


Article

Exploiting the Abstract Calculus Pattern for the Integration of Ordinary Differential Equations for Dynamics Systems: An Object-Oriented Programming Approach in Modern Fortran

Stefano Zaghi ^{1,*}  and Cristiano Andolfi ^{2,†}¹ CNR-IAC, Istituto per le Applicazioni del Calcolo “Mauro Picone”, Via dei Taurini 19, 00185 Rome, Italy² Department of Engineering, Università Niccolò Cusano, Via Don Carlo Gnocchi 3, 00166 Rome, Italy; cristiano.andolfi@unicusano.it

* Correspondence: stefano.zaghi@cnr.it

† Ph.D. Aerospace Engineer, Research Scientist at CNR-IAC.

‡ Ph.D. Student at Università Niccolò Cusano, Aeronautical Engineer.

Abstract: This manuscript relates to the exploiting of the abstract calculus pattern (ACP) for the (numerical) solution of ordinary differential equation (ODEs) systems, which are ubiquitous mathematical formulations of many physical (dynamical) phenomena. We present FOODIE, a software suite aimed to numerically solve ODE problems by means of a clear, concise, and efficient abstract interface. The results presented prove manifold findings, in particular that our ACP approach enables ease of code development, clearness and robustness, maximization of code re-usability, and conciseness comparable with computer algebra system (CAS) programming (interpreted) but with the computational performance of compiled programming. The proposed programming model is also proven to be agnostic with respect to the parallel paradigm of the computational architecture: the results show that FOODIE applications have good speedup with both shared (OpenMP) and distributed (MPI, CAF) memory architectures. The present paper is the first announcement of the FOODIE project: the current implementation is extensively discussed, and its capabilities are proved by means of tests and examples.

Keywords: ordinary differential equations (ODE); partial differential equations (PDE); object-oriented programming (OOP); abstract calculus pattern (ACP); Fortran



Citation: Zaghi, S.; Andolfi, C. Exploiting the Abstract Calculus Pattern for the Integration of Ordinary Differential Equations for Dynamics Systems: an Object-Oriented Programming Approach in Modern Fortran. *Dynamics* **2023**, *3*, 488–529. <https://doi.org/10.3390/dynamics3030026>

Academic Editor: Ravi P. Agarwal and Maria Alessandra Ragusa

Received: 23 June 2023

Revised: 27 July 2023

Accepted: 29 July 2023

Published: 28 August 2023



Copyright: © 2023 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (<https://creativecommons.org/licenses/by/4.0/>).

1. Introduction

1.1. Background

The initial value problem (IVP, or Cauchy problem, see [1]) constitutes a class of mathematical models of paramount relevance, being applied to the modeling of a wide range of dynamic phenomena. Briefly, an IVP is an ordinary differential equation (ODE) system coupled with specified initial values of the unknown state variables, the solution of which are searched for at a given time after the initial time considered.

The prototype of IVP can be expressed as

$$\begin{aligned} U_t &= R(t, U) \\ U_0 &= U(t_0) \end{aligned} \quad (1)$$

where $U(t)$ is the vector of state variables being a function of the time-like independent variable t , $U_t = \frac{dU}{dt} = R(t, U)$ is the (vectorial) residuals function, and $U(t_0)$ is the (vectorial) initial conditions, namely the state variables function evaluated at the initial time t_0 . In general, the residuals function R is a function of the state variable U through which it is a function of time, but it can also be a direct function of time, and thus, in general, $R = R(t, U(t))$ holds.

The problem prototype (1) is ubiquitous in the mathematical modeling of physical problems: essentially, whenever an *evolutionary* (i.e., dynamic) phenomenon is considered, the prevision (simulation) of the future solutions involves the solution of an IVP. As a matter of fact, many physical problems (fluid dynamics, chemistry, biology, evolutionary-anthropology, among others) are described by means of an IVP.

It is worth noting that the state vector variables U and its corresponding residuals function $U_t = \frac{dU}{dt} = R(t, U)$ are *problem dependent*: the number and the meaning of the state variables as well as the equations governing their evolution (which are embedded into the residuals function) are different for the Navier–Stokes conservation laws with respect to the Burgers one, as an example. Nevertheless, the solution of the IVP model prototype can be generalized, allowing the application of the same solver to many different problems, thus eliminating the necessity of re-implementing the same solver for each different problem.

In this work, we present the FOODIE library: it is designed for solving the generalized IVP (1), being completely unaware of the actual problem's definition. The FOODIE library provides a high-level, well-documented, simple application program interface (API) for many well-known ODE integration solvers, with its aims being twofold:

- Provide a robust set of ODE solvers ready to be applied to a wide range of different problems;
- Provide a simple framework for the rapid development of new ODE solvers.

1.2. Related Works

There are many ODE solvers described in the literature. In [2], a SODES (stepwise ordinary differential equations solver) is presented: the authors describe an ODE solver able to provide a step-by-step ODE solution exploiting a computer Algebra system (CAS) written in the Python programming language. In the framework of computational fluid dynamics (CFD), and in particular for solving detailed chemical kinetics problems, in [3], a novel neural ODE solver, ChemNODE, is presented: exploiting the neural networks, the chemical source terms are predicted and integrated and the networks themselves are adjusted during the training to minimize errors. In [4], the problem of ODE solving is considered with respect to the computational efficiency point of view: the authors analyze the performance of three different solvers written in the C++ and Julia programming languages on both CPU and GPU architectures, with a special focus on the parallel optimization of the ODE solving algorithms. In [5], the Python framework TensorFlow is exploited to implement a neural-network-based ODE solver: their approach, is hybrid in the sense that the neural model combines both physics-informed and data-driven kernels in order to improve the accuracy of the ODE solutions. The MAPLE CAS software (a symbolic and numeric computing environment) has been used in [6] to apply a novel iterative scheme based on the Mohand homotopy perturbation transform (MHPT) to the simulation of nonlinear shock wave equations, proving a good computational efficiency.

The ODE framework solver presented in this work has different aims, as explained in the following subsection.

1.3. Motivations and Aims

The FOODIE library is a free software application (<https://github.com/Fortran-FOSS-Programmers/FOODIE>, accessed 20 August 2023) and is designed by the authors of the current paper with the following specifications:

- It is written in modern Fortran (standard 2008 or newer);
- It is written by means of the object-oriented programming (OOP) paradigm;
- It is well documented;
- It is test-driven developed (TDD);
- It is collaboratively developed;
- It is free.

FOODIE, meaning Fortran Object oriented Ordinary Differential Equations integration library, has been developed with the aim to satisfy the above specifications. The present paper is its first comprehensive presentation.

The Fortran (Formula Translator, [7,8]) programming language is the de facto standard into computer science field: it strongly facilitates the effective and efficient translation of (even complex) mathematical and numerical models into an operative software without compromise on computations speed and accuracy. Moreover, its simple syntax is suitable for scientific researchers that are interested (and skilled) in the physical aspects of the numerical computations rather than computer technicians. Consequently, we develop FOODIE using Fortran language: FOODIE is written by research scientists for research scientists.

One key-point of the FOODIE development is the *problem generalization*: the problem solved must be the IVP (1) rather than any of its actual definitions. Consequently, we must rely on a *generic* implementation of the solvers. To this aim, OOP is very useful (see [9]): it allows to express IVP (1) in a very concise and clear formulation that is really generic. In particular, our implementation is based on *Abstract Calculus Pattern* (ACP) concept.

1.3.1. The Abstract Calculus Pattern

The abstract calculus pattern provides a simple solution for the connection between the very high-level expression of IVP (1) and the eventual concrete (low-level) implementation of the ODE problem being solved. ACP essentially constitutes a *contract* based on an Abstract Data Type (ADT): we specify an ADT supporting a certain set of mathematical operators (differential and integral ones) and implement FOODIE solvers only on the basis of this ADT. FOODIE clients must formulate the ODE problem under integration defining their own concrete extensions of our ADT (implementing all the deferred operators). Such an approach defines the abstract calculus pattern: FOODIE solvers are aware of only the ADT, while FOODIE clients extend the ADT for defining the concrete ODE problem.

Is worth noting that this ACP emancipates the solvers implementations from any low-level problem-dependent details: the ODE solvers developed with this pattern are extremely concise, clear, maintainable and less errors-prone with respect a low-level (non abstract) pattern. Moreover, the FOODIE clients can use solvers being extremely robust: as a matter of facts, FOODIE solvers are expressed in a formulation very close to the mathematical one and are tested on an extremely varying family of problems. As shown in the following, such a great flexibility does not compromise the computational efficiency.

1.3.2. FOODIE Novelty

The main novelty of our approach is to combine many advantages of CAS programming with the computational performances of compiled (parallel) programming: as a matter of fact, CAS approaches generally enable fast and easy numerical methods implementation, but at the cost of low computational performances they being generally based on interpreted programming languages. On the other hand, compiled programming languages have (extremely) higher computational performances (especially on parallel architectures), but are more constrained with respect interpreted languages: the resulting programming approach requires, in general, more effort to implement complicated algorithms and more errors-prone. The novelty of the ACP implemented in FOODIE consists in enabling code development easiness, clearness and robustness, maximization of code re-usability and conciseness while retaining the computational performances of Fortran compiled programming language.

1.3.3. Manuscript Organization

The present paper is organized as following: in Section 2 a brief description of the mathematical and numerical methods currently implemented into FOODIE is presented; in Section 3 a detailed discussion on the implementation specifications is provided by means of an analytical code-listings review; in Section 4 a verification analysis on the results of FOODIE applications is presented; Section 5 provides an analysis of FOODIE

performances under parallel frameworks scenario like the OpenMP and MPI paradigms; finally, in Section 6 concluding remarks and perspectives are depicted.

2. Mathematical and Numerical Models

In many (most) circumstances, the solution of Equation (1) cannot be computed in a closed, exact form (even if it exists and is unique) due to the complexity and nature of the residuals functions, that is often non linear. Consequently, the problem is often solved relying on a numerical approach: the solution of system (1) at a time t^n , namely $U(t^n)$, is approximated by a subsequent time-marching approximations $U_0 = u_0 \rightarrow u_1 \rightarrow u_2 \rightarrow \dots \rightarrow u_N \approx U(t^n)$ where the relation $u_i \rightarrow u_{i+1}$ implies a *stepping, numerical integration* from the time t^i to time t^{i+1} and N is the total number of numerical time steps necessary to evolve the initial conditions toward the searched solution $U(t^n)$. To this aim, many numerical schemes have been devised. Notably, the numerical schemes of practical usefulness must possess some necessary proprieties such as *consistency* and *stability* to ensure that the numerical approximation *converges* to the exact solution as the numerical time step tends to zero. A detailed discussion of these details is out the scope of the present work and is omitted. Here, we briefly recall some classifications necessary to introduce the schemes implemented into the FOODIE library.

A non comprehensive classification of the most widely used schemes could distinguish between *multi-step* versus *one-step* schemes and between *explicit* versus *implicit* schemes.

Essentially, the multi-step schemes have been developed to obtain an accurate approximation of the subsequent numerical steps using the informations contained into the previously computed steps, thus this approach relates the next step approximation to a set of the previously computed steps. On the contrary, a one-step scheme evolves the solution toward the next step using only the information coming from the current time approximation. In the framework of one-step schemes family an equivalent accurate approximation can be obtained by means of a multi-stage approach as the one due to Runge-Kutta. The current version of FOODIE provides schemes belonging to both these families.

The other ODE solvers classification concerns with explicit or implicit nature of the schemes employed. Briefly, an explicit scheme computes the next step approximation using the previously computed steps at most to the current time, whereas an implicit scheme uses also the next step approximation (that is the unknown), thus it requires extra computations. The implicit approach is of practical use for *stiff* systems where the usage of explicit schemes could require an extremely small time step to evolve in a *stable* way the solution. Mixing together explicit and implicit schemes it is possible to build a family of *predictor-corrector* methods: using an explicit scheme to predict a guess for the next step approximation it is possible to use an implicit method for correcting this guess.

FOODIE currently implements the following ODE schemes:

- one-step schemes:
 - explicit forward Euler scheme, it being 1st order accurate;
 - explicit Runge-Kutta schemes (see [10,11]):
 - * TVD/SSP Runge-Kutta schemes:
 - 2-stages, it being 2nd order accurate;
 - 3-stages, it being 3rd order accurate;
 - 5-stages, it being 4th order accurate;
 - * low storage Runge-Kutta schemes:
 - 5-stages 2N registers schemes, it being 4th order accurate;
 - 6-stages 2N registers schemes, it being 4th order accurate;
 - 7-stages 2N registers schemes, it being 4th order accurate;
 - 12-stages 2N registers schemes, it being 4th order accurate;
 - 13-stages 2N registers schemes, it being 4th order accurate;
 - 14-stages 2N registers schemes, it being 4th order accurate;
- multi-step schemes (see [12]):

- explicit Adams-Bashforth schemes:
 - * 2-steps, it being 2nd order accurate;
 - * 3-steps, it being 3rd order accurate;
 - * 4-steps, it being 4th order accurate;
- implicit Adams-Moulton schemes:
 - * 1-step, it being 2nd order accurate;
 - * 2-steps, it being 3rd order accurate;
 - * 3-steps, it being 4th order accurate;
- predictor-corrector Adams-Bashforth-Moulton schemes:
 - * 1-step, it being 2nd order accurate;
 - * 2-steps, it being 3rd order accurate;
 - * 3-steps, it being 4th order accurate;
- explicit Leapfrog schemes:
 - * 2-steps unfiltered, it being 2nd order accurate, but mostly *unstable*;
 - * 2-steps Robert-Asselin filtered, it being 1st order accurate (on *amplitude error*);
 - * 2-steps Robert-Asselin-Williams filtered, it being 3rd order accurate (on *amplitude error*);

2.1. The Explicit forward Euler Scheme

The explicit forward Euler scheme for ODE integration is probably the simplest solver ever devised. Considering the system (1), the solution (approximation) of the state vector U at the time $t^{n+1} = t^n + \Delta t$ (Δt being the time step considered) assuming to know the solution at time t^n is:

$$U(t^{n+1}) = U(t^n) + \Delta t \cdot R[t^n, U(t^n)] \quad (2)$$

where the solution at the new time step is computed by means of only the current time solution, thus this is an explicit scheme. The solution is an approximation of 1st order, the local truncation error being $O(\Delta t^2)$. As well known, this scheme has an absolute (linear) stability locus equals to $|1 + \Delta t \lambda| \leq 1$ where λ contains the eigenvalues of the linear (or linearized) Jacobian matrix of the system.

This scheme is Total Variation Diminishing (TVD) in the stability region under the CFL limit, thus satisfies the maximum principle (or the equivalent positivity preserving property, see [13]).

2.2. The Explicit TVD/SSP Runge-Kutta Class of Schemes

Runge-Kutta methods belong to the more general multi-stage family of schemes. This kind of schemes has been designed to achieve a more accurate solution than the 1st Euler scheme, but without increasing the number of time steps used, as it is done with the multi-step schemes, see [11]. Essentially, the high order of accuracy is obtained by means of *intermediate values* (the stages) of the solution and its derivative are generated and used within a single time step. This commonly implies the allocation of some auxiliary memory registers for storing the intermediate stages.

Notably, the multi-stage schemes class has the attractive property to be *self-starting*: the high order accurate solution can be obtained directly from the previous one, without the necessity to compute *before* a certain number of previous steps, as it happens for the multi-step schemes. Moreover, one-step multi-stage methods are suitable for adaptively-varying time-step size (that is also possible for multi-step schemes, but at a cost of more complexity) and for discontinuous solutions, namely discontinued solutions happening at a certain time t^* (that in a multi-step framework can involve an overall accuracy degradation).

In general, the TVD/SSP Runge-Kutta schemes provided by FOODIE library are written by means of the following algorithm:

$$U^{n+1} = U^n + \Delta t \cdot \sum_{s=1}^{N_s} \beta_s K_s \quad (3)$$

where N_s is the number of Runge-Kutta stages used and K_s is the s th stage defined as:

$$K_s = R \left(t^n + \gamma_s \Delta t, U^n + \Delta t \sum_{l=1}^{s-1} \alpha_{s,l} K_l \right) \quad (4)$$

It is worth noting that the Equations (3) and (4) can be easily adapted for implicit schemes. A scheme belonging to this family is operative once the coefficients α , β , γ are provided. We represent these coefficients using the Butcher's table, that for an explicit scheme where $\gamma_1 = \alpha_{1,*} = \alpha_{i,i} = 0$ has the form reported in Table 1.

