm osc}$, in GeV.
- `Axion.a_osc`: $\dfrac{a}{a_{\rm ini}}$ at the oscillation temperature.
- `Axion.theta_osc`: $\theta_{\rm osc}$, *i.e.* θ at $T_{\rm osc}$.
- `Axion.gamma`: the entropy injection between the last peak ($T = T_{\rm peak}$) and today ($T = T_0$), γ (defined as in eq. (2.31)).
- `Axion.relic` The relic abundance of the axion.

The evolution of $a/a_{\rm ini}$, T, θ, ζ, ρ_a at the integration steps, is not automatically accessible to user, but they can be made so using

```
Axion.getPoints()
```

Then, the following member variables are filled

- `Axion.a`: The scale factor over its initial value, $\dfrac{a}{a_{\rm ini}}$.
- `Axion.T`: The temperature in GeV.
- `Axion.theta`: The axion angle, θ.
- `Axion.zeta`: The derivative of θ, $\zeta \equiv \dfrac{d\theta}{d\log(a/a_{\rm ini})}$.
- `Axion.rho_axion`: The axion energy density in GeV^4.

Moreover, the function

```
Axion.getPeaks()
```

fills the (`numpy`) arrays `Axion.a_peak`, `Axion.T_peak`, `Axion.theta_peak`, `Axion.zeta_peak`, `Axion.rho_axion_peak`, and `Axion.adiabatic_invariant` with the quantities $a/a_{\rm ini}$, T, θ, ζ, ρ_a, J, at the peaks of the oscillation. These points are computed using linear interpolation between two integration points with a change in the sign of ζ.

The local integration errors for θ and ζ are stored in `Axion.dtheta` and `Axion.dzeta`, after the following function is run

```
Axion.getErrors()
```

Another initial condition, $\theta_{\rm ini}$, can be used without declaring a new instance using

```
Axion.setTheta_i(theta_i)
```

We should note that running this function all variables are cleared.

As in the previous `python` classes, once an instance of the `Axion` class is no longer needed, it needs to be deleted, by calling the destructor, `del`.

Important difference between the `C++` version Since the axion mass is passed by value in the constructor, a change of the `AxionMass` instance has no effect on the `Axion` instance that uses it. Therefore, if the definition of the axion mass changes, one has to declare a new instance of the `Axion` class. The new instance can be named using the name of the previous one, if the latter is deleted by running its destructor.

D WKB module

The WKB module can be used to calculate the axion relic abundance using the WKB approximation discussed in section 2. The module can be imported using

```
from interfacePy import WKB
```

It contains the definition of a function that returns the relic abundance using the WKB approximation, which is

```
WKB.relic(Tosc, theta_osc ,ma2, gamma=1,
          cosmo=Cosmo(_PATH_+r'/src/data/eos2020.dat',0,Cosmo.mP))
```

Here, Tosc is the oscillation temperate, ma2(T,fa) is $m_a{}^2(T)$ as a function that takes T and f_a as arguments, cosmo an instance of the Cosmo class, and gamma the entropy injection (as defined in eq. (2.13)).

Moreover, there is a function that helps to determine $T_{\rm osc}$ and γ, which is

```
WKB.getPoints(fa, ma2, inputFile, cosmo=Cosmo(_PATH_+r'/src/data/eos2020.dat',0,Cosmo.mP))
```

The arguments fa, inputFile,f_a, and cosmo the path to a file that describes the cosmology (described in table 2), f_a (in GeV), an instance of the Cosmo class, respectively. This function returns γ (the entropy injection between Tosc and today) and $T_{\rm osc}$.

E The ScanScript module

The python interface of MiMeS has two simple classes that help to scan over $\theta_{\rm ini}$ and f_a, in parallel. These two classes can be imported from the module ScanScript.

Tis module is based on the bash script MiMeS/src/util/parallel_scan.sh, which automatically performs scans in parallel. This script is used as

```
bash src/util/parallel_scan.sh executable cpus inputFile
```

Here, executable is the path to an executable, cpus the number of instances of the executable to launch simultaneously, and inputFile a file that contains the arguments the executable expects. This file should contain arguments for the executable in each line. This script, then, separates all in arguments in the inputFile in batches of size cpus, and runs one batch at a time.

