NSCpp: Non-Standard Cosmologies in C++

Karamitros Dimitrios

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

December 20, 2022

Abstract

Program summary:
Program title: NSCpp.
Developer's respository link: https://github.com/dkaramit/NSCpp.
Programming language: C++ *and* python.
Licensing provisions: MIT license.
Nature of problem:
Solution method: Embedded Runge-Kutta for the numerical integration of the system of differential equations. The user may choose between explicit and Rosenborck methods, or implement their own Butcher tableau. For the various interpolations, the library uses cubic splines.
Restrictions:

Contents

1 Introduction

2 Evolution equations

In order to model a non-standard cosmological history, we assume that the Universe at early was dominated by two components; the plasma and a fluid (Φ) with an equation of state

$$p_\Phi = (c/3 - 1)\rho_\Phi \,, \tag{2.1}$$

with c a constant.

Since the contribution to the energy budget of the Universe of such fluid is severely constraint at temperatures lower than $T \sim \mathcal{O}(10)$ MeV, Φ must lose energy to the plasma. We model this by introducing a constant energy loss rate, Γ_Φ. The equations that describe the evolution of both components is

$$\frac{ds_\mathrm{R}}{dt} = -3\,H\,s_\mathrm{R} + \frac{\Gamma_\Phi}{T}\,\rho_\Phi \tag{2.2}$$

$$\frac{d\rho_\Phi}{dt} = -c\,H\,\rho_\Phi - \Gamma_\Phi\,\rho_\Phi \,, \tag{2.3}$$

with s_R the entropy density of the plasma, and H the Hubble parameter of the Universe.

2.1 Notation

In order to solve this system of equations, we change the integration variable to

$$u = \log\frac{a}{a_\mathrm{ini}} \,, \tag{2.4}$$

with a the scale factor of the Universe, and a_ini its value at some initial time. The initial condition can now be chosen at $u_\mathrm{ini} = 0$, corresponding to $T = T_\mathrm{ini}$ and an initial value for $\rho_\Phi = \rho_{\Phi\,\mathrm{ini}}$.

We can reparametrise the temperature of hte plasma and the energy density of Φ as

$$T(u) = T_\mathrm{ini}\,f_\mathrm{R}(u)\,e^{-u} \tag{2.5}$$

$$\rho_\Phi(u) = \rho_{\Phi\,\mathrm{ini}}\,f_\Phi(u)\,e^{-cu} \,, \tag{2.6}$$

with $f_\mathrm{R}(u)$ and f_Φ functions of u that obey the equations

$$\frac{d\log f_\mathrm{R}}{du} = 1 - \left(1 - \frac{\Gamma_\Phi}{H}\,\frac{\rho_\Phi}{3HTs}\right)\delta_h \tag{2.7}$$

$$\frac{d\log f_\Phi}{du} = -\frac{\Gamma_\Phi}{H} \,, \tag{2.8}$$

with

$$\delta_h = 1 + \frac{1}{3}\frac{d\log h_\mathrm{eff}}{d\log T} \,. \tag{2.9}$$

Notice that for $\Gamma_\Phi = 0$ and $\delta_h = 1$, $f_\mathrm{R} = f_\Phi = 1$. These are the equations that `NSCpp` actually solves.

3 `NSCpp` usage

The latest stable version of `NSCpp` is available at `github.com/dkaramit/NSCpp/tree/stable`, which can also be obtained by running: [1]

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

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

It is important to note that NSCpp relies on NaBBODES [1] and SimpleSplines [2]. These are two libraries developed independently, and in order to get NSCpp with the latest version of these libraries, one needs to run following commands

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

This downloads the master branch of NSCpp; and NaBBODES [1] and SimpleSplines [2] as submodules. This guaranties that NSCpp uses the most updated version of these libraries, although it may not be stable.

Once everything is downloaded successfully, we can go inside the NSCpp 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 4.3 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).

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

3.1 First steps

```
#include"src/NSC/NSCSolve.hpp"
```

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

Setting up the solver

```
nsc::NSC<LD,SOLVER,METHOD<LD>> BE(TEND,c,Ti,ratio,umax,TSTOP,
  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 table3.

The various parameters are as follows:

1.
2.
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.
5. initial_stepsize (optional): Initial step the solver takes.
6. maximum_stepsize (optional): This limits the step-size to an upper limit.
7. minimum_stepsize (optional): This limits the step-size to a lower limit.
8. absolute_tolerance (optional): Absolute tolerance of the RK solver
9. relative_tolerance (optional): Relative tolerance of the RK solver.

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

11. `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.

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

```
BE.solveNSC();
```

3.1.1 Using nsc in python

The modules for the `python` interface are located in `NSCpp/src/interfacePy`. Although the usage of the 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 `NSCpp`/Definitions.mk before running "`make`". The various options are discussed in section 4.2, and outlined in table 4.

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.NSC import NSC
```

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

The solver We can define an `NSC` instance as follows

```
BE=NSC(TEND,c,Ti,ratio,umax,TSTOP,
     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, and outlined in table 1. Moreover, the usage of the class can be found by running `?NSC` after loading the module.

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

```
BE.solveNSC()
```

in order to solve the BEs. In contrast to the `C++` implementation, this only gives us access to the points where the behaviour changes; the corresponding variables are `BE.TE1,BE.TE2,BE.TD1,BE.TD2`. In order to get the evolution of the densities, we need to run