Table 1. Butcher's table for explicit Runge-Kutta schemes.

γ_2	$\alpha_{2,1}$				
γ_3	$\alpha_{3,1}$	$\alpha_{3,2}$			
\vdots	\vdots		\ddots		
γ_{N_s}	$\alpha_{N_s,1}$	$\alpha_{N_s,2}$	\cdots	α_{N_s,N_s-1}	
	β_1	β_2	\cdots	β_{N_s-1}	β_{N_s}

The Equations (3) and (4) show that Runge-Kutta methods do not require any additional differentiations of the ODE system for achieving high order accuracy, rather they require additional evaluations of the residuals function R .

The nature of the scheme and the properties of the solutions obtained depend on the number of stages and on the value of the coefficients selected. Currently, FOODIE provides 3 Runge-Kutta schemes having TVD or Strong Stability Preserving (SSP) propriety (thus they being suitable for ODE systems involving rapidly changing non linear dynamics) the Butcher's coefficients of which are reported in Tables 2–4.

Table 2. Butcher's table of 2 stages, 2nd order, Runge-Kutta TVD scheme (also known as trapezoidal method).

1	1	0
	1/2	1/2

Table 3. Butcher's table of 3 stages, 3rd order, Runge-Kutta SSP scheme.

1	1	
1/2	1/4	1/4
	1/6	1/6
		2/3

Table 4. Butcher's table of 5 stages, 4th order accurate, Runge-Kutta SSP scheme.

0.39175222700392	0.39175222700392				
0.58607968896779	0.21766909633821	0.36841059262959			
0.47454236302687	0.08269208670950	0.13995850206999	0.25189177424738		
0.93501063100924	0.06796628370320	0.11503469844438	0.20703489864929	0.54497475021237	
	0.14681187618661	0.24848290924556	0.10425883036650	0.27443890091960	0.22600748319395

The absolute stability locus depends on the coefficients selected, however, as a general principle, we can assume that greater is the stages number and wider is the stability locus on equal accuracy orders.

It is worth noting that FODDiE also provides a one-stage TVD Runge-Kutta solver that reverts back to the explicit forward Euler scheme: it can be used, for example, into a Recursive Order Reduction (ROR) framework that automatically checks some properties of the solution and, in case, reduces the order of the Runge-Kutta solver until those properties are obtained.

2.3. The Explicit Low Storage Runge-Kutta Class of Schemes

As aforementioned, standard Runge-Kutta schemes have the drawback to require N_s auxiliary memory registers to store the necessary stages data. In order to make an efficient use of the available limited computer memory, the class of low storage Runge-Kutta scheme was devised. Essentially, the standard Runge-Kutta class (under some specific conditions) can be reformulated allowing a more efficient memory management. Currently FOODIE provides a class of $2N$ registers storage Runge-Kutta schemes, meaning that the storage of all stages requires only 2 registers of memory with a *word* length N (namely the length of the state vector) in contrast to the standard formulation where N_s registers of the same length N are required. This is a dramatic improvement of memory efficiency especially for schemes using a high number of stages ($N_s \geq 4$) where the memory necessary is an half with respect the original formulation. Unfortunately, not all standard Runge-Kutta schemes can be reformulated as a low storage one.

Following the Williamson's approach (see [14–17]) the standard coefficients are reformulated to the coefficients vectors A , B and C and the Runge-Kutta algorithm becomes:

$$\left. \begin{aligned} K_1 &= U(t^n) \\ K_2 &= 0 \\ K_2 &= A_s K_2 + \Delta t \cdot R(t^n + C_s \Delta t, K_1) \\ K_1 &= K_1 + B_s K_2 \\ U(t^{n+1}) &= K_1 \end{aligned} \right\} s = 1, 2, \dots, N_s \quad (5)$$

Currently FOODIE provides 5/6/7/12/13/14 stages, all 4th order, $2N$ registers explicit schemes, the coefficients of which are listed in Table 5.

Similarly to the TVD/SSP Runge-Kutta class, the low storage class also provides a fail-safe one-stage solver reverting back to the explicit forward Euler solver, that is useful for ROR-like frameworks.

2.4. The Explicit Adams-Bashforth Class of Schemes

Adams-Bashforth methods belong to the more general (linear) explicit multi-step family of schemes. This kind of schemes has been designed to achieve a more accurate solution than the 1st Euler scheme using the information coming from the solutions already computed at previous time steps.

In general, the Adams-Bashforth schemes provided by FOODIE library are written by means of the following algorithm (for only explicit schemes):

$$U(t^{N_s}) = U(t^{N_s-1}) + \Delta t \sum_{s=1}^{N_s} b_s \cdot R[t^{s-1}, U(t^{s-1})] \quad (6)$$

where N_s is the number of time steps considered and b_s are the linear coefficients selected.

Table 5. Williamson’s table of low storage Runge-Kutta schemes.

(a) 5 Stages, 4th Order				(b) 6 Stages, 4th Order			
Stage	A	B	C	Stage	A	B	C
1	0	1432997174477 9575080441755	0	1	0	0.122000000000	0
2	− 567301805773 1357537059087	5161836677717 13612068292357	1432997174477 9575080441755	2	−0.691750960670	0.477263056358	0.122000000000
3	− 2404267990393 2016746695238	1720146321549 2090206949498	2526269341429 6820363962896	3	−1.727127405211	0.381941220320	0.269115878630
4	− 3550918686646 2091501179385	3134564353537 4481467310338	2006345519317 3224310063776	4	−0.694890150986	0.447757195744	0.447717183551
5	− 1275806237668 842570457699	2277821191437 14882151754819	2802321613138 2924317926251	5	−1.039942756197	0.498614246822	0.749979795490
				6	−1.531977447611	0.186648570846	0.898555413085
(c) 7 Stages, 4th Order				(d) 12 Stages, 4th Order			
Stage	A	B	C	Stage	A	B	C
1	0.000000000000	0.117322146869	0.000000000000	1	0.0000000000000000	0.0650008435125904	0.0000000000000000
2	−0.647900745934	0.503270262127	0.117322146869	2	−0.0923311242368072	0.0161459902249842	0.0650008435125904
3	−2.704760863204	0.233663281658	0.294523230758	3	−0.9441056581158819	0.5758627178358159	0.0796560563081853
4	−0.460080550118	0.283419634625	0.305658622131	4	−4.3271273247576394	0.1649758848361671	0.1620416710085376
5	−0.500581787785	0.540367414023	0.582864148403	5	−2.1557771329026072	0.3934619494248182	0.2248877362907778
6	−1.906532255913	0.371499414620	0.858664273599	6	−0.9770727190189062	0.0443509641602719	0.2952293985641261
7	−1.450000000000	0.136670099385	0.868664273599	7	−0.7581835342571139	0.2074504268408778	0.3318332506149405
				8	−1.7977525470825499	0.6914247433015102	0.4094724050198658
				9	−2.6915667972700770	0.3766646883450449	0.6356954475753369
				10	−4.6466798960268143	0.0757190350155483	0.6806551557645497
				11	−0.1539613783825189	0.2027862031054088	0.7143773712418350
				12	−0.5943293901830616	0.2167029365631842	0.9032588871651854
(e) 13 Stages, 4th Order				(f) 14 Stages, 4th Order			
Stage	A	B	C	Stage	A	B	C
1	0.0000000000000000	0.0271990297818803	0.0000000000000000	1	0.0000000000000000	0.0367762454319673	0.0000000000000000
2	−0.6160178650170565	0.1772488819905108	0.0271990297818803	2	−0.7188012108672410	0.3136296607553959	0.0367762454319673
3	−0.4449487060774118	0.0378528418949694	0.0952594339119365	3	−0.7785331173421570	0.1531848691869027	0.1249685262725025
4	−1.0952033345276178	0.6086431830142991	0.1266450286591127	4	−0.0053282796654044	0.0030097086818182	0.2446177702277698
5	−1.2256030785959187	0.2154313974316100	0.1825883045699772	5	−0.8552979934029281	0.3326293790646110	0.2476149531070420
6	−0.2740182222332805	0.2066152563885843	0.3737511439063931	6	−3.9564138245774565	0.2440251405350864	0.2969311120382472
7	−0.0411952089052647	0.0415864076069797	0.5301279418422206	7	−1.5780575380587385	0.3718879239592277	0.3978149645802642
8	−0.1797084899153560	0.0219891884310925	0.5704177433952291	8	−2.0837094552574054	0.6204126221582444	0.5270854589440328
9	−1.1771530652064288	0.9893081222650993	0.5885784947099155	9	−0.7483334182761610	0.1524043173028741	0.6981269994175695
10	−0.4078831463120878	0.0063199019859826	0.6160769826246714	10	−0.7032861106563359	0.0760894927419266	0.8190890835352128
11	−0.8295636426191777	0.3749640721105318	0.6223252334314046	11	0.0013917096117681	0.0077604214040978	0.8527059887098624
12	−4.7895970584252288	1.6080235151003195	0.6897593128753419	12	−0.0932075369637460	0.0024647284755382	0.8604711817462826
13	−0.6606671432964504	0.0961209123818189	0.9126827615920843	13	−0.9514200470875948	0.0780348340049386	0.8627060376969976
				14	−7.1151571693922548	5.5059777270269628	0.8734213127600976

Currently FOODIE provides 2, 3, and 4 steps schemes having 2nd, 3rd and 4th formal order of accuracy, respectively. The b_s coefficients are reported in Table 6.

Table 6. Explicit Adams-Bashforth coefficients.

N_s	b_1	b_2	b_3	b_4
2	$-\frac{1}{2}$	$\frac{3}{2}$	/	/
3	$\frac{5}{12}$	$-\frac{16}{12}$	$\frac{23}{12}$	/
4	$-\frac{9}{24}$	$\frac{34}{24}$	$-\frac{59}{24}$	$\frac{55}{24}$

Similarly to the Runge-Kutta classes, the Adams-Bashforth class also provides a fail-safe one-step solver reverting back to the explicit forward Euler solver, that is useful for ROR-like frameworks.

It is worth noting that for $N_s > 1$ the Adams-Bashforth class of solvers is not *self-starting*: the values of $U(t^1), U(t^2), \dots, U(t^{N_s-1})$ must be provided. To this aim, a lower order multi-step scheme or an equivalent order one-step multi-stage scheme can be used.

2.5. The Implicit Adams-Moulton Class of Schemes

Adams-Moulton methods belong to the more general (linear) implicit multi-step family of schemes. This kind of schemes has been designed to achieve a more accurate solution than the 1st Euler scheme using the information coming from the solutions already computed at previous time steps.

In general, the Adams-Moulton schemes provided by FOODIE library are written by means of the following algorithm (for only implicit schemes):

$$U(t^{N_s}) = U(t^{N_s-1}) + \Delta t \sum_{s=0}^{N_s-1} b_s \cdot R[t^s, U(t^s)] + b_{N_s} \cdot R[t^{N_s}, U(t^{N_s})] \quad (7)$$

where N_s is the number of time steps considered and b_s are the linear coefficients selected.

Currently FOODIE provides 1, 2, and 3 steps schemes having 2nd, 3rd and 4th formal order of accuracy, respectively. The b_s coefficients are reported in Table 7.

Table 7. Implicit Adams-Moulton coefficients.

N_s	b_0	b_1	b_2	b_3
1	$\frac{1}{2}$	$\frac{1}{2}$	/	/
2	$-\frac{1}{12}$	$\frac{8}{12}$	$\frac{5}{12}$	/
3	$\frac{1}{24}$	$-\frac{5}{24}$	$\frac{19}{24}$	$\frac{9}{24}$

Similarly to the Runge-Kutta and Adams-Bashforth classes, the Adams-Moulton class also provides a fail-safe zero-step solver reverting back to the implicit backward Euler solver, that is useful for ROR-like frameworks.

It is worth noting that for $N_s > 1$ the Adams-Moulton class of solvers is not *self-starting*: the values of $U(t^1), U(t^2), \dots, U(t^{N_s-1})$ must be provided. To this aim, a lower order multi-step scheme or an equivalent order one-step multi-stage scheme can be used.

2.6. The Predictor-Corrector Adams-Bashforth-Moulton Class of Schemes

Adams-Bashforth-Moulton methods belong to the more general (linear) predictor-corrector multi-step family of schemes. This kind of schemes has been designed to achieve a more accurate solution than the 1st Euler scheme using the information coming from the solutions already computed at previous time steps.

In general, the Adams-Bashforth-Moulton schemes provided by FOODIE library are written by means of the following algorithm:

$$\begin{aligned} U(t_s^{N_s^p})_p &= U(t_s^{N_s^p-1}) + \Delta t \sum_{s=1}^{N_s^p} b_s^p \cdot R[t^{s-1}, U(t^{s-1})] \\ U(t_s^{N_s^c})_c &= U(t_s^{N_s^c-1}) + \Delta t \sum_{s=0}^{N_s^c-1} b_s^c \cdot R[t^s, U(t^s)] + b_{N_s^c}^c \cdot R[t_s^{N_s^p}, U(t_s^{N_s^p})_p] \end{aligned} \quad (8)$$

where $N_s^{p,c}$ is the number of time steps considered for the Adams-Bashforth predictor/Adams-Moulton corrector (respectively) and $b_s^{p,c}$ are the corresponding linear coefficients selected. Essentially, the Adams-Bashforth prediction $U(t_s^{N_s^p})_p$ is corrected by means of the Adams-Moulton correction resulting in $U(t_s^{N_s^c})_c$. In order to preserve the formal order of accuracy the relation $N_s^p = N_s^c + 1$ always holds.

Currently FOODIE provides $N_s^c = 1, 2, 3 \rightarrow N_s^c = 2, 3, 4$ steps schemes having 2nd, 3rd and 4th formal order of accuracy, respectively. The $b_s^{p,c}$ coefficients are those reported in Tables 6 and 7.

2.7. The Leapfrog Solver

The *leapfrog* scheme belongs to the multi-step family, it being formally a centered second order approximation in time, see [18–21]. The leapfrog method (in its original formulation) is mostly unstable, however it is well suited for periodic-oscillatory problems providing a null error on the amplitude value and a formal second order error on the phase one, under the satisfaction of the time-step size stable limit. Commonly, the leapfrog methods are said to provide a $2\Delta t$ computational mode that can generate unphysical, unstable solutions. As consequence, the original leapfrog scheme is generally *filtered* in order to suppress these computational modes.

The unfiltered leapfrog scheme provided by FOODIE is:

$$U(t^{n+2}) = U(t^n) + 2\Delta t \cdot R[t^{n+1}, U(t^{n+1})] \quad (9)$$

FOODIE provides, in a *seamless* API, also filtered leapfrog schemes. A widely used filter is due to Robert and Asselin, that suppress the computational modes at the cost of accuracy reduction resulting into a 1st order error in amplitude value. A more accurate filter, able to provide a 3rd order error on amplitude, is a modification of the Robert-Asselin filter due to Williams known as Robert-Asselin-Williams (RAW) filter, that filters the approximation of $U(t^{n+1})$ and $U(t^{n+2})$ by the following scalar coefficient:

$$\begin{aligned} U(t^{n+1}) &= U(t^{n+1}) + \Delta * \alpha \\ U(t^{n+2}) &= U(t^{n+2}) + \Delta * (\alpha - 1) \end{aligned} \quad (10)$$

where

$$\Delta = \frac{\nu}{2} (U^n - 2U^{n+1} + U^{n+2})$$

The filter coefficients should be taken as $\nu \in (0, 1]$ and $\alpha \in (0.5, 1]$. If $\alpha = 0.5$ the filters of time t^{n+1} and t^{n+2} have the same amplitude and opposite sign thus allowing to the optimal 3rd order error on amplitude. The default values of the FOODIE provided scheme are $\nu = 0.01$ $\alpha = 0.53$, but they can be customized at runtime. The RA-filtered leapfrog scheme is widely used in numerical weather prediction and, in general, in atmospheric/oceanic circulation models probably because it is really simple to implement and has a good accuracy. However, the RAW filter reported in (10) is a major upgrade and generalizes a family of leapfrog solver ranging from unconditionally unstable, to conditionally stable, and up to unconditionally stable. Tuning the family coefficients it is possible to reach (almost) the ideal third order accuracy in amplitude while retaining a conditional stability, see [21].