E.1 The Scan class

The Scan class writes a time-coded file (so it would be unique) with columns that correspond to $\theta_{\rm ini}$, f_a (GeV), $\theta_{\rm osc}$, $T_{\rm osc}$ (in GeV), and Ωh^2, for every combination of $\theta_{\rm ini}$ and f_a that are passed as input.

This class is imported using

```
from interfacePy.ScanScript import Scan
```

Its constructor is

```
ScanScript.Scan(cpus,table_fa,table_theta_i,umax,TSTOP,ratio_ini,
            N_convergence_max,convergence_lim,inputFile,
            PathToCppExecutable, break_after=5*60,break_time=5,break_command='',
            initial_step_size=1e-2, minimum_step_size=1e-8, maximum_step_size=1e-2,
            absolute_tolerance=1e-8, relative_tolerance=1e-8,
            beta=0.9, fac_max=1.2, fac_min=0.8, maximum_No_steps=int(1e7))
```

Some arguments are the same as in the Axion class, and their definition can be found in table 2. The other arguments are

- cpus: number of points to run simultaneously (number of cpus available).
- table_fa: table of f_a to scan.

- `table_theta_i`: table of $\theta_{\rm ini}$ to scan
- `PathToCppExecutable`: path to an executable that takes `theta_i`, `fa`, `umax`, `TSTOP`, `ratio_ini`, `N_convergence_max`, `convergence_lim`, and `inputFile`; and prints `theta_i`, `fa`, `theta_osc`, `T_osc`, and `relic`.
- `break_after`, `break_time`: take a break after `break_after` seconds for `break_time` seconds.
- `break_command` (optional): before it takes a break, run this system command (this may be a script to send the results via e-mail, or back them up).

The scan runs using the method

```
Scan.run()
```

For every value of f_a, this method calls another one `Scan.run_batch()`, which in turn calls the `bash` script `MiMeS/src/util/parallel_scan.sh` with arguments `PathToCppExecutable`, `cpus`, and a file that contains all the inputs for `PathToCppExecutable` for all values in `table_theta`.

As the scan runs, it prints the number of batches that have been evaluated, the mean time it takes to evaluate one batch, and an estimate of the remaining time. It should be noted that if the scan exits before it finishes, the next run will continue from the point at which it was stopped even if the inputs have changed. In order to start from the beginning, the user must delete the file `count._mimes_`.

E.2 The `ScanObs` class

The `ScanObs` class scans for different values of f_a (given as a table) and finds the value of $\theta_{\rm ini}$ closer to the observed DM relic abundance. The result file is a time-coded file with columns correspond that to $\theta_{\rm ini}$, f_a (in GeV), $\theta_{\rm osc}$, $T_{\rm osc}$ (in GeV), and Ωh^2. It is imported using

```
from interfacePy.ScanScript import ScanObs
```

Its constructor is

```
ScanScript.ScanObs(cpus,table_fa,len_theta,umax,TSTOP,ratio_ini,
            N_convergence_max,convergence_lim,inputFile,axionMass,
            PathToCppExecutable, relic_obs,relic_err_up,relic_err_low,
            cosmo=Cosmo(_PATH_+r'/src/data/eos2020.dat',0,Cosmo.mP),
            break_after=5*60,break_time=5,break_command='',
            initial_step_size=1e-2, minimum_step_size=1e-8, maximum_step_size=1e-2,
            absolute_tolerance=1e-8, relative_tolerance=1e-8,
            beta=0.9, fac_max=1.2, fac_min=0.8, maximum_No_steps=int(1e7)
```

Here, `len_theta` is the number of different values of θ to be used in the search of a value closer to the central value of the observed DM relic abunance. The arguments `relic_obs`, `relic_err_up`, `relic_err_low` are the central value of Ωh^2, and its upper and lower error; respectively. All the other arguments have already been defined previously.

The scan runs using the method

```
ScanObs.run()
```

For every value of f_a, this method first calculates the relic abundance for $\theta_{\rm ini} \ll 1$, and finds the value of $\theta_{\rm ini}$ that would result in $\Omega h^2 =$`relic_obs`; ${\theta_{\rm ini}}^{\rm (approx)}$. This is easy to do, since for $\theta_{\rm ini} \ll 1$, the EOM becomes independent of $\theta_{\rm ini}$.