```
BE.getPoints()
```

This will make `numpy` [**?**] arrays that contain the ˙·. Moreover, we can run the following

```
BE.getErrors()
```

4 Assumptions and user input

4.1 Restrictions

4.2 Options at Compile-time

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

1. rootDir: The relative path of root directory of NSCpp.

2. LONG: This sets the numeric types for the C++ examples. It 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 compile the shared libraries is via running "make" in the root directory of NSCpp, this variable needs to be defined inside NSCpp/Definitions.mk. By default, this variable is set to long, since this is the most stable choice in general.

4. SOLVER: NSCpp uses the ordinary differential equation (ODE) integrators of ref. [1]. 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 Apendix of ref.[3]). By default inside the Definitions.mk files 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 NSCpp/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 NSCpp is used.

5. METHOD: Depending on the type of solver, there are some available methods. [2]

    - 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. [**?**]). The ROS34PW2 choice corresponds to a third order Rosenbrock-Wanner method [**?**].

[2] 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 **??**.

- For SOLVER=2, the only reliable method available in NaBBODES is the Dormand-Prince [?] 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++. NSCpp 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 NSCpp 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 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 NSCpp.

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. [3] For example, the choice LONG=long, SOLVER=1, and METHOD=RODASPR2 will be used to compile the shared libraries (and C++ example in NSCpp/UserSpace/Cpp/NSC) with nsc::NSC<long double,1,RODASPR2<long double>>.

4.3 User input

4.3.1 Compile-time input

Files NSCpp requires files that provide data for the relativistic degrees of freedom (RDOF) of the plasma. Although NSCpp is shipped with the standard model RDOF found in [?], the user can use their prefered tabulated ones, by providing a file and writing its path to the cosmoDat file. The format of the file has to be: T (in GeV), $h_{\rm eff}$, and $g_{\rm eff}$.

The path to this file should be given at compile time. That is, once Paths.mk changes, we must run "bash configure.sh" and then "make" in order to make sure that they will be used. The user can change the content of the data files (without changing their paths), in order to use them without compiling NSCpp 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 ?? 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 NSCpp.

This path is stored as (constexpr) string in NSCpp/src/misc_dir/path.hpp at compile-time, and can be accessed once this header file is included. The corresponding variable is cosmo_PATH.

5 Acknowledgements

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

[3] 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.

6 Summary

Appendix

A C++ classes

B NSCpp python interface

C Quick guide to the user input

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

	User run-time input for solving the BEs.
`TSTOP`	Once $T <$`TSTOP`, integration stops. Typical value: 10^{-4} GeV.
`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. Default value: 10^{-8}.
`relative_tolerance`	Relative tolerance of the RK solver. Default value: 10^{-8}.
`beta`	Aggressiveness of the adaptation strategy. 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 . 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 1: Table of the constructor arguments of the `nsc::NSC<LD,Solver,Method>` class.

Required data files, with corresponding variables in `NSCpp/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`.

Table 2: Paths to the required data files. Variables defined in the `NSCpp/Paths.mk` files, and used when running `bash configure.sh` in the root directory of `NSCpp`.

Template arguments.	
`LD`	This template argument appears in all classes of `NSCpp`. The preferred choice is `long double`. However, in many cases `double` can be used. The user should be careful, as the later can lead to an inaccurate result; especially for low tolerances, and small values of θ.
`Solver`	This is the second template argument of the `nsc::NSC<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 `nsc::NSC<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 the first template argument of the `nsc::NSC<LD,Solver,Method>` class. If one defines their own Butcher table, then they would have to follow their definitions and assumptions.

Table 3: Template arguments of the various `NSCpp` classes.

References

[1] D. Karamitros, *NaBBODES: Not a Black Box Ordinary Differential Equation Solver in C++*, 2019.

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

[3] D. Karamitros, *MiMeS: Misalignment mechanism solver*, *Comput. Phys. Commun.* **275** (2022) 108311, [arXiv:2110.12253].

User compile-time options. Variables in the various `Definitions.mk` files.	
`rootDir`	The relative path of root directory of `NSCpp`. 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 `NSCpp/UserSpace/Cpp`.
`LONGpy`	`long` or empty. Same as `LONG`, applies in the `python` modules. Available in `NSCpp/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 `NSCpp/Definitions.mk` applies to the `python` modules. The variable in `NSCpp/UserSpace/Cpp/NSC/Definitions.mk` applies to the example in the same directory.
`METHOD`	Depending on the solver, this variable 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 `NSCpp/lib/libNSC.so`. The corresponding variable in `NSCpp/Definitions.mk` applies to the `python` modules. The variable in `NSCpp/UserSpace/Cpp/NSC/Definitions.mk` applies to the example in that directory.
Compiler options	
`CC`	The preferred `C++` compiler (`g++` by default). Corresponding variable in all `Definitions.mk` files.
`OPT`	Available options are `OPT`=`O1`, `O2`, `O3` (be default). This variable defines the optimization level of the compiler. The variable can be changed in all `Definitions.mk` files. In the root directory of `NSCpp`, the optimization level applies to the python modules (*i.e.* the shared libraries), while in the subdirectories of `NSCpp/UserSpace/Cpp` it only applies to example inside them.

Table 4: User compile-time input and options. These are available in the various `Definitions.mk` files, which are used when compiling using `make`.