3. Application Program Interface

In this section we review the FOODIE API providing a detailed discussion of the implementation choices.

As aforementioned, the programming language used is the Fortran 2008 standard, that is a minor revision of the previous Fortran 2003 standard. Such a new Fortran idioms provide (among other useful features) an almost complete support for OOP, in particular for ADT concept. Fortran 2003 has introduced the *abstract derived type*: it is a derived type suitable to serve as *contract* for concrete type-extensions that has not any actual implementations, rather it provides a well-defined set of type bound procedures interfaces, that in Fortran nomenclature are called *deferred* procedures. Using such an abstract definition, we can implement algorithms operating on only this abstract type and on *all its concrete extensions*. This is the key feature of FOODIE library: all the above described ODE solvers are implemented on the knowledge of *only one abstract type*, allowing an implementation-style based on a very high-level syntax. In the meanwhile, client codes must implement their own IVPs extending only one simple abstract type.

In the Section 3.1 a review of the FOODIE main ADT, the *integrand* type, is provided, while Sections 3.2–3.5 and 3.8 cover the API of the currently implemented solvers.

It is worth noting that all FOODIE public *entities* (ADT and solvers) must be accessed by the FOODIE module, see Listing 1 for an example on how access to all public FOODIE entities.

Listing 1. Usage example importing all public entities of FOODIE main module.

```
use foodie, only: integrand, &
                 adams_bashforth_integrator, &
                 adams_moulton_integrator, &
                 adams_bashforth_moulton_integrator, &
                 euler_explicit_integrator, &
                 leapfrog_integrator, &
                 ls_runge_kutta_integrator, &
                 tvd_runge_kutta_integrator

! or simply
use foodie
```

3.1. The Main FOODIE Abstract Data Type: The Integrand Type

The implemented ACP is based on one main ADT, the *integrand* type, the definition of which is shown in Listing 2.

Listing 2. Integrand type definition.

```
type, abstract :: integrand
!< Abstract type for building FOODIE ODE integrators.
contains
! public deferred procedures that concrete integrand-field must implement
procedure(time_derivative), pass(self), deferred, public:: t
! operators
procedure(symmetric_operator), pass(lhs), deferred, public:: integrand_multiply_integrand
procedure(integrand_op_real), pass(lhs), deferred, public:: integrand_multiply_real
procedure(real_op_integrand), pass(rhs), deferred, public:: real_multiply_integrand
procedure(symmetric_operator), pass(lhs), deferred, public:: add
procedure(symmetric_operator), pass(lhs), deferred, public:: sub
procedure(assignment_integrand), pass(lhs), deferred, public:: assign_integrand
! operators overloading
generic, public:: operator(+) => add
generic, public:: operator(-) => sub
generic, public:: operator(*) => integrand_multiply_integrand, &
                             real_multiply_integrand, &
                             integrand_multiply_real
generic, public:: assignment(=) => assign_integrand
endtype integrand
```

The *integrand* type does not implement any actual integrand field, it being an abstract type. It only specifies which deferred procedures are necessary for implementing an actual concrete integrand type that can use a FOODIE solver.

As shown in Listing 2, the number of the deferred type bound procedures that clients must implement into their own concrete extension of the *integrand* ADT is very limited:

essentially, there are 1 ODE-specific procedure plus some operators definition constituted by symmetric operators between 2 integrand objects, asymmetric operators between integrand and real numbers (and viceversa) and an assignment statement for the creation of new integrand objects. These procedures are analyzed in the following paragraphs.

3.1.1. Time Derivative Procedure, the Residuals Function

The abstract interface of the time derivative procedure *t* is shown in Listing 3.

Listing 3. Time derivative procedure interface.

```
function time_derivative(self, t) result(dState_dt)
import :: integrand, R_P, I_P
class(integrand), intent(IN) :: self      !< Integrand field.
real(R_P), optional, intent(IN) :: t      !< Time.
class(integrand), allocatable :: dState_dt !< Result of the time derivative function of integrand field.
endfunction time_derivative
```

This procedure-function takes two arguments, the first passed as a *type bounded* argument, while the latter is optional, and it returns an integrand object. The passed dummy argument, *self*, is a polymorphic argument that could be any extensions of the *integrand* ADT. The optional argument *t* is the *time* at which the residuals function must be computed: it can be omitted in the case the residuals function does not depend directly on time.

Commonly, into the concrete implementation of this deferred abstract procedure clients embed the actual ODE equations being solved. As an example, for the Burgers equation, that is a Partial Differential Equations (PDE) system involving also a boundary value problem, this procedure embeds the spatial operator that convert the PDE to a system of algebraic ODE. As a consequence, the eventual concrete implementation of this procedure can be very complex and errors-prone. Nevertheless, the FOODIE solvers are implemented only on the above abstract interface, thus emancipating the solvers implementation from any concrete complexity.

3.1.2. Symmetric operators procedures

The abstract interface of *symmetric* procedures is shown in Listing 4.

Listing 4. Symmetric operator procedure interface.

```
function symmetric_operator(lhs, rhs) result(operator_result)
import :: integrand
class(integrand), intent(IN) :: lhs      !< Left hand side.
class(integrand), intent(IN) :: rhs      !< Right hand side.
class(integrand), allocatable :: operator_result !< Operator result.
endfunction symmetric_operator
```

This interface defines a class of procedures operating on 2 *integrand* objects, namely it is used for the definition of the operators *multiplication*, *summation* and *subtraction* of integrand objects. These operators are used into the above described ODE solvers, for example see Equations (2), (3), (6) or (9). The implementation details of such a procedures class are strictly dependent on the concrete extension of the integrand type. From the FOODIE solvers point of view, we need to know only that first argument passed as bounded one, the left-hand-side of the operator, and the second argument, the right-hand-side of the operator, are two integrand object and the returned object is still an integrand one. This *agnostic nature* is a feature shared by all FOODIE operators.

3.1.3. Integrand/Real and Real/Integrand Operators Procedures

The abstract interfaces of *Integrand/real* and *real/integrand operators* procedures are shown in Listing 5.

Listing 5. Integrand/real and real/integrand operators procedure interfaces.

```
function integrand_op_real(lhs, rhs) result(operator_result)
import :: integrand, R_P
class(integrand), intent(IN) :: lhs      !< Left hand side.
real(R_P), intent(IN) :: rhs            !< Right hand side.
```

```

class(integrand), allocatable :: operator_result !< Operator result.
endfunction integrand_op_real

function real_op_integrand(lhs, rhs) result(operator_result)
import :: integrand, R_P
real(R_P), intent(IN) :: lhs !< Left hand side.
class(integrand), intent(IN) :: rhs !< Right hand side.
class(integrand), allocatable :: operator_result !< Operator result.
endfunction real_op_integrand

```

These two interfaces are necessary in order to complete the *algebra* operating on the integrand object class, allowing the multiplication of an integrand object for a real number, circumstance that happens in all solvers, see Equations (2), (3), (6) or (9). The implementation details of these procedures are strictly dependent on the concrete extension of the integrand type.

3.1.4. Integrand Assignment Procedure

The abstract interface of *integrand assignment* procedure is shown in Listing 6.

Listing 6. Integrand assignment procedure interface.

```

subroutine assignment_integrand(lhs, rhs)
import :: integrand
class(integrand), intent(INOUT) :: lhs !< Left hand side.
class(integrand), intent(IN) :: rhs !< Right hand side.
endsubroutine assignment_integrand

```

The assignment statement is necessary in order to complete the *algebra* operating on the integrand object class, allowing the assignment of an integrand object by another one, circumstance that happens in all solvers, see Equations (2), (3), (6) or (9). The implementation details of this assignment is strictly dependent on the concrete extension of the integrand type.

3.2. The Explicit forward Euler Solver

The explicit forward Euler solver is exposed (by the FOODIE main module that must be imported, see Listing 1) as a single derived type (that is a standard convention for all FOODIE solvers) named *euler_explicit_integrator*. It provides the type bound procedure (also referred as *method*) *integrate* for integrating in time an *integrand* object, or any of its polymorphic concrete extensions. Consequently, for using such a solver it must be previously defined as an instance of the exposed FOODIE integrator type, see Listing 7.

Listing 7. Definition of an explicit forward Euler integrator.

```

use FOODIE, only: euler_explicit_integrator
type(euler_explicit_integrator) :: integrator

```

Once an integrator of this type has been instantiated, it can be directly used without any initialization, for example see Listing 8.

Listing 8. Example of usage of an explicit forward Euler integrator.

```

type(my_integrand) :: my_field
call integrator%integrate(U=my_field, Dt=0.1)

```

where *my_integrand* is a concrete (valid) extension of *integrand* ADT.

The complete implementation of the *integrate* method of the explicit forward Euler solver is reported in Listing 9.

Listing 9. Implementation of the *integrate* method of Euler solver.

```

subroutine integrate(U, Dt, t)
class(integrand), intent(INOUT) :: U !< Field to be integrated.
real(R_P), intent(IN) :: Dt !< Time step.
real(R_P), optional, intent(IN) :: t !< Time.
U = U + U%(t=t) * Dt
return
endsubroutine integrate

```

This method takes three arguments, the first argument is an integrand class, it being the integrand field that must be integrated one-step-over in time, the second is the time step used and the third, that is optional, is the current time value that is passed to the residuals function for taking into account the cases where the time derivative explicitly depends on time. The time step is not automatically computed (for example inspecting the passed integrand field), thus its value must be externally computed and passed to the *integrate* method.

3.3. The Explicit TVD/SSP Runge-Kutta Class of Solvers

The TVD/SSP Runge-Kutta class of solvers is exposed as a single derived type named *tvd_runge_kutta_integrator*. This type provides three methods:

- *init*: initialize the integrator accordingly the possibilities offered by the class of solvers;
- *destroy*: destroy the integrator previously initialized, eventually freeing the allocated dynamic memory registers;
- *integrate*: integrate integrand field one-step-over in time.

As common for FOODIE solvers, for using such a solver it must be previously defined as an instance of the exposed FOODIE integrator type, see Listing 10.

Listing 10. Definition of an explicit TVD/SSP Runge-Kutta integrator.

```
use FOODIE, only: tvd_runge_kutta_integrator
type(tvd_runge_kutta_integrator) :: integrator
```

Once an integrator of this type has been instantiated, it must be initialized before used, for example see Listing 11.

Listing 11. Example of initialization of an explicit TVD/SSP Runge-Kutta integrator.

```
call integrator%init(stages=3)
```

In the Listing 11 a 3-stages solver has been initialized. As a matter of facts, from the Equations (3) and (4) a solver belonging to this class is completely defined once the number of stages adopted has been chosen. The complete definition of the *tvd_runge_kutta_integrator* type is reported into Listing 12. As shown, the Butcher's coefficients are stored as allocatable arrays the values of which are initialized by the *init* method.

Listing 12. Definition of *tvd_runge_kutta_integrator* type.

```
type :: tvd_runge_kutta_integrator
integer(I_P) :: stages=0 ! Number of stages.
real(R_P), allocatable :: alph(:,:) ! alpha Butcher's coefficients.
real(R_P), allocatable :: beta(:) ! beta Butcher's coefficients.
real(R_P), allocatable :: gamm(:) ! gamma Butcher's coefficients.
contains
  procedure, pass(self), public :: destroy
  procedure, pass(self), public :: init
  procedure, pass(self), public :: integrate
  final :: finalize
endtype tvd_runge_kutta_integrator
```

After the solver has been initialized it can be used for integrating an integrand field, as shown in Listing 13.

Listing 13. Example of usage of a TVD/SSP Runge-Kutta integrator.

```
type(my_integrand) :: my_field
type(my_integrand) :: my_stages(1:3)
call integrator%integrate(U=my_field, stage=my_stage, Dt=0.1)
```

where *my_integrand* is a concrete (valid) extension of *integrand* ADT. Listing 13 shows that the memory registers necessary for storing the Runge-Kutta stages must be supplied by the client code.

The complete implementation of the *integrate* method of the explicit TVD/SSP Runge-Kutta class of solvers is reported in Listing 14.

Listing 14. Implementation of the *integrate* method of explicit TVD/SSP Runge-Kutta class.

```

subroutine integrate(self, U, stage, Dt, t)
class(tvd_runge_kutta_integrator), intent(IN) :: self      ! Actual RK integrator.
class(integrand), intent(INOUT) :: U                    ! Field to be integrated.
class(integrand), intent(INOUT) :: stage(1:)            ! Runge-Kutta stages [1:stages].
real(R_P), intent(IN) :: Dt                             ! Time step.
real(R_P), intent(IN) :: t                             ! Time.
integer(I_P) :: s                                      ! First stages counter.
integer(I_P) :: ss                                     ! Second stages counter.
select type(stage)
class is(integrand)
  do s=1, self%stages
    stage(s) = U
    do ss=1, s - 1
      stage(s) = stage(s) + stage(ss) * (Dt * self%alph(s, ss))
    enddo
    stage(s) = stage(s)%t(t=t + self%gamma(s) * Dt)
  enddo
  do s=1, self%stages
    U = U + stage(s) * (Dt * self%beta(s))
  enddo
endselect
return
endsubroutine integrate

```

This method takes five arguments, the first argument is passed as bounded argument and it is the solver itself, the second is of an integrand class, it being the integrand field that must be integrated one-step-over in time, the third is the stages array for storing the stages computations, the fourth is the time step used and the fifth, that is optional, is the current time value that is passed to the residuals function for taking into account the cases where the time derivative explicitly depends on time. The time step is not automatically computed (for example inspecting the passed integrand field), thus its value must be externally computed and passed to the *integrate* method.

It is worth noting that the stages memory registers, namely the array *stage*, must be passed as argument because it is defined as a *not-passed* polymorphic argument, thus we are not allowed to define it as an automatic array of the *integrate* method.

3.4. The Explicit Low Storage Runge-Kutta Class of Solvers

The low storage variant of Runge-Kutta class of solvers is exposed as a single derived type named *ls_runge_kutta_integrator*. This type provides three methods:

- *init*: initialize the integrator accordingly the possibilities offered by the class of solvers;
- *destroy*: destroy the integrator previously initialized, eventually freeing the allocated dynamic memory registers;
- *integrate*: integrate integrand field one-step-over in time.

As common for FOODIE solvers, for using such a solver it must be previously defined as an instance of the exposed FOODIE integrator type, see Listing 15.

Listing 15. Definition of an explicit low storage Runge-Kutta integrator.

```

use FOODIE, only: ls_runge_kutta_integrator
type(ls_runge_kutta_integrator) :: integrator

```

Once an integrator of this type has been instantiated, it must be initialized before used, for example see Listing 16.

Listing 16. Example of initialization of an explicit low storage Runge-Kutta integrator.

```

call integrator%init(stages=5)

```

In the Listing 16 a 5-stages solver has been initialized. As a matter of facts, from the Equation (5) a solver belonging to this class is completely defined once the number of stages adopted has been chosen. The complete definition of the *ls_runge_kutta_integrator* type is reported into Listing 17. As shown, the Williamson's coefficients are stored as allocatable arrays the values of which are initialized by the *init* method.

Listing 17. Definition of *ls_runge_kutta_integrator* type.

```

type :: ls_runge_kutta_integrator
  integer(I_P)      :: stages=0 ! Number of stages.
  real(R_P), allocatable :: A(:) ! Low storage*A* coefficients.
  real(R_P), allocatable :: B(:) ! Low storage*B* coefficients.
  real(R_P), allocatable :: C(:) ! Low storage*C* coefficients.
contains
  procedure, pass(self), public :: destroy
  procedure, pass(self), public :: init
  procedure, pass(self), public :: integrate
  final                          :: finalize
endtype ls_runge_kutta_integrator

```

After the solver has been initialized it can be used for integrating an integrand field, as shown in Listing 18.