Then, it creates an array, `table_theta`, of size `len_theta` that contains values of $\theta_{\rm ini}$ between $Min(0.85\ {\theta_{\rm ini}}^{\rm (approx)}, 1)$ and $Max(1.2\ {\theta_{\rm ini}}^{\rm (approx)}, \pi)$. Then it calls `ScanObs.run_batch()`, which works as `Scan.run_batch()`.

Similarly to the `Scan` class, if the scan exits before it finishes, the next run will continue from the point at which it was stopped. In order to start from the beginning, delete the file `count._mimes_`.

E.3 FT class

The FT class, defined in FT module that can be found in src/interfacePy/FT, is used to format the ticks when plotting using matplotlib [33]. The class can be imported using

```
from interfacePy.FT import FT
```

The constructor of the class is

```
FT(_M_xticks,_M_yticks, _M_xticks_exception,_M_yticks_exception, _m_xticks,_m_yticks,
      xmin,xmax,ymin,ymax,xscale,yscale)
```

The various arguments are

- _M_xticks,_M_yticks: A list for the major ticks in the x and y axes, respectively.
- _M_xticks,_M_yticks: A list for the major ticks in the x and y axes, respectively, for which no number is printed.
- _m_xticks,_m_yticks: A list for the minor ticks in the x and y axes, respectively.
- xmin, xmax (ymin,ymax): Minimum and maximum of the x (y) axes, respectively.
- xscale, yscale: The scale of the x and y axes, respectively. The available values are "linear", "log", and "symlog".

Instances of FT should be defined after subplots have been defined, because the code seems clearer. Once this is done, we can format the ticks of a subplot, sub, using

```
FT.format_ticks(plt,sub)
```

It is important to not that plt is the matplotlib.pyplot module that is usually imported as

```
import matplotlib.pyplot as plt
```

Example of FT

Consider as an example the plot of $f(x) = e^{-\frac{1}{2}x^2}$ for $-5 \leq x \leq 5$. In order to do this, we need to run

```
import numpy as np#you usually need numpy

#---these are for plots---#
import matplotlib.pyplot as plt

#load the FT module
from sys import path as sysPath
sysPath.append('../../src')
from interfacePy.FT import FT

fig=plt.figure(figsize=(9,4))
fig.subplots_adjust(bottom=0.15, left=0.15, top = 0.95, right=0.9,wspace=0.0,hspace=0.0)
sub = fig.add_subplot(1,1,1)

x=np.linspace(-5,5,500)
sub.plot(x,np.exp(-1/2. * x**2))
```

Now we can format the ticks using FT. For the shake of this example, for the x-axis, we choose as major ticks all integrs except 3 and -3, and minor ticks the halves of each major tick. Also, for the y-axis, we choose to only show major ticks for every 0.1 in the interval of interest. Finally, we set the plot limits as $x \in [-5, 5]$ ad $y \in [0, 1]$, usng linear scale for both. This can be done as

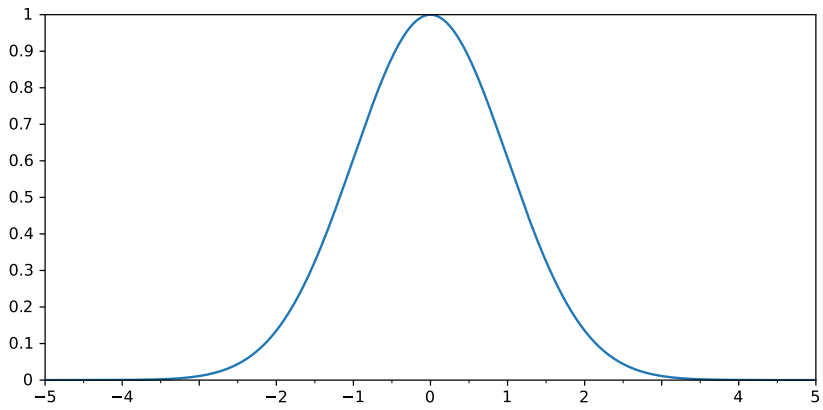


Figure 6: Example plot obtained by using the `FT` class.

```
#set major ticks
_M_xticks=[i for i in range(-5,6)]
_M_yticks=[i/10. for i in range(0,11)]

#set major ticks that will not have a label
_M_xticks_exception=[-3,3]
_M_yticks_exception=[]

_m_xticks=[i+j/2. for i in range(-5,6) for j in [0,1]]
_m_yticks=[]
ft=FT(_M_xticks, _M_yticks, _M_xticks_exception, _M_yticks_exception, _m_xticks, _m_yticks,
   xmin=-5, xmax=5, ymin=0, ymax=1, xscale='linear', yscale='linear')

ft.format_ticks(plt,sub)
```

The resulting plot is shown in Fig. (6).