Listing 18. Example of usage of a low storage Runge-Kutta integrator.

```

type(my_integrand) :: my_field
type(my_integrand) :: my_stages(1:2)
call integrator%integrate(U=my_field, stage=my_stage, Dt=0.1)

```

where *my_integrand* is a concrete (valid) extension of *integrand* ADT. Listing 18 shows that the memory registers necessary for storing the Runge-Kutta stages must be supplied by the client code, as it happens of the TVD/SSP Runge-Kutta class. However, now the registers necessary is always 2, independently on the number of stages used, that in the example considered are 5.

The complete implementation of the *integrate* method of the explicit low storage Runge-Kutta class of solvers is reported in Listing 19.

Listing 19. Implementation of the *integrate* method of explicit low storage Runge-Kutta class.

```

subroutine integrate(self, U, stage, Dt, t)
class(ls_runge_kutta_integrator), intent(IN) :: self ! Actual RK integrator.
class(integrand), intent(INOUT) :: U ! Field to be integrated.
class(integrand), intent(INOUT) :: stage(1:2) ! Runge-Kutta registers [1:2].
real(R_P), intent(IN) :: Dt ! Time step.
real(R_P), intent(IN) :: t ! Time.
integer(I_P) :: s ! First stages counter.
select type(stage)
class is(integrand)
  stage(1) = U
  stage(2) = U*0._R_P
  do s=1, self%stages
    stage(2) = stage(2) * self%A(s) + stage(1)%t(t=t + self%C(s) * Dt) * Dt
    stage(1) = stage(1) + stage(2) * self%B(s)
  enddo
  U = stage(1)
endselect
return
endsubroutine integrate

```

This method takes five arguments, the first argument is passed as bounded argument and it is the solver itself, the second is of an integrand class, it being the integrand field that must integrated one-step-over in time, the third is the stages array for storing the stages computations, the fourth is the time step used and the fifth, that is optional, is the current time value that is passed to the residuals function for taking into account the cases where the time derivative explicitly depends on time. The time step is not automatically computed (for example inspecting the passed integrand field), thus its value must be externally computed and passed to the *integrate* method.

It is worth noting that the stages memory registers, namely the array *stage*, must be passed as argument because it is defined as a *not-passed* polymorphic argument, thus we are not allowed to define it as an automatic array of the *integrate* method.

3.5. The Explicit Adams-Bashforth Class of Solvers

The explicit Adams-Bashforth class of solvers is exposed as a single derived type named *adams_bashforth_integrator*. This type provides three methods:

- *init*: initialize the integrator accordingly the possibilities offered by the class of solvers;

- *destroy*: destroy the integrator previously initialized, eventually freeing the allocated dynamic memory registers;
- *integrate*: integrate integrand field one-step-over in time;
- *update_previous*: auto update (cyclically) previous time steps solutions.

As common for FOODIE solvers, for using such a solver it must be previously defined as an instance of the exposed FOODIE integrator type, see Listing 20.

Listing 20. Definition of an explicit Adams-Bashforth integrator.

```
use FOODIE, only: adams_bashforth_integrator
type(adams_bashforth_integrator) :: integrator
```

Once an integrator of this type has been instantiated, it must be initialized before used, for example see Listing 21.

Listing 21. Example of initialization of an explicit Adams-Bashforth integrator.

```
call integrator%init(steps=4)
```

In the Listing 21 a 4-steps solver has been initialized. As a matter of facts, from the Equation (6) a solver belonging to this class is completely defined once the number of time steps adopted has been chosen. The complete definition of the *adams_bashforth_integrator* type is reported into Listing 22. As shown, the linear coefficients are stored as allocatable arrays the values of which are initialized by the *init* method.

Listing 22. Definition of *adams_bashforth_integrator* type.

```
type :: adams_bashforth_integrator
private
integer(I_P) :: steps=0 ! Number of time steps.
real(R_P), allocatable :: b(:) ! b coefficients.
contains
procedure, pass(self), public :: destroy
procedure, pass(self), public :: init
procedure, pass(self), public :: integrate
procedure, pass(self), public :: update_previous
final :: finalize
endtype adams_bashforth_integrator
```

After the solver has been initialized it can be used for integrating an integrand field, as shown in Listing 23.

Listing 23. Example of usage of an Adams-Bashforth integrator.

```
real :: times(1:4)
type(my_integrand) :: my_field
type(my_integrand) :: previous(1:4)
call integrator%integrate(U=my_field, previous=previous, Dt=Dt, t=times)
```

where *my_integrand* is a concrete (valid) extension of *integrand* ADT, *times* are the time at each 4 steps considered for the current one-step-over integration and *previous* are the memory registers where previous time steps solutions are saved.

The complete implementation of the *integrate* method of the explicit Adams-Bashforth class of solvers is reported in Listing 24.

Listing 24. Implementation of the *integrate* method of explicit Adams-Bashforth class.

```
subroutine integrate(self, U, previous, Dt, t, autoupdate)
class(adams_bashforth_integrator), intent(IN) :: self ! Actual AB integrator.
class(integrand), intent(INOUT) :: U ! Field to be integrated.
class(integrand), intent(INOUT) :: previous(1:) ! Previous time steps solutions.
real(R_P), intent(IN) :: Dt ! Time steps.
real(R_P), intent(IN) :: t(:) ! Times.
logical, optional, intent(IN) :: autoupdate ! Autoupdate previous time steps.
logical :: autoupdate_ ! Autoupdate previous time steps.
integer(I_P) :: s ! Steps counter.
autoupdate_ = .true. ; if (present(autoupdate)) autoupdate_ = autoupdate
do s=1, self%steps
U = U + previous(s)%t(t=t(s)) * (Dt * self%b(s))
enddo
```

```

if (autoupdate_) call self%update_previous(U=U, previous=previous)
return
endsubroutine integrate

```

This method takes five arguments, the first argument is passed as bounded argument and it is the solver itself, the second is of an integrand class, it being the integrand field that must be integrated one-step-over in time, the third are the previous time steps solutions, the fourth is the time step used, the fifth is an array of the time values of the steps considered for the current one-step-over integration that are passed to the residuals function for taking into account the cases where the time derivative explicitly depends on time and the sixth is a logical flag for enabling/disabling the cyclic update of previous time steps solutions. The time step is not automatically computed (for example inspecting the passed integrand field), thus its value must be externally computed and passed to the *integrate* method.

It is worth noting that the method also performs the cyclic update of the previous time steps solutions memory registers. This can be disabled passing *autoupdate=false*: it is useful in the framework of predictor-corrector solvers.

3.6. The Implicit Adams-Moulton Class of Solvers

The implicit Adams-Moulton class of solvers is exposed as a single derived type named *adams_moulton_integrator*. This type provides three methods:

- *init*: initialize the integrator accordingly the possibilities offered by the class of solvers;
- *destroy*: destroy the integrator previously initialized, eventually freeing the allocated dynamic memory registers;
- *integrate*: integrate integrand field one-step-over in time;
- *update_previous*: auto update (cyclically) previous time steps solutions.

As common for FOODIE solvers, for using such a solver it must be previously defined as an instance of the exposed FOODIE integrator type, see Listing 25.

Listing 25. Definition of an implicit Adams-Moulton integrator.

```

use FOODIE, only: adams_moulton_integrator
type(adams_moulton_integrator) :: integrator

```

Once an integrator of this type has been instantiated, it must be initialized before used, for example see Listing 26.

Listing 26. Example of initialization of an implicit Adams-Moulton integrator.

```

call integrator%init(steps=3)

```

In the Listing 26 a 3-steps solver has been initialized. As a matter of facts, from the Equation (7) a solver belonging to this class is completely defined once the number of time steps adopted has been chosen. The complete definition of the *adams_moulton_integrator* type is reported into Listing 27. As shown, the linear coefficients are stored as allocatable arrays the values of which are initialized by the *init* method.

Listing 27. Definition of *adams_moulton_integrator* type.

```

type :: adams_moulton_integrator
  private
  integer(I_P)          :: steps=-1 ! Number of time steps.
  real(R_P), allocatable :: b(:)    ! b coefficients.
  contains
    procedure, pass(self), public :: destroy
    procedure, pass(self), public :: init
    procedure, pass(self), public :: integrate
    procedure, pass(self), public :: update_previous
    final                          :: finalize
endtype adams_moulton_integrator

```

After the solver has been initialized it can be used for integrating an integrand field, as shown in Listing 28.

Listing 28. Example of usage of an Adams-Moulton integrator.

```

real                :: times(1:3)
type(my_integrand) :: my_field
type(my_integrand) :: previous(1:3)
call integrator%integrate(U=my_field, previous=previous, Dt=Dt, t=times)

```

where *my_integrand* is a concrete (valid) extension of *integrand* ADT, *times* are the time at each 4 steps considered for the current one-step-over integration and *previous* are the memory registers where previous time steps solutions are saved.

The complete implementation of the *integrate* method of the implicit Adams-Moulton class of solvers is reported in Listing 29.

Listing 29. Implementation of the *integrate* method of explicit Adams-Moulton class.

```

subroutine integrate(self, U, previous, Dt, t, autoupdate)
class(adams_bashforth_integrator), intent(IN) :: self      ! Actual AB integrator.
class(integrand), intent(INOUT) :: U                      ! Field to be integrated.
class(integrand), intent(INOUT) :: previous(1:)           ! Previous time steps solutions.
real(R_P), intent(IN) :: Dt                               ! Time steps.
real(R_P), intent(IN) :: t(:)                             ! Times.
logical, optional, intent(IN) :: autoupdate               ! Autoupdate previous time steps.
logical :: autoupdate_ ! Autoupdate previous time steps.
integer(I_P) :: s ! Steps counter.
autoupdate_ = .true.; if (present(autoupdate)) autoupdate_ = autoupdate
if (autoupdate_) call self%update_previous(U=U, previous=previous)
if (self%steps>0) then
  U = previous(self%steps) + U%(t=t(self%steps) + Dt) * (Dt * self%b(self%steps))
  do s=0, self%steps - 1
    U = U + previous(s+1)%t(t=t(s+1)) * (Dt * self%b(s))
  enddo
  if (autoupdate_) call self%update_previous(U=U, previous=previous)
else
  U = U + U%(t=t(s+1)) * (Dt * self%b(0))
endif
return
endsubroutine integrate

```

This method takes six arguments, the first argument is passed as bounded argument and it is the solver itself, the second is of an integrand class, it being the integrand field that must be integrated one-step-over in time, the third are the previous time steps solutions, the fourth is the time step used, the fifth is an array of the time values of the steps considered for the current one-step-over integration that are passed to the residuals function for taking into account the cases where the time derivative explicitly depends on time and the sixth is a logical flag for enabling/disabling the cyclic update of previous time steps solutions. The time step is not automatically computed (for example inspecting the passed integrand field), thus its value must be externally computed and passed to the *integrate* method.

It is worth noting that the method also performs the cyclic update of the previous time steps solutions memory registers. This can be disabled passing *autoupdate=false*: it is useful in the framework of predictor-corrector solvers.

3.7. The Predictor-Corrector Adams-Bashforth-Moulton Class of Solvers

The predictor-corrector Adams-Bashforth-Moulton class of solvers is exposed as a single derived type named *adams_bashforth_moulton_integrator*. This type provides three methods:

- *init*: initialize the integrator accordingly the possibilities offered by the class of solvers;
- *destroy*: destroy the integrator previously initialized, eventually freeing the allocated dynamic memory registers;
- *integrate*: integrate integrand field one-step-over in time;

As common for FOODIE solvers, for using such a solver it must be previously defined as an instance of the exposed FOODIE integrator type, see Listing 30.

Listing 30. Definition of an implicit Adams-Moulton integrator.

```

use FOODIE, only: adams_bashforth_moulton_integrator
type(adams_bashforth_moulton_integrator) :: integrator

```

Once an integrator of this type has been instantiated, it must be initialized before used, for example see Listing 31.

Listing 31. Example of initialization of an implicit Adams-Moulton integrator.

```
call integrator%init(steps=3)
```

In the Listing 31 a 3-steps solver has been initialized. As a matter of facts, from the Equation (8) a solver belonging to this class is completely defined once the number of time steps adopted has been chosen. The complete definition of the *adams_moulton_integrator* type is reported into Listing 32. As shown, the linear coefficients are stored as allocatable arrays the values of which are initialized by the *init* method.

Listing 32. Definition of *adams_bashforth_moulton_integrator* type.

```
type, extends(integrator_multistep_object) :: integrator_adams_bashforth_moulton
private
type(integrator_adams_bashforth) :: predictor ! Predictor solver.
type(integrator_adams_moulton)   :: corrector ! Corrector solver.
contains
procedure, pass(self) :: destroy
procedure, pass(self) :: initialize
procedure, pass(self) :: scheme_number
endtype integrator_adams_bashforth_moulton
```

After the solver has been initialized it can be used for integrating an integrand field, as shown in Listing 33.

Listing 33. Example of usage of an Adams-Bashforth-Moulton integrator.

```
real :: times(1:3)
type(my_integrand) :: my_field
type(my_integrand) :: previous(1:3)
call integrator%integrate(U=my_field, previous=previous, Dt=Dt, t=times)
```

where *my_integrand* is a concrete (valid) extension of *integrand* ADT, *times* are the time at each 4 steps considered for the current one-step-over integration and *previous* are the memory registers where previous time steps solutions are saved.

The complete implementation of the *integrate* method of the predictor-corrector Adams-Bashforth-Moulton class of solvers is reported in Listing 34.

Listing 34. Implementation of the *integrate* method of predictor-corrector Adams-Bashforth-Moulton class.

```
subroutine integrate(self, U, Dt, t)
class(integrator_adams_bashforth_moulton), intent(inout) :: self ! Integrator.
class(integrand_object), intent(inout) :: U ! Field to be integrated.
real(R_P), intent(in) :: Dt ! Time steps.
real(R_P), intent(in) :: t ! Times.
integer(I_P) :: s ! Step counter.

do s=1, self%steps
self%predictor%previous(s) = self%previous(s)
self%predictor%t(s) = self%t(s)
self%predictor%Dt(s) = self%Dt(s)
enddo
do s=1, self%steps - 1
self%corrector%previous(s) = self%predictor%previous(s+1)
self%corrector%t(s) = self%predictor%t(s+1)
self%corrector%Dt(s) = self%predictor%Dt(s+1)
enddo
call self%predictor%integrate(U=U, Dt=Dt, t=t)
call self%corrector%integrate(U=U, Dt=Dt, t=t)
if (self%autoupdate) &
call self%update_previous(U=U, previous=self%previous, Dt=Dt, t=t, previous_t=self%t)
endsubroutine integrate
```

This method takes four arguments, the first argument is passed as bounded argument and it is the solver itself, the second is of an integrand class, it being the integrand field that must be integrated one-step-over in time, the third is the time step used, and the fourth is the current time. The time step is not automatically computed (for example inspecting

the passed integrand field), thus its value must be externally computed and passed to the *integrate* method.

3.8. The Leapfrog Solver

The explicit Leapfrog class of solvers is exposed as a single derived type named *leapfrog_integrator*. This type provides three methods:

- *init*: initialize the integrator accordingly the possibilities offered by the class of solvers;
- *integrate*: integrate integrand field one-step-over in time.

As common for FOODIE solvers, for using such a solver it must be previously defined as an instance of the exposed FOODIE integrator type, see Listing 35.

Listing 35. Definition of an explicit Leapfrog integrator.

```
use FOODIE, only: leapfrog_integrator
type(leapfrog_integrator) :: integrator
```

Once an integrator of this type has been instantiated, it must be initialized before used, for example see Listing 36.

Listing 36. Example of initialization of an explicit Leapfrog integrator.

```
! default coefficients nu=0.01, alpha=0.53
call integrator%init()
! custom coefficients
call integrator%init(nu=0.015, alpha=0.6)
```

The complete definition of the *leapfrog_integrator* type is reported into Listing 37. As shown, the filter coefficients are initialized to zero, suitable values are initialized by the *init* method.

Listing 37. Definition of *leapfrog_integrator* type.

```
type :: leapfrog_integrator
  private
  real(R_P) :: nu=0.01_R_P ! Robert-Asselin filter coefficient.
  real(R_P) :: alpha=0.53_R_P ! Robert-Asselin-Williams filter coefficient.
  contains
    procedure, pass(self), public :: init
    procedure, pass(self), public :: integrate
endtype leapfrog_integrator
```

After the solver has been initialized it can be used for integrating an integrand field, as shown in Listing 38.