F Utilities

There are various utilities, that can be found `MiMeS/src/util`, which can be used to make the use of `MiMeS` easier. In this section we will show how they work, and when they should be used.

F.1 `FormatFile.sh`

This is a `bash` script that formats a data file so that it is compatible with the interpolation assumptions. It takes a path to a file as an arguments, and it returns the same file, sorted (in ascenting order) with respect to the first column, removes duplicate and empty lines, and it removes the last "new line" (*i.e.* `\n`). The script is called as

```
bash MiMeS/src/util/FormatFile.sh path_to_dat_file
```

Here `path_to_dat_file` is a path to the data file we would like to replace with the formatted one. Notice that `FormatFile.sh` must be run in order to ensure that the cosmological input needed by the `mimes::Axion<LD,SOLVER,METHOD<LD>>` class, is acceptable.

Moreover, note that `FormatFile.sh` runs every time `configure.sh` is called, in order to format the data files for the RDOFs, $\chi(T)$, and the anharmonic factor, in order to ensure that they comply with what `MiMeS` expects.

F.2 `timeit.sh`

This script takes two arguments. The first should be a path to an executable, the second a file that contains arguments to passed to the executable. The script, then, runs the executable, and prints

in `stderr` the time it took (is seconds) to run it. It is important to note that each argument of the executable should be in a different line of the file. [18]

The script is called as

```
bash MiMeS/src/util/timeit.sh path_to_executable path_to_dat_file
```

This script is usefful for running the example code in `MiMeS/UserSpace/Cpp/Axion/Axion.cpp`, which is compiled by calling "`make examples`" in the root directory of `MiMeS`. The executable that results from the compilation of this source, `Axion.run`, expects the same arguments as the constructor of the `mimes::Axion<LD,SOLVER,METHOD<LD>>` class (see section B). This means that we can write a file (call it `INPUT`) that contains all these arguments in each line, and simply run

```
bash MiMeS/src/util/timeit.sh MiMeS/UserSpace/Cpp/Axion/Axion.run INPUT
```

The files `KinationInputs`, `MatterInputs`, and `RDInputs`; in `MiMeS/UserSpace/Cpp/Axion`, show these files should look like in different cases.

F.3 `Timer` C++ class

The class `mimes::util::Timer`, defined in `MiMeS/src/util/timeit.hpp`, can be used in order to time processes in C++. An instance of this class prints its life-time – *i.e.* time from the moment it was created until it went out of scope – in `stderr`. Instances of this class are intended to be used inside a scope in order to count te time it took to run a block of code. For example the time it takes to compute $\sum_{i=0}^{10^4} i$ can be determined as

```
#include "src/util/timeit.hpp"

int main(){
   auto sum = 0;
   {
      Timer _timer_;
      for(auto i=0; i<=10000; ++i){ ++sum;}
   }

   return 0;
}
```

F.4 `linspace` function

The function `linspace`, defined in `MiMeS/src/util/linspace.hpp`, is used to generate linearly spaced numbers between some boundary. This function is overloaded; there are two functions with the same name and different signature. The first one is

```
 /*linspace where the list is pushed in a vector that is pass by reference*/
template<class LD>
void mimes::util::linspace(LD min, LD max, unsigned int length, std::vector<LD> &X)
```

Here, `X` is cleared, and gets filled with `lenght` linearly spaced numbers `min` and `max`.

The overloaded version of this function is

```
 /*linspace where the list returned as a vector*/
template<class LD>
std::vector<LD> mimes::util::linspace(LD min, LD max, unsigned int length)
```

This function does not get a reference to a `std::vector<LD>`, instead it return one.

[18] For example , if we have an executable that takes two arguments, let's assume the first is 3 and the second is "foo", the file should read:

```
3
foo
```

F.5 `logspace` function

The function `logspace`, defined in `MiMeS/src/util/logspace.hpp`, is used to generate $\log_{10}$ spaced points between a given boundary. The signatures of this function are

```
1    /*logspace where the list is pushed in a vector that is pass by reference*/
2   template<class LD>
3   void mimes::util::logspace(LD min, LD max, unsigned int length, std::vector<LD> &X)
4
5    /*logspace where the list returned as a vector*/
6   template<class LD>
7   std::vector<LD> mimes::util::logspace(LD min, LD max, unsigned int length)
```

Both of these functions work as `linspace`.

F.6 `map` function

The function `map`, defined in `MiMeS/src/util/map.hpp`, applys a function of an `std::vector`, and fills anther `std::vector` with the result. The signatures of this function are

```
1    /*The list is pushed in a vector that is pass by reference*/
2   template<class LD>
3   void map(const std::vector<LD> &X, std::function<LD(LD)> func, std::vector<LD> &Y )
4
5    /*The list returned as a vector*/
6   template<class LD>
7   std::vector<LD> map(const std::vector<LD> &X, std::function<LD(LD)> func)
```

Both of these functions evaluate `func` (a callable object) for every element of `X`. If another `std::vector` is given, then it is cleared and filled with the results, otherwise the function returns a `std::vector` with the values of `func(X)`.

G Quick guide to the user input

We present tables (2), (3), and (4), with the various available run-time inputs, required files, and template arguments. In table 5 we show the available compile-time options, that can be used when compiling using the various `makefile` files.

User run-time input.	
`theta_i`	Initial angle.
`fa`	The PQ scale.
`umax`	Once $u = \log a/a_i >$`umax`, the integration stops. Typical value: ~ 500.
`TSTOP`	Once $T <$`TSTOP`, integration stops. Typical value: 10^{-4} GeV.
`ratio_ini`	Integration starts at u with $3H/\tilde{m}_a \approx$`ratio_ini`. Typical value: $\sim 10^3$.
`N_convergence_max` `convergence_lim`	Integration stops when the relative difference between two consecutive peaks is less than `convergence_lim` for `N_convergence_max` consecutive peaks.
`inputFile`	Relative (or absolute) path to a file that describes the cosmology. The columns should be: u T [GeV] $\log H$, with acceding u. Entropy injection should have stopped before the lowest temperature of given in `inputFile`.
`axionMass`	Instance of `mimes::AxionMass<LD>` class. In `C++` this instance is passed as a pointer to the constructor of the `mimes::Axion<LD,Solver,Method>` class, while in `python` it is simply passed as a variable.
`initial_stepsize`	Initial step-size of the solver. Default value: 10^{-2}.
`minimum_stepsize`	Lower limit of the step-size. Default value: 10^{-8}.
`maximum_stepsize`	Upper limit of the step-size. Default value: 10^{-2}.
`absolute_tolerance`	Absolute tolerance of the RK solver (see also table 1). Default value: 10^{-8}.
`relative_tolerance`	Relative tolerance of the RK solver (see also table 1). Default value: 10^{-8}.
`beta`	Aggressiveness of the adaptation strategy (see also table 1). Default value: 0.9.
`fac_max`, `fac_min`	The step-size does not change more than `fac_max` and less than `fac_min` within a trial step (see also table 1). Default values: 1.2 and 0.8, respectively.
`maximum_No_steps`	If integration needs more than `maximum_No_steps` integration stops. Default value: 10^7.

Table 2: Table of run-time user input.

Required data files, with corresponding variables in `MiMeS/Paths.mk`.	
`cosmoDat`	Relative path to data file with T (in GeV), $h_{\rm eff}$, $g_{\rm eff}$. If the path changes one must run `bash configure.sh` and `make`.
`axMDat`	Relative path to data file with T (in GeV), χ (defined from eq. (2.2)). If the path changes one must run `bash configure.sh` and `make`. This variable can be omitted, since the user can define an AxionMass instance using any path.
`anFDat`	Relative path to data file with $\theta_{\rm peak}$, $f(\theta_{\rm peak})$. If the path changes one must run `bash configure.sh` and `make`.

Table 3: Paths to the required data files.

References

[1] R. D. Peccei and H. R. Quinn, *CP Conservation in the Presence of Instantons*, *Phys. Rev. Lett.* **38** (1977) 1440–1443.