Listing 38. Example of usage of a Leapfrog integrator.

```
real :: times(1:2)
type(my_integrand) :: filter_displacement
type(my_integrand) :: my_field
type(my_integrand) :: previous(1:2)
call integrator%integrate(U=my_field, previous=previous, filter=filter_displacement, Dt=Dt, &
                          t=times)
```

Where *my_integrand* is a concrete (valid) extension of *integrand* ADT, *previous* are the memory registers where previous time steps solutions are saved, *filter_displacement* is the register necessary for computing the eventual displacement of the applied filter and *times* are the time at each 2 steps considered for the current one-step-over integration.

The complete implementation of the *integrate* method of the explicit Leapfrog class of solvers is reported in Listing 39.

Listing 39. Implementation of the *integrate* method of explicit Leapfrog class.

```
subroutine integrate(self, U, previous, Dt, t, filter)
  class(leapfrog_integrator), intent(IN) :: self ! LF integrator.
  class(integrand), intent(INOUT) :: U ! Field to be integrated.
  class(integrand), intent(INOUT) :: previous(1:2) ! Previous time steps solutions.
  real(R_P), intent(in) :: Dt ! Time step.
  real(R_P), intent(IN) :: t ! Time.
```

```

class(integrand), optional, intent(INOUT) :: filter      ! Filter field displacement.
U = previous(1) + previous(2)%t(t=t) * (Dt * 2._R_P)
if (present(filter)) then
  filter = (previous(1) - previous(2) * 2._R_P + U) * self%au * 0.5_R_P
  previous(2) = previous(2) + filter * self%alpha
  U = U + filter * (self%alpha - 1._R_P)
endif
previous(1) = previous(2)
previous(2) = U
return
endsubroutine integrate

```

This method takes six arguments, the first argument is passed as bounded argument and it is the solver itself, the second is of an integrand class, it being the integrand field that must be integrated one-step-over in time, the third are the previous time steps solutions, the fourth is the optional filter-displacement-register, the fifth is the time step used and the sixth is an array of the time values of the steps considered for the current one-step-over integration that are passed to the residuals function for taking into account the cases where the time derivative explicitly depends on time. The time step is not automatically computed (for example inspecting the passed integrand field), thus its value must be externally computed and passed to the *integrate* method. It is worth noting that if the filter displacement argument is not passed, the solver reverts back to the standard unfiltered Leapfrog method.

It is worth noting that the method also performs the cyclic update of the previous time steps solutions memory registers. In particular, if the filter displacement argument is passed the method performs the RAW filtering.

3.9. General Remarks

Table 8 presents a comparison of the relevant parts of Equations (2)–(6) and (9) with the corresponding FOODIE implementations reported in Listings 9, 14, 19, 24 and 39, respectively. This comparison proves that the *integrand* ADT has allowed a very high-level implementation syntax. The Fortran implementation is almost equivalent to the rigorous mathematical formulation. This aspect directly implies that the implementation of a ODE solver into the FOODIE library is very clear, concise and less-errors-prone than an *hard-coded* implementation where the solvers must be implemented for each specific definition of the integrand type, it being not abstract.

Table 8. Comparison between rigorous mathematical formulation and FOODIE high-level implementation; the syntax “(s)” and “(ss)” imply the summation operation.

Solver	Mathematical formulation	FOODIE implementation
explicit forward Euler	$U(t^{n+1}) = U(t^n) + \Delta t \cdot R[t^n, U(t^n)]$	$U = U + U\%t(t = t) * Dt$
TVD/SSP Runge-Kutta	$K_s = R\left(t^n + \gamma_s \Delta t, U^n + \Delta t \sum_{l=1}^{s-1} \alpha_{s,l} K_l\right)$ $U^{n+1} = U^n + \Delta t \cdot \sum_{s=1}^{N_s} \beta_s K_s$	$stage(s) = stage(s) + stage(ss) * (Dt * self\%alpha(s, ss))$ $U = U + stage(s) * (Dt * self\%beta(s))$
low storage Runge-Kutta	$K_2 = A_s K_1 + \Delta t \cdot R(t^n + C_s \Delta t, K_1)$ $K_1 = K_1 + B_s K_2$	$stage(2) = stage(2) * self\%A(s) +$ $+ stage(1)\%t(t = t + self\%C(s) * Dt) * Dt$ $stage(1) = stage(1) + stage(2) * self\%B(s)$
explicit Adams-Bashforth	$U(t^{n+N_s}) = U(t^{n+N_s-1}) +$ $+ \Delta t \sum_{s=1}^{n+N_s} b_s \cdot R[t^{n+s-1}, U(t^{n+s-1})]$	$U = U + U\%t(n = s, t = t(s)) * (Dt * self\%b(s))$
explicit Leapfrog	$U(t^{n+2}) = U(t^n) + 2\Delta t \cdot R[t^{n+1}, U(t^{n+1})]$	$U = U\%previous_step(n = 1) + U\%t(n = 2, t = t) * (Dt * 2.)$

4. Tests and Examples

For the assessment of FOODIE capabilities the *oscillator* test is considered:

4.1. Oscillation Equations Test

Let us consider the *oscillator* problem, it being a simple, yet interesting IVP. Briefly, the oscillator problem is a prototype problem of non dissipative, oscillatory phenomena.

For example, let us consider a pendulum subjected to the Coriolis accelerations without dissipation, the motion equations of which can be described by the ODE system (11).

$$U_t = R(U) \quad U = \begin{bmatrix} x \\ y \end{bmatrix} \quad R(U) = \begin{bmatrix} -fy \\ fx \end{bmatrix} \quad (11)$$

where the frequency is chosen as $f = 10^{-4}$. The ODE system (11) is completed by the following initial conditions:

$$\begin{aligned} x(t_0) &= 0 \\ y(t_0) &= 1 \end{aligned} \quad (12)$$

where $t_0 = 0$ is the initial time considered.

The IVP constituted by Equations (11) and (12) is (apparently) simple and its exact solution is known:

$$\begin{aligned} x(t_0 + \Delta t) &= X_0 \cos(f\Delta t) - y_0 \sin(f\Delta t) \\ y(t_0 + \Delta t) &= X_0 \sin(f\Delta t) + y_0 \cos(f\Delta t) \end{aligned} \quad (13)$$

where Δt is an arbitrary time step. This problem is non-stiff meaning that the solution is constituted by only one time-scale, namely the single frequency f .

This problem is only apparently simple. As a matter of facts, in a non dissipative oscillatory problem the eventual errors in the amplitude approximation can rapidly drive the subsequent series of approximations to an unphysical solution. This is of particular relevance if the solution (that is numerically approximated) constitutes a *prediction* far from the initial conditions, that is the common case in weather forecasting.

Because the Oscillation system (11) posses a closed exact solution, the discussion on this test has twofolds aims: to assess the accuracy of the FOODIE's built-in solvers comparing the numerical solutions with the exact one and to demonstrate how it is simple to solve this prototypical problem by means of FOODIE.

4.1.1. Errors Analysis

For the analysis of the accuracy of each solver, we have integrated the Oscillation Equation (11) with different, decreasing time steps in the range [5000, 2500, 1250, 625, 320, 100]. The error is estimated by the L2 norm of the difference between the exact (U_e) and the numerical ($U_{\Delta t}$) solutions for each time step:

$$\varepsilon(\Delta t) = \|U_e - U_{\Delta t}\|_2 = \sqrt{\sum_{s=1}^{N_s} (U_e(t_0 + s * \Delta t) - U_{\Delta t}(t_0 + s * \Delta t))^2} \quad (14)$$

where N_s is the total number of time steps performed to reach the final integration time.

Using two pairs of subsequent-decreasing time steps solution is possible to estimate the order of accuracy of the solver employed computing the *observed order* of accuracy:

$$p = \frac{\log_{10}\left(\frac{\varepsilon(\Delta t_1)}{\varepsilon(\Delta t_2)}\right)}{\log_{10}\left(\frac{\Delta t_1}{\Delta t_2}\right)} \quad (15)$$

where $\frac{\Delta t_1}{\Delta t_2} > 1$.

4.1.2. FOODIE Aware Implementation of an Oscillation Numerical Solver

The IVP (11) can be easily solved by means of FOODIE library. The first block of a FOODIE aware solution consists to define an *oscillation integrand field* defining a concrete extension of the FOODIE *integrand* type. Listing 40 reports the implementation of such an integrand field that is contained into the tests suite shipped within the FOODIE library.

Listing 40. Implementation of the *oscillation integrand* type.

```

type, extends(integrand) :: oscillation
  private
  integer(I_P) :: dims=0 ! Space dimensions.
  real(R_P) :: f=0._R_P ! Oscillation frequency (Hz).
  real(R_P), dimension(:), allocatable :: U ! Integrand (state) variables, [1:dims].
contains
  ! auxiliary methods
  procedure, pass(self), public :: init
  procedure, pass(self), public :: output
  ! type_integrand deferred methods
  procedure, pass(self), public :: t => dOscillation_dt
  procedure, pass(lhs), public :: integrand_multiply_integrand => &
    oscillation_multiply_oscillation
  procedure, pass(lhs), public :: integrand_multiply_real => oscillation_multiply_real
  procedure, pass(rhs), public :: real_multiply_integrand => real_multiply_oscillation
  procedure, pass(lhs), public :: add => add_oscillation
  procedure, pass(lhs), public :: sub => sub_oscillation
  procedure, pass(lhs), public :: assign_integrand => oscillation_assign_oscillation
  procedure, pass(lhs), public :: assign_real => oscillation_assign_real
endtype oscillation

```

The *oscillation* field extends the *integrand* ADT making it a concrete type. This derived type is very simple: it has 5 data members for storing the state vector and some auxiliary variables, and it implements all the deferred methods necessary for defining a valid concrete extension of the *integrand* ADT (plus 2 auxiliary methods that are not relevant for our discussion). The key point is here constituted by the implementation of the deferred methods: the *integrand* ADT does not impose any structure for the data members, that are consequently free to be customized by the client code. In this example the data members have a very simple, clean and concise structure:

- *dims* is the number of space dimensions; in the case of Equation (11) we have $dims = 2$, however the test implementation has been kept more general parametrizing this dimension in order to easily allow future modification of the test-program itself;
- *f* stores the frequency of the oscillatory problem solved, that is here set to 10^4 , but it can be changed at runtime in the test-program;
- *U* is the state vector corresponding directly to the state vector of Equation (11);

As the Listing 40 shows, the FOODIE implementation strictly corresponds to the mathematical formulation embracing all the relevant mathematical aspects into one derived type, a single *object*. Here we not review the implementation of all deferred methods, this being out of the scope of the present work: the interested reader can see the tests suite sources shipped within the FOODIE library. However, some of these methods are relevant for our discussion, thus they are reviewed.

dOscillation_dt, the Oscillation Residuals Function

Probably, the most important methods for an IVP solver is the residuals function. As a matter of facts, the ODE equations are implemented into the residuals function. However, the FOODIE ADT strongly alleviates the subtle problems that could arise when the ODE solver is hard-implemented within the specific ODE equations. As a matter of facts, the *integrand* ADT specifies the precise interface the residuals function must have: if the client code implements a compliant interface, the FOODIE solvers will work as expected, reducing the possible errors location into the ODE equations, having designed the solvers on the ADT and not on the concrete type.

Listing 41 reports the implementation of the oscillation residuals function: it is very clear and concise. Moreover, comparing this listing with the Equation (11) the close correspondence between the mathematical formulation and Fortran implementation is evident.

Listing 41. Implementation of the *oscillation integrand* residuals function.

```

function dOscillation_dt(self, t) result(dState_dt)
class(oscillation), intent(IN) :: self ! Oscillation field.
real(R_P), optional, intent(IN) :: t ! Time.
class(integrand), allocatable :: dState_dt ! Oscillation field time derivative.
integer(I_P) :: dn ! Time level, dummy variable.

```

```

allocate(oscillation :: dState_dt)
select type(dState_dt)
class is(oscillation)
  dState_dt = self
  dState_dt%U(1) = -self%f * self%U(2)
  dState_dt%U(2) = self%f * self%U(1)
endselect
return
endfunction dOscillation_dt

```

Add Method, an Example of Oscillation Symmetric Operator

As a prototype of the operators overloading let us consider the *add* operator, it being a prototype of symmetric operators, the implementation of which is presented in Listing 42.

Listing 42. Implementation of the *oscillation integrand* add operator.

```

function add_oscillation(lhs, rhs) result(opr)
class(oscillation), intent(IN) :: lhs ! Left hand side.
class(integrand), intent(IN) :: rhs ! Right hand side.
class(integrand), allocatable :: opr ! Operator result.
allocate(oscillation :: opr)
select type(opr)
class is(oscillation)
  opr = lhs
  select type(rhs)
  class is(oscillation)
    opr%U = lhs%U + rhs%U
  endselect
endselect
return
endfunction add_Oscillation

```

It is very simple and clear: firstly all the auxiliary data are copied into the operator result, then the state vector of the result is populated with the addition between the state vectors of the left-hand-side and right-hand-side. This is very intuitive from the mathematical point of view and it helps to reduce implementation errors. Similar implementations are possible for all the other operators necessary to define a valid *integrand* ADT concrete extension.

Assignment of an Oscillation Object

The assignment overloading of the *oscillation* type is the last key-method that enforces the conciseness of the FOODIE aware implementation. Listing 43 reports the implementation of the assignment overloading. Essentially, to all the data members of the left-hand-side are assigned the values of the corresponding right-hand-side. Notably, for the assignment of the state vector and of the previous time steps solution array we take advantage of the automatic re-allocation of the left-hand-side variables when they are not allocated or allocated differently from the right-hand-side, that is a Fortran 2003 feature. In spite its simplicity, the assignment overloading is a key-method enabling the usage of FOODIE solver: effectively, the assignment between two *integrand* ADT variables is ubiquitous into the solvers implementations, see Equation (3) for example.

Listing 43. Implementation of the *oscillation integrand* assignment.

```

subroutine oscillation_assign_oscillation(lhs, rhs)
class(oscillation), intent(INOUT) :: lhs ! Left hand side.
class(integrand), intent(IN) :: rhs ! Right hand side.
select type(rhs)
class is(oscillation)
  lhs%dims = rhs%dims
  lhs%f = rhs%f
  if (allocated(rhs%U)) lhs%U = rhs%U
endselect
return
endsubroutine oscillation_assign_oscillation

```

FOODIE Numerical Integration

Using the above discussed *oscillation* type it is very easy to solve IVP (11) by means of FOODIE library. Listing 44 presents the numerical integration of system (11) by means of the Leapfrog RAW-filtered method. In the example, the integration is performed with 10^4 steps with a fixed $\Delta t = 10^2$ until the time $t = 10^6$ is reached. The example shows also that for starting a multi-step scheme such as the Leapfrog one a lower-order or equivalent order one-scheme is necessary: in the example the first 2 steps are computed by means of one-step TVD/SSP Runge-Kutta 2-stages schemes. Note that the memory registers for storing the Runge-Kutta stages and the RAW filter displacement must be handled by the client code. Listing 44 demonstrates how it is simple, clear and concise to solve a IVP by FOODIE solvers. Moreover, it proves how it is simple and effective to apply different solvers in a coupled algorithm, that greatly simplify the development of new hybrid solvers for self-adaptive time step size.

Listing 44. Numerical integration of the *oscillation* system by means of Leapfrog RAW-filtered method.

```

use foodie, only: leapfrog_integrator, tvd_runge_kutta_integrator
type(leapfrog_integrator)      :: lf_integrator ! Leapfrog integrator.
type(tvd_runge_kutta_integrator) :: rk_integrator ! Runge-Kutta integrator.
type(oscillation)              :: rk_stage(1:2) ! Runge-Kutta stages.
type(oscillation)              :: previous(1:2) ! Previous time steps solution.
type(oscillation)              :: oscillator    ! Oscillation field.
type(oscillation)              :: filter        ! Filter displacement.
integer                        :: step          ! Time steps counter.
real                          :: Dt            ! Time step.

call lf_integrator%init()
call rk_integrator%init(stages=2)
call oscillator%init(initial_state=[0.0,1.0], f=10e4, steps=2)
Dt = 100.0
do step=1, 10000
  if (2>=step) then
    call rk_integrator%integrate(U=oscillator, stage=rk_stage, Dt=Dt, t=step*Dt)
    previous(step) = oscillator
  else
    call lf_integrator%integrate(U=oscillator, previous=previous, filter=filter, Dt=Dt, &
                                t=step*Dt)
  endif
enddo
call print_results(U=oscillator)

```

4.1.3. Adams-Bashforth

Table 9 summarizes the Adams-Bashforth error analysis. As expected, the Adams-Bashforth 1 step solution, that reverts back to the explicit forward Euler one, is unstable for all the Δt exercised.

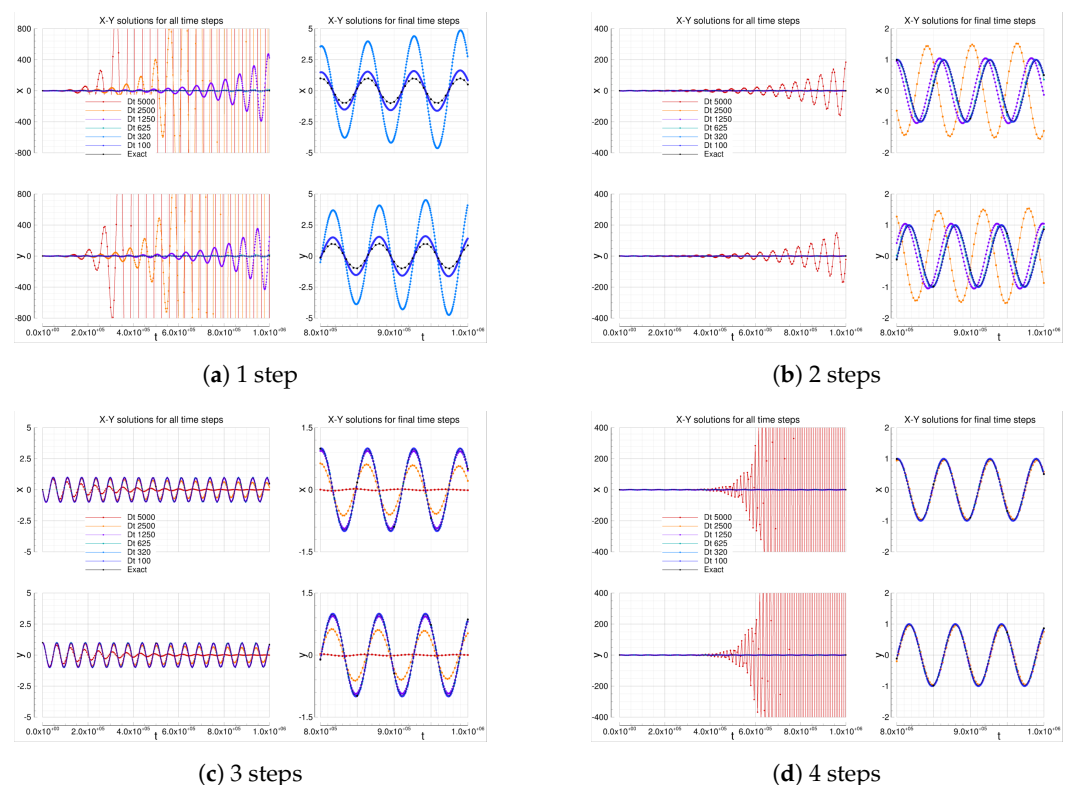
The expected observed orders of accuracy for the Adams-Bashforth solvers using 2, 3 and 4 time steps tend to 1.5, 2.5 and 3.5 that are in agreement with the expected formal order of 2, 3 and 4, respectively. Comparing the errors of the finest time resolution, i.e., $\Delta t = 100$, we find that the L2 norm decreases of the 2 orders of magnitude as the solver's accuracy increases by 1 order. This also means that fixing a tolerance on the errors, the higher is the solver's accuracy the larger is the time resolution available. As an example, assuming that admissible errors are of $O(10^{-2})$ with the 4-steps solver we can use $\Delta t = 625$ performing $N_s = t_{final}/625$ numerical integration steps, whereas using a 3-steps solvers we must adopt $\Delta t = 100$ performing $6.25 \times N_s$ numerical integration steps instead of N_s . Considering that the computational costs is only slightly affected by the number of previous time steps considered (recalling Equation (6) one can observe that there is only one new evaluation of the residuals function R independently of the previous time steps considered, thus, the computational costs is affected only by the increasing number of residuals summations, the costs of which are typically negligible with respect the cost of R evaluation.), the accuracy order has strong impact on the overall numerical efficiency: to improve the numerical efficiency reducing the computational costs, the usage of high order Adams-Bashforth solvers with larger time steps should be preferred instead of low order solvers with smaller time steps.

Table 9. Oscillation test: errors analysis of explicit Adams-Bashforth solvers.

(a) 1 Step					(b) 2 Steps				
Time Step	Error X	Error Y	Order X	Order Y	Time Step	Error X	Error Y	Order X	Order Y
5000.0	0.840×10^{10}	0.706×10^{10}	/	/	5000.0	0.596×10^3	0.583×10^3	/	/
2500.0	0.503×10^6	0.570×10^6	14.03	13.60	2500.0	0.221×10^2	0.218×10^2	4.75	4.74
1250.0	0.289×10^4	0.272×10^4	7.45	7.71	1250.0	0.764×10^1	0.769×10^1	1.53	1.50
625.0	0.239×10^3	0.232×10^3	3.59	3.55	625.0	0.265×10^1	0.268×10^1	1.53	1.52
320.0	0.737×10^2	0.722×10^2	1.76	1.74	320.0	0.968×10^0	0.981×10^0	1.51	1.50
100.0	0.250×10^2	0.247×10^2	0.93	0.92	100.0	0.169×10^0	0.171×10^0	1.50	1.50

(c) 3 Steps					(d) 4 Steps				
Time Step	Error X	Error Y	Order X	Order Y	Time Step	Error X	Error Y	Order X	Order Y
5000.0	0.857×10^1	0.854×10^1	/	/	5000.0	0.128×10^7	0.143×10^7	/	/
2500.0	0.391×10^1	0.386×10^1	1.13	1.14	2500.0	0.106×10^1	0.107×10^1	20.21	20.34
1250.0	0.825×10^0	0.814×10^0	2.24	2.25	1250.0	0.967×10^{-1}	0.981×10^{-1}	3.45	3.45
625.0	0.150×10^0	0.148×10^0	2.46	2.46	625.0	0.859×10^{-2}	0.871×10^{-2}	3.49	3.49
320.0	0.282×10^{-1}	0.278×10^{-1}	2.49	2.49	320.0	0.827×10^{-3}	0.838×10^{-3}	3.50	3.50
100.0	0.154×10^{-2}	0.152×10^{-2}	2.50	2.50	100.0	0.141×10^{-4}	0.143×10^{-4}	3.50	3.50

Figure 1 shows, for each solver exercised, the $X(t)$ and $Y(t)$ solution for $t \in [0, 10^6]$: the plots into the figure report a global overview of the solution for all the instants considered (left subplots) and a detailed zoom over the last instants of the integration (right subplots) for evaluating the numerical errors accumulation. For the sake of clarity, the strongly unstable solutions are omitted into the subplots concerning the final integration instants, namely the solutions for large Δt . Figure 1 emphasizes the instability generation for some pairs steps number/ Δt . The 2 and 4 steps solutions are instable for $\Delta t = 5000 \rightarrow f * \Delta t = 0.5$. On the contrary, the 3 steps solution is stable, but the amplitude is damped and the solution vanishes as the integration proceeds. The 2 and 4 steps solutions show a phase error that decreases as the time resolution increases, whereas 3 steps solution has null phase error.

**Figure 1.** Oscillation equations solutions computed by means of Adams-Bashforth solvers

4.1.4. Adams-Bashforth-Moulton

Table 10 summarizes the Adams-Bashforth-Moulton error analysis. The same considerations done for the Adams-Bashforth solutions can be repeated for the Adams-Bashforth-Moulton ones, thus they are omitted for the sake of conciseness. An interesting result concerns the observed errors: the $O(10^{-2})$ error is now obtained with $\Delta t = 1250$ for the 4-steps solver, thus it is 2 times faster than the corresponding Adams-Bashforth 4-step solver. Considering that the computational costs of a single Adams-Bashforth-Moulton step is only slightly greater than the corresponding Adams-Bashforth step, the efficiency increasing is not negligible.

Table 10. Oscillation test: errors analysis of predictor-corrector Adams-Bashforth-Moulton solvers.

(a) 1 Step					(b) 2 Steps				
Time Step	Error X	Error Y	Order X	Order Y	Time Step	Error X	Error Y	Order X	Order Y
5000.0	0.241×10^{20}	0.266×10^{20}	/	/	5000.0	$0.704 \times 10^{E+01}$	$0.701 \times 10^{E+01}$	/	/
2500.0	0.664×10^{11}	0.716×10^{11}	28.44	28.47	2500.0	$0.392 \times 10^{E+01}$	$0.395 \times 10^{E+01}$	0.84	0.83
1250.0	0.952×10^6	0.100×10^7	16.09	16.12	1250.0	$0.148 \times 10^{E+01}$	$0.150 \times 10^{E+01}$	1.40	1.39
625.0	0.413×10^4	0.407×10^4	7.85	7.95	625.0	$0.526 \times 10^{E+00}$	$0.534 \times 10^{E+00}$	1.49	1.49
320.0	0.387×10^3	0.383×10^3	3.54	3.53	320.0	$0.193 \times 10^{E+00}$	$0.196 \times 10^{E+00}$	1.50	1.50
100.0	0.145×10^3	0.145×10^3	0.84	0.83	100.0	$0.338 \times 10^{E-01}$	$0.342 \times 10^{E-01}$	1.50	1.50
(c) 3 Steps					(d) 4 Steps				
Time Step	Error X	Error Y	Order X	Order Y	Time Step	Error X	Error Y	Order X	Order Y
5000.0	0.457×10^1	0.464×10^1	/	/	5000.0	0.229×10^1	0.225×10^1	/	/
2500.0	0.656×10^0	0.654×10^0	2.80	2.83	2500.0	0.119×10^0	0.118×10^0	4.26	4.25
1250.0	0.100×10^0	0.987×10^{-1}	2.71	2.73	1250.0	0.825×10^{-2}	0.833×10^{-2}	3.85	3.83
625.0	0.169×10^{-1}	0.167×10^{-1}	2.56	2.56	625.0	0.671×10^{-3}	0.681×10^{-3}	3.62	3.61
320.0	0.314×10^{-2}	0.310×10^{-2}	2.52	2.51	320.0	0.631×10^{-4}	0.640×10^{-4}	3.53	3.53
100.0	0.171×10^{-3}	0.169×10^{-3}	2.50	2.50	100.0	0.107×10^{-5}	0.108×10^{-5}	3.51	3.51

Figure 2 shows similar plots of Figure 1 above discussed. Differently from the Adams-Bashforth class, the amplitude damping feature is now possessed by the 2-steps solver, see Figure 2b, while all solutions show phase errors that decrease as the time resolution increases.

4.1.5. Adams-Moulton

Table 11 summarizes the Adams-Moulton error analysis. The implicit Adams-Moulton solvers behave much like the Adams-Bashforth-Moulton ones: they have similar errors and observed orders for the same formal order considered. However, the implicit Adams-Moulton class uses one less step with respect to the corresponding Adams-Bashforth-Moulton class: this could lead to the promise of higher computational efficiency. Notwithstanding, for solving the implicit non-linearity embedded into the Adams-Moulton solvers an iterative algorithm must be employed: for the results presented, a 5 iterations of *fixed point algorithm* have been computed. This strongly reduces the eventual gain of computational efficiency with respect to the Adams-Bashforth-Moulton class.

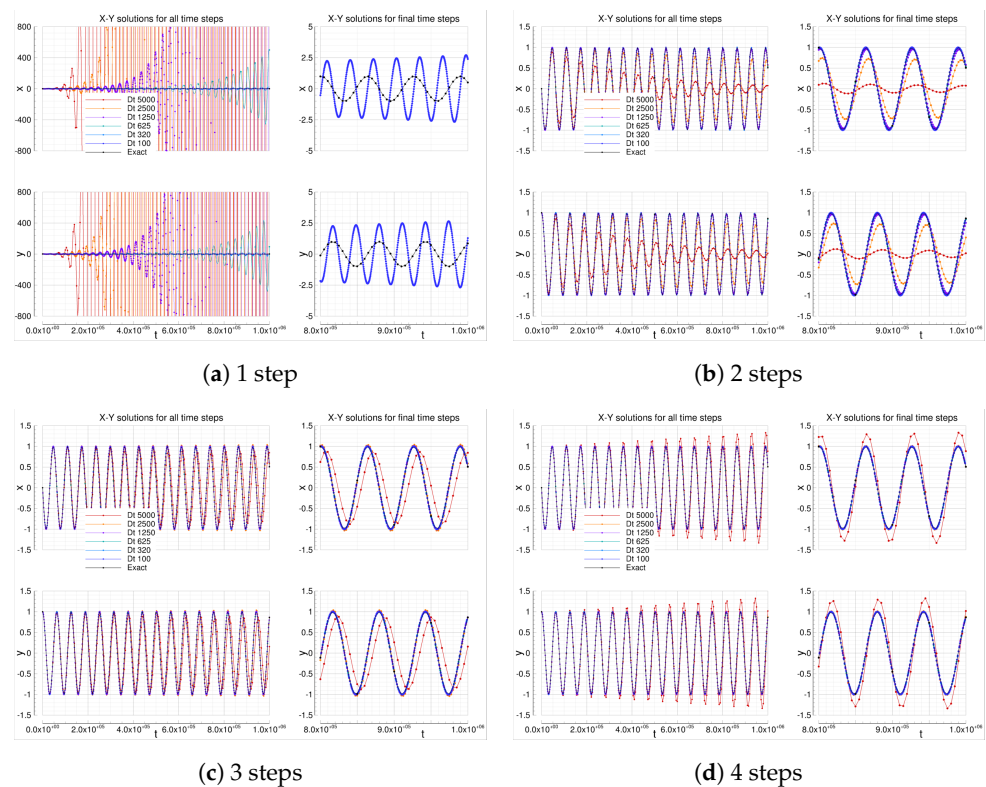


Figure 2. Oscillation equations solutions computed by means of Adams-Bashforth-Moulton solvers.

Table 11. Oscillation test: errors analysis of explicit Adams-Moulton solvers; the implicit non-linearity is solved by 5 iterations of *fixed point algorithm*.

(a) 1 Step					(b) 2 Steps				
Time Step	Error X	Error Y	Order X	Order Y	Time Step	Error X	Error Y	Order X	Order Y
5000.0	0.840×10^{10}	0.706×10^{10}	/	/	5000.0	0.108×10^2	0.109×10^2	/	/
2500.0	0.503×10^6	0.570×10^6	14.03	13.60	2500.0	0.412×10^1	0.419×10^1	1.39	1.38
1250.0	0.289×10^4	0.272×10^4	7.45	7.71	1250.0	0.148×10^1	0.150×10^1	1.48	1.48
625.0	0.239×10^3	0.232×10^3	3.59	3.55	625.0	0.527×10^0	0.533×10^0	1.49	1.49
320.0	0.737×10^2	0.722×10^2	1.76	1.74	320.0	0.193×10^0	0.196×10^0	1.50	1.50
100.0	0.250×10^2	0.247×10^2	0.93	0.92	100.0	0.338×10^{-1}	0.342×10^{-1}	1.50	1.50
5000.0	0.390×10^1	0.384×10^1	/	/	5000.0	0.983×10^0	0.999×10^0	/	/
2500.0	0.551×10^0	0.544×10^0	2.82	2.82	2500.0	0.832×10^{-1}	0.845×10^{-1}	3.56	3.56
1250.0	0.947×10^{-1}	0.934×10^{-1}	2.54	2.54	1250.0	0.736×10^{-2}	0.746×10^{-2}	3.50	3.50
625.0	0.167×10^{-1}	0.165×10^{-1}	2.50	2.50	625.0	0.652×10^{-3}	0.660×10^{-3}	3.50	3.50
320.0	0.313×10^{-2}	0.309×10^{-2}	2.50	2.50	320.0	0.626×10^{-4}	0.635×10^{-4}	3.50	3.50
100.0	0.171×10^{-3}	0.169×10^{-3}	2.50	2.50	100.0	0.107×10^{-5}	0.108×10^{-5}	3.50	3.50

Figure 3 shows similar plots of Figure 2 above discussed: there are not relevant differences between the 2 classes of solvers.