Template arguments.

`LD`	This template argument appears in all classes of `MiMeS`. The preferred choice should be either `long double`. However, in many cases `double` can be used. The user should be careful, as the later can lead to a inaccurate result; especially for low tolerances, and small values of θ.
`Solver`	This is the second template argument of the `mimes::Axion<LD,Solver,Method>` class. The available choices are `Solver`=1 for Rosenbrock method, and `Solver`=2 for explicit RK method.
`Method`	The third template argument of the `mimes::Axion<LD,Solver,Method>` class. Its value depends on the choice of `Solver`; For `Solver`=1, `Method` can be either `RODASPR2<LD>` (fourth order) or `ROS34PW2<LD>` (third order). For `Solver`=2, `Method` can only be `DormandPrince<LD>` (seventh order). Notice that the definitions of the various method classes, also need a template argument,`LD`, that must be the same as thi first template argument of the `mimes::Axion<LD,Solver,Method>` class. If one defines their own Butcher table (see Appendix A), then they would have to follow their definitions and assumptions.

Table 4: Template arguments of the various `MiMeS` classes.

[2] S. Weinberg, *A New Light Boson?*, *Phys. Rev. Lett.* **40** (1978) 223–226.

[3] F. Wilczek, *Problem of Strong P and T Invariance in the Presence of Instantons*, *Phys. Rev. Lett.* **40** (1978) 279–282.

[4] J. Preskill, M. B. Wise, and F. Wilczek, *Cosmology of the Invisible Axion*, *Phys. Lett. B* **120** (1983) 127–132.

[5] M. Dine and W. Fischler, *The Not So Harmless Axion*, *Phys. Lett. B* **120** (1983) 137–141.

[6] L. F. Abbott and P. Sikivie, *A Cosmological Bound on the Invisible Axion*, *Phys. Lett. B* **120** (1983) 133–136.

[7] **Planck** Collaboration, N. Aghanim et al., *Planck 2018 results. VI. Cosmological parameters*, *Astron. Astrophys.* **641** (2020) A6, [arXiv:1807.06209].

[8] Y. Chikashige, R. N. Mohapatra, and R. D. Peccei, *Are There Real Goldstone Bosons Associated with Broken Lepton Number?*, *Phys. Lett. B* **98** (1981) 265–268.

[9] H. M. Georgi, L. J. Hall, and M. B. Wise, *Grand Unified Models With an Automatic Peccei-Quinn Symmetry*, *Nucl. Phys. B* **192** (1981) 409–416.

[10] A. Ringwald, *Axions and Axion-Like Particles*, in *49th Rencontres de Moriond on Electroweak Interactions and Unified Theories*, pp. 223–230, 2014. arXiv:1407.0546.

[11] D. J. Marsh, *Axion cosmology*, *Physics Reports* **643** (2016) 1–79. Axion cosmology.

[12] S. Chang, C. Hagmann, and P. Sikivie, *The Cold axion populations*, in *2nd International Heidelberg Conference on Dark Matter in Astro and Particle Physics*, 7, 1998. hep-ph/9812327.

[13] S. Borsanyi et al., *Calculation of the axion mass based on high-temperature lattice quantum chromodynamics*, *Nature* **539** (2016), no. 7627 69–71, [arXiv:1606.07494].

[14] R. T. Co, L. J. Hall, and K. Harigaya, *Axion Kinetic Misalignment Mechanism*, *Phys. Rev. Lett.* **124** (2020), no. 25 251802, [arXiv:1910.14152].

[15] D. H. Lyth, *Axions and inflation: Sitting in the vacuum*, *Phys. Rev. D* **45** (1992) 3394–3404.

User compile-time options. Variables in the various `Definitions.mk` files.	
`rootDir`	The relative path of root directory of `MiMeS`. Relevant only when compiling using `make`. Available in all `Definitions.mk`.
`LONG`	`long` for `long double` or empty for `double`. This is defines a macro in the source files of the various `C++` examples. Available in `Definitions.mk` inside the various subdirectories of `MiMeS/UserSpace/Cpp`.
`LONGpy`	`long` or empty. Same as `LONG`, applies in the `python` modules. Available in `MiMeS/Definitions.mk`.
`SOLVER`	In order to use a Rosenbrock method `SOLVER`=1. For explicit RK method, `SOLVER`=2. This defines a macro that is passes as the second template argument of `mimes::Axion<LD,Solver,Method>`. The corresponding variable in `MiMeS/Definitions.mk` applies to the `python` modules. The variable in `MiMeS/UserSpace/Cpp/Axion/Definitions.mk` applies to the example in the same directory.
`METHOD`	Depending of the solver, this macro should name one of its available methods. For `SOLVER`=1, `METHOD`=RODASPR2 (fourth order) or ROS34PW2 (third order). For `SOLVER`=2, `METHOD`=DormandPrince (seventh order). There is a macro (`METHOD`) used by the shared library `MiMeS/lib/Axion_py.so`. The corresponding variable in `MiMeS/Definitions.mk` applies to the `python` modules. The variable in `MiMeS/UserSpace/Cpp/Axion/Definitions.mk` applies to the examples in the that directory.
Compiler options	
`CC`	The preferred `C++` compiler (`g++` by default). Corresponding variable in all `Definitions.mk` files.
`OPT`	Optimization level of the compiler. Available options `OPT`=`O1`, `O2`, `O3` (be default). By Corresponding variable in all `Definitions.mk` files.

Table 5: User compile-time input. These are available in the various `Definitions.mk` files, which are used when compiling using `make`.

[16] K. J. Bae, J.-H. Huh, and J. E. Kim, *Update of axion CDM energy*, *JCAP* **09** (2008) 005, [arXiv:0806.0497].

[17] P. Arias, N. Bernal, D. Karamitros, C. Maldonado, L. Roszkowski, and M. Venegas, *New opportunities for axion dark matter searches in nonstandard cosmological models*, arXiv:2107.13588.

[18] D. Karamitros, *NaBBODES: Not a black box ordinary differential equation solver in C++*, 2019–.

[19] D. Karamitros, *SimpleSplines: A header only library for linear and cubic spline interpolation in C++.*, 2021–.

[20] E. W. Kolb and M. S. Turner, *The early universe.* Frontiers in Physics. Westview Press, Boulder, CO, 1990.

[21] P. J. E. Peebles, *Principles of physical cosmology.* Princeton University Press, 1993.

[22] J. Rang, *Improved traditional rosenbrock–wanner methods for stiff odes and daes*, *Journal of Computational and Applied Mathematics* **286** (2015) 128–144.

[23] *New rosenbrock w-methods of order 3 for partial differential algebraic equations of index 1*, *BIT Numerical Mathematics* **45** (2005) 761–787.

[24] J. Dormand and P. Prince, *A family of embedded runge-kutta formulae*, *Journal of Computational and Applied Mathematics* **6** (1980), no. 1 19–26.

[25] K. Saikawa and S. Shirai, *Precise WIMP Dark Matter Abundance and Standard Model Thermodynamics*, *JCAP* **08** (2020) 011, [arXiv:2005.03544].

[26] P. Arias, N. Bernal, A. Herrera, and C. Maldonado, *Reconstructing Non-standard Cosmologies with Dark Matter*, *JCAP* **10** (2019) 047, [arXiv:1906.04183].

[27] P. Arias, D. Karamitros, and L. Roszkowski, *Frozen-in fermionic singlet dark matter in non-standard cosmology with a decaying fluid*, *JCAP* **05** (2021) 041, [arXiv:2012.07202].

[28] C. OḤARE, *cajohare/axionlimits: Axionlimits*, .

[29] E. Hairer, S. Nørsett, and G. Wanner, *Solving Ordinary Differential Equations I Nonstiff problems*. Springer, Berlin, second ed., 2000.

[30] E. Hairer and G. Wanner, *Solving Ordinary Differential Equations II: Stiff and Differential-Algebraic Problems*. Springer Series in Computational Mathematics. Springer Berlin Heidelberg, 2010.

[31] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery, *Numerical Recipes 3rd Edition: The Art of Scientific Computing*. Cambridge University Press, USA, 3 ed., 2007.

[32] **Particle Data Group** Collaboration, P. Zyla et al., *Review of Particle Physics*, *PTEP* **2020** (2020), no. 8 083C01.

[33] J. D. Hunter, *Matplotlib: A 2d graphics environment*, *Computing in Science & Engineering* **9** (2007), no. 3 90–95.