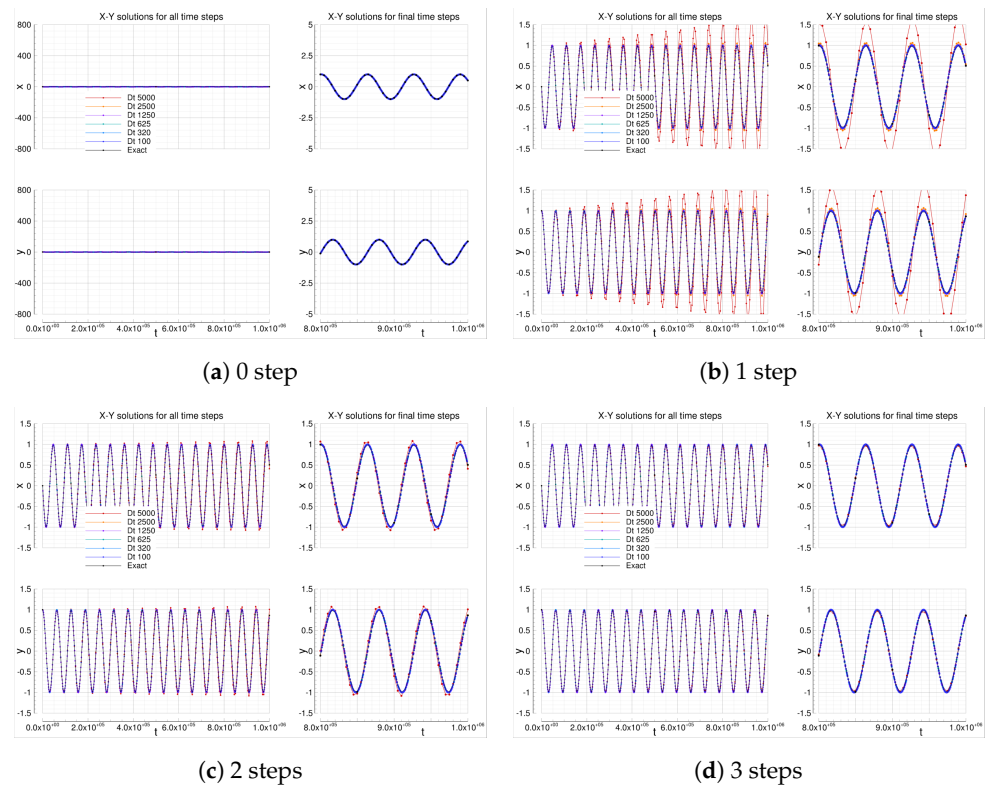


Figure 3. Oscillation equations solutions computed by means of Adams-Moulton solvers.

4.1.6. Leapfrog

The Leapfrog solutions are in agreement with the expected results: both unfiltered and RAW-filtered solutions show an observed order of accuracy that tends to the formal 2nd order, as reported in Table 12. The two solutions are almost the same, see Figure 4.

Table 12. Oscillation test: errors analysis of explicit Leapfrog solvers.

(a) Unfiltered					(b) RAW-Filtered				
Time Step	Error X	Error Y	Order X	Order Y	Time Step	Error X	Error Y	Order X	Order Y
5000.0	0.156×10^2	0.156×10^2	/	/	5000.0	0.156×10^2	0.156×10^2	/	/
2500.0	0.849×10^1	0.846×10^1	0.87	0.88	2500.0	0.855×10^1	0.852×10^1	0.86	0.87
1250.0	0.300×10^1	0.303×10^1	1.50	1.48	1250.0	0.303×10^1	0.305×10^1	1.50	1.48
625.0	0.106×10^1	0.107×10^1	1.51	1.50	625.0	0.107×10^1	0.108×10^1	1.51	1.50
320.0	0.387×10^0	0.392×10^0	1.50	1.50	320.0	0.390×10^0	0.395×10^0	1.50	1.50
100.0	0.676×10^{-1}	0.685×10^{-1}	1.50	1.50	100.0	0.685×10^{-1}	0.692×10^{-1}	1.50	1.50

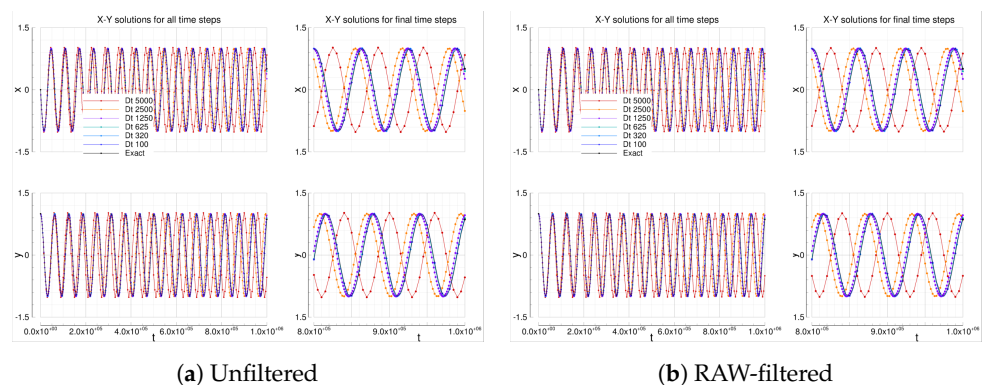


Figure 4. Oscillation equations solutions computed by means of Leapfrog solvers.

4.1.7. Low Storage Runge-Kutta

Table 13 summarizes the error analysis of Low Storage Runge-Kutta solver. The 1 stage solution, that reverts back to the explicit forward Euler one, is unstable for all the Δt exercised.

The expected observed orders of accuracy using 5, 6, 7, 12, 13 and 14 stages tend to 3.5 that are in agreement with the expected formal order 4. Comparing the errors of the finest time resolution, i.e., $\Delta t = 100$, we find that the L2 norm decreases (slowly) as the number of stages increases. Figures 5 and 6 show the solutions for all the stages tested.

Table 13. Oscillation test: errors analysis of explicit Low Storage Runge-Kutta solvers.

(a) 1 Stage					(b) 5 Stages				
Time Step	Error X	Error Y	Order X	Order Y	Time Step	Error X	Error Y	Order X	Order Y
5000.0	0.840×10^{10}	0.706×10^{10}	/	/	5000.0	0.120×10^0	0.122×10^0	/	/
2500.0	0.503×10^6	0.570×10^6	14.03	13.60	2500.0	0.106×10^{-1}	0.107×10^{-1}	3.51	3.51
1250.0	0.289×10^4	0.272×10^4	7.45	7.71	1250.0	0.935×10^{-3}	0.947×10^{-3}	3.50	3.50
625.0	0.239×10^3	0.232×10^3	3.59	3.55	625.0	0.826×10^{-4}	0.836×10^{-4}	3.50	3.50
320.0	0.737×10^2	0.722×10^2	1.76	1.74	320.0	0.793×10^{-5}	0.803×10^{-5}	3.50	3.50
100.0	0.250×10^2	0.247×10^2	0.93	0.92	100.0	0.135×10^{-6}	0.137×10^{-6}	3.50	3.50
(c) 6 Stages					(d) 7 Stages				
Time Step	Error X	Error Y	Order X	Order Y	Time Step	Error X	Error Y	Order X	Order Y
5000.0	0.979×10^{-1}	0.994×10^{-1}	/	/	5000.0	0.238×10^{-1}	0.240×10^{-1}	/	/
2500.0	0.876×10^{-2}	0.888×10^{-2}	3.48	3.48	2500.0	0.203×10^{-2}	0.205×10^{-2}	3.55	3.55
1250.0	0.776×10^{-3}	0.786×10^{-3}	3.50	3.50	1250.0	0.177×10^{-3}	0.180×10^{-3}	3.51	3.51
625.0	0.686×10^{-4}	0.695×10^{-4}	3.50	3.50	625.0	0.156×10^{-4}	0.158×10^{-4}	3.50	3.50
320.0	0.659×10^{-5}	0.667×10^{-5}	3.50	3.50	320.0	0.150×10^{-5}	0.152×10^{-5}	3.50	3.50
100.0	0.112×10^{-6}	0.114×10^{-6}	3.50	3.50	100.0	0.269×10^{-7}	0.273×10^{-7}	3.46	3.46
(e) 12 Stages					(f) 13 Stages				
Time Step	Error X	Error Y	Order X	Order Y	Time Step	Error X	Error Y	Order X	Order Y
5000.0	0.195×10^{-1}	0.198×10^{-1}	/	/	5000.0	0.795×10^{-2}	0.805×10^{-2}	/	/
2500.0	0.175×10^{-2}	0.177×10^{-2}	3.48	3.48	2500.0	0.703×10^{-3}	0.712×10^{-3}	3.50	3.50
1250.0	0.155×10^{-3}	0.157×10^{-3}	3.50	3.50	1250.0	0.621×10^{-4}	0.629×10^{-4}	3.50	3.50
625.0	0.137×10^{-4}	0.139×10^{-4}	3.50	3.50	625.0	0.549×10^{-5}	0.556×10^{-5}	3.50	3.50
320.0	0.132×10^{-5}	0.133×10^{-5}	3.50	3.50	320.0	0.527×10^{-6}	0.534×10^{-6}	3.50	3.50
100.0	0.225×10^{-7}	0.228×10^{-7}	3.50	3.50	100.0	0.899×10^{-8}	0.911×10^{-8}	3.50	3.50
(g) 14 Stages									
Time Step	Error X	Error Y	Order X	Order Y					
5000.0	0.849×10^{-2}	0.860×10^{-2}	/	/					
2500.0	0.750×10^{-3}	0.759×10^{-3}	3.50	3.50					
1250.0	0.662×10^{-4}	0.671×10^{-4}	3.50	3.50					
625.0	0.585×10^{-5}	0.593×10^{-5}	3.50	3.50					
320.0	0.562×10^{-6}	0.569×10^{-6}	3.50	3.50					
100.0	0.959×10^{-8}	0.972×10^{-8}	3.50	3.50					

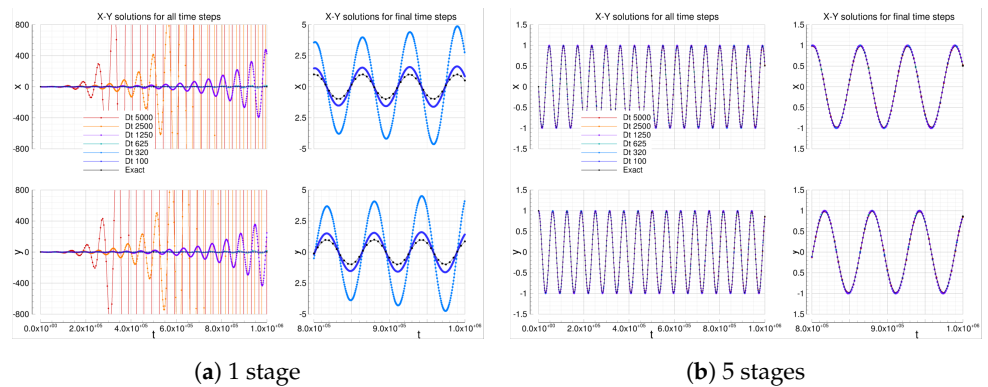


Figure 5. Oscillation equations solutions computed by means of low storage Runge-Kutta solvers with 1 and 5 stages.

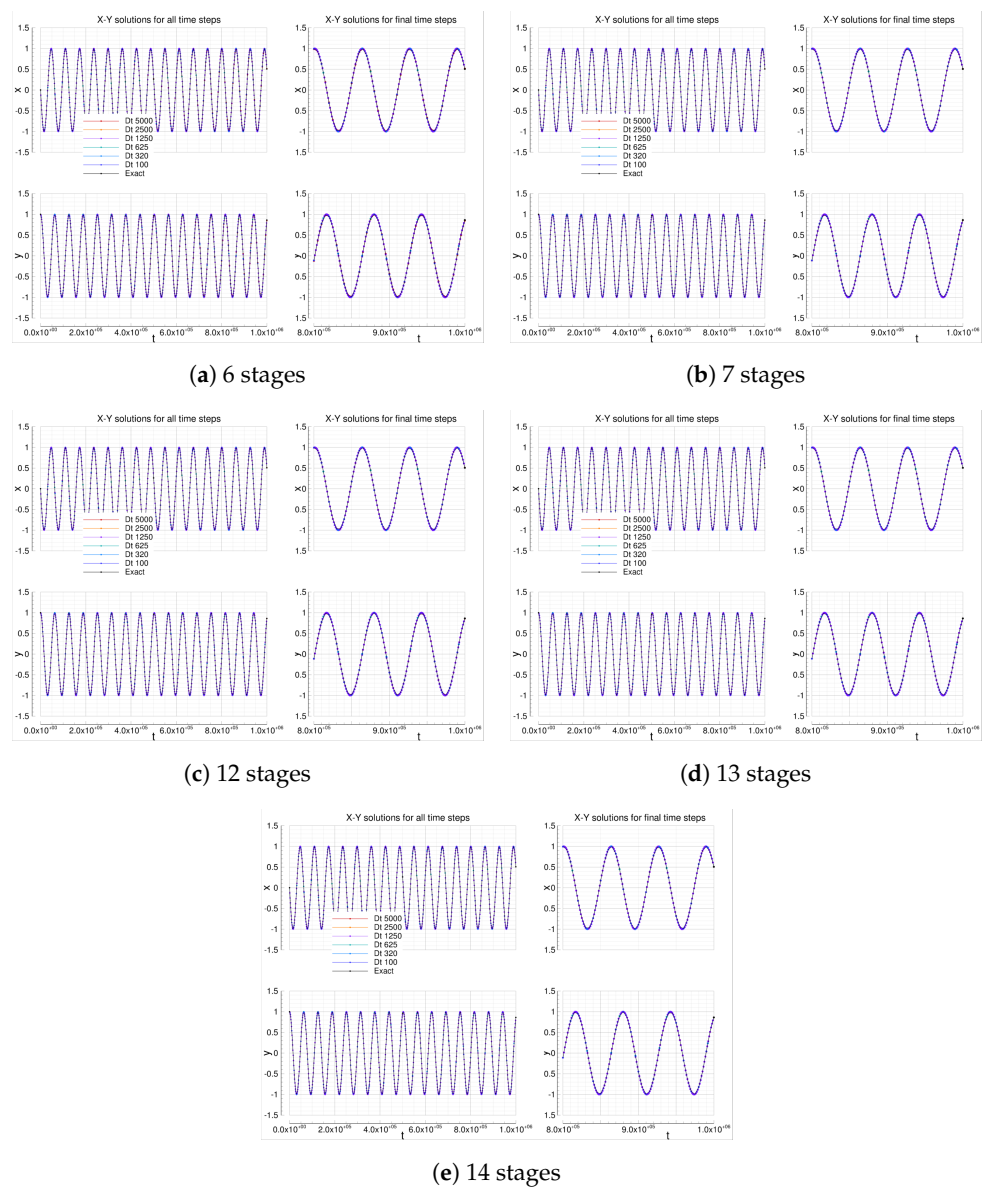


Figure 6. Oscillation equations solutions computed by means of low storage Runge-Kutta solvers with 6, 7, 12, 13 and 14 stages.

4.1.8. TVD/SSP Runge-Kutta

Table 14 summarizes the error analysis of TVD/SSP Runge-Kutta solver. The 1 stage solution, that reverts back to the explicit forward Euler one, is unstable for all the Δt exercised.

The expected observed orders of accuracy using 2, 3, and 5 stages tend to 1.5, 2.5 and 3.5 that are in agreement with the expected formal order 2, 3 and 4, respectively. Comparing the errors of the finest time resolution, i.e., $\Delta t = 100$, we find that the L2 norm decreases as the number of stages increases (roughly 2 order of magnitude for each stage). Figure 7 shows the solutions for all the stages tested.

Table 14. Oscillation test: errors analysis of explicit TVD/SSP Runge-Kutta.

(a) 1 Stage					(b) 2 Stages				
Time Step	Error X	Error Y	Order X	Order Y	Time Step	Error X	Error Y	Order X	Order Y
5000.0	0.840×10^{10}	0.706×10^{10}	/	/	5000.0	0.316×10^2	0.319×10^2	/	/
2500.0	0.503×10^6	0.570×10^6	14.03	13.60	2500.0	0.892×10^1	0.894×10^1	1.83	1.84
1250.0	0.289×10^4	0.272×10^4	7.45	7.71	1250.0	0.301×10^1	0.305×10^1	1.57	1.55
625.0	0.239×10^3	0.232×10^3	3.59	3.55	625.0	0.106×10^1	0.107×10^1	1.51	1.51
320.0	0.737×10^2	0.722×10^2	1.76	1.74	320.0	0.387×10^0	0.392×10^0	1.50	1.50
100.0	0.250×10^2	0.247×10^2	0.93	0.92	100.0	0.676×10^{-1}	0.685×10^{-1}	1.50	1.50
(c) 3 Stages					(d) 5 Stages				
Time Step	Error X	Error Y	Order X	Order Y	Time Step	Error X	Error Y	Order X	Order Y
5000.0	0.255×10^1	0.252×10^1	/	/	5000.0	0.139×10^0	0.141×10^0	/	/
2500.0	0.523×10^0	0.516×10^0	2.28	2.29	2500.0	0.122×10^{-1}	0.124×10^{-1}	3.50	3.50
1250.0	0.944×10^{-1}	0.931×10^{-1}	2.47	2.47	1250.0	0.108×10^{-2}	0.110×10^{-2}	3.50	3.50
625.0	0.167×10^{-1}	0.165×10^{-1}	2.50	2.50	625.0	0.956×10^{-4}	0.969×10^{-4}	3.50	3.50
320.0	0.314×10^{-2}	0.310×10^{-2}	2.50	2.50	320.0	0.937×10^{-5}	0.949×10^{-5}	3.47	3.47
100.0	0.171×10^{-3}	0.169×10^{-3}	2.50	2.50	100.0	0.512×10^{-6}	0.519×10^{-6}	2.50	2.50

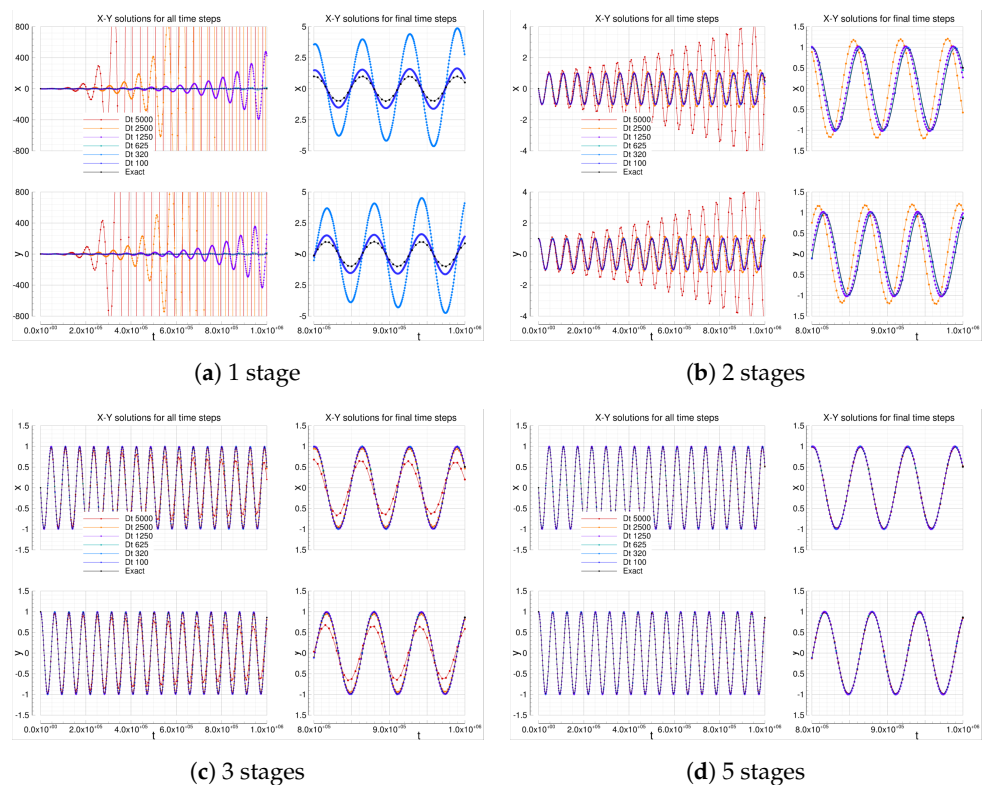


Figure 7. Oscillation equations solutions computed by means of TVD/SSP Runge-Kutta solvers.

5. Benchmarks on Parallel Frameworks

As aforementioned, FOODIE is unaware of any parallel paradigms or programming models the client codes adopt. As a consequence, the parallel performances measurements presented into this section are aimed only to prove that FOODIE environment does not destroy the parallel scaling of the baseline code implemented without FOODIE.

To obtain such a prove, the 1D Euler PDE system described previously is numerically solved with FOODIE-aware test codes that in turn exploit parallel resources by means:

- CoArray Fortran (CAF) model, for shared and distributed memory architectures;
- OpenMP directive-based model, for only shared memory architectures;

In order to measure the performances of the parallel-enabled FOODIE tests, the *strong* and *weak* scaling have been considered. For the strong scaling the *speedup* has been computed:

$$speedup(N, k) = \frac{T_{serial}(N)}{T_{parallel}(N, k)} \quad (16)$$

where N is the problem size, K the number of parallel resources used (namely the physical cores), T_{serial} is the CPU time of the serial code and $T_{parallel}$ the one of the parallel code. The ideal speedup is linear with slope equals to 1. The efficiency correlated to the strong scaling measurement is defined as:

$$efficiency(N, k) = \frac{speedup(N, k)}{k} \quad (17)$$

The maximum ideal efficiency is obviously the unity.

For the of weak scaling measurement the *sizeup* has been computed:

$$sizeup(N, k) = \frac{N_k}{N_1} \cdot \frac{T_{serial}(N_1)}{T_{parallel}(N_k, k)} \quad (18)$$

where N_1 is the minimum size considered and N_k is the size used for the test computed with k parallel resources. If N_k is scaled proportional to N_1 , the ideal sizeup is again linear and if $N_k = k \cdot N_1$ the slope is again linear. The efficiency correlated to the weak scaling is defined as:

$$efficiency(N, k) = \frac{sizeup(N, k)}{k} \quad (19)$$

The maximum ideal efficiency is obviously the unity.

The same 1D Euler PDEs problem is also solved by parallel-enabled codes that are not based on FOODIE: their solutions provide a reference for measuring the effect of FOODIE abstraction on the parallel scaling.

5.1. CAF Benchmark

This subsection reports the parallel scaling analysis of Euler 1D test programs (with and without FOODIE) being parallelized by means of CoArrays Fortran (CAF) model. This parallel model is based on the concept of *coarray* introduced into the Fortran 2008 standard: the array syntax is extended introducing the so called *codimension* that is a new arrays indexing. Essentially, a CAF enabled code is designed to be replicated a certain number of times and all copies, conventionally named *images*, are executed asynchronously. Each image has its own set of data (memory) and the codimension indexes are used to access to the (remote) memory of the other images. The CAF approach allows the implementation of Partitioned Global Address Space (PGAS) model following the SPMD (single program, multiple data) parallelization paradigm. The programmer must take care of defining the coarray variables and of synchronizing the images when necessary. This approach requires the refactoring of legacy serial codes, but it allows the exploitation of both shared and

distributed memory architectures. Moreover, it is a standard feature of Fortran (2008), thus it is not chained to any particular compiler vendors extension.

The benchmarks shown in this section have been done on a *dual Intel(R) Xeon(R) CPU X5650* exacores workstation for a total of 12 physical cores, coupled with 24 GB of RAM. In order to perform an accurate analysis 4 different codes have been considered:

- FOODIE-aware codes:
 - serial code;
 - CAF-enabled code;
- procedural codes without using FOODIE library:
 - serial code;
 - CAF-enabled code;

These codes (see Appendix A.1 for the implementation details) have been compiled by means of the GNU gfortran compiler v5.2.0 coupled with OpenCoarrays v1.1.0 (an open-source software project for developing, porting and tuning transport layers that support coarray Fortran (CAF) compilers, see <http://www.opencoarrays.org/>).

The Euler conservation laws are integrated for 30 time steps by means of the TVD RK(5,4) solver: the measured CPU time used for computing the scaling efficiencies is the average of the 30 integrations, thus representing the mean CPU time for computing one time step integration.

For the strong scaling, the benchmark has been conducted with 240,000 finite volumes. Figure 8a summarizes the strong scaling analysis: it shows that FOODIE-based code scales similarly to the baseline code without FOODIE.

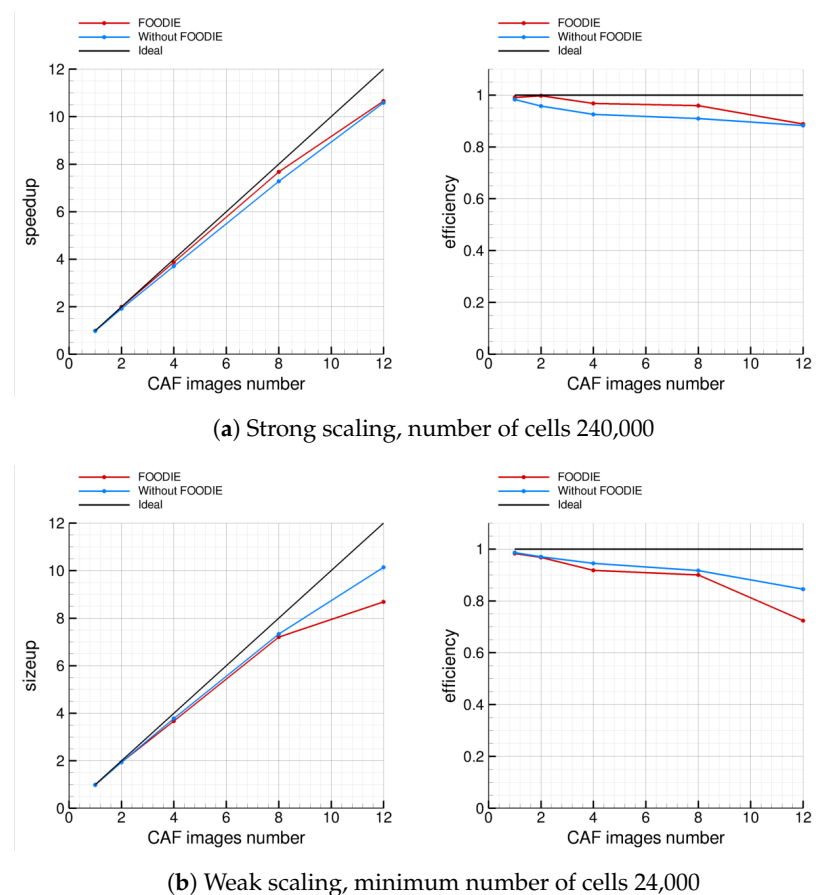


Figure 8. Scaling efficiency with CAF programming model.

For the weak scaling the minimum size is 24,000 finite volumes and the size is scaled linearly with the CAF images, thus $N_{12} = 288,000$ cells. Figure 8b summarizes the weak

scaling analysis and it essentially confirms that FOODIE-based code scales similarly to the baseline code without FOODIE.

Both strong and weak scaling analysis point out that for the computing architecture considered the parallel scaling is reasonable up to 12 cores, the efficiency being always satisfactory.

To complete the comparison, the absolute CPU-time consumed by the two families of codes (with and without FOODIE) must be considered. Table 15 summarizes the benchmarks results. As shown, procedural and FOODIE-aware codes consume a very similar CPU-time for both the strong and the weak benchmarks. The same results are shown in Figure 9. These results prove that the abstraction of FOODIE environment does not degrade the computational efficiency.

Table 15. Caf benchmarks results.

(a) Strong Benchmarks, Number of Cells 240,000					(b) Weak Benchmarks, Minimum Number of Cells 24,000					
Number of caf threads	CPU time for 1 time step integration				Number of caf threads	Number of Cells	CPU time for 1 time step integration			
	FOODIE Serial	FOODIE Parallel	Procedu- ral Serial	Procedural Parallel			FOODIE Serial	FOODIE Parallel	Procedural Serial	Procedu- ral Parallel
1	3.2970	3.3297	3.0049	3.0563	1	24,000	0.3105	0.3159	0.3089	0.3133
2	/	1.6536	/	1.5686	2	48,000	/	0.3209	/	0.3185
4	/	0.8515	/	1.8116	4	96,000	/	0.3384	/	0.3269
8	/	0.4296	/	0.4130	8	192,000	/	0.3449	/	0.3369
12	/	0.3094	/	0.2839	12	288,000	/	0.4291	/	0.3657

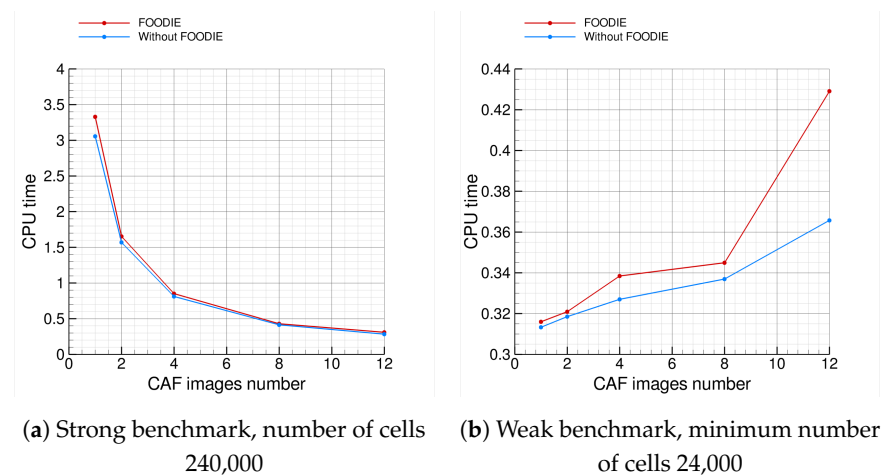


Figure 9. CPU time consumed with caf programming model.

5.2. OpenMP Benchmark

This subsection reports the parallel scaling analysis of Euler 1D test programs (with and without FOODIE) being parallelized by means of OpenMP directives-based paradigm. This parallel model is based on the concept of *threads*: an OpenMP enabled code start a single (master) threaded program and, at run-time, it is able to generate a team of (many) threads that work concurrently on the parallelized parts of the code, thus reducing the CPU time necessary for completing such parts. The parallelization is made by means of *directives* explicitly inserted by the programmer: the communications between threads are automatically handled by the compiler (through the provided OpenMP library used as back-end). OpenMP parallel paradigm is not a standard feature of Fortran, rather it is an extension provided by the compiler vendors. This parallel paradigm constitutes an effective and easy approach for parallelizing legacy serial codes, however its usage is limited to shared memory architectures because all threads must have access to the same memory.

The benchmarks shown in this section have been done on a *dual Intel(R) Xeon(R) CPU X5650* exacores workstation for a total of 12 physical cores, coupled with 24 GB of RAM. In order to perform an accurate analysis 4 different codes have considered:

- FOODIE-aware codes:
 - serial code;
 - OpenMP-enabled code;
- procedural codes without using FOODIE library:
 - serial code;
 - OpenMP-enabled code;

These codes (see Appendix A.1 for the implementation details) have been compiled by means of the GNU gfortran compiler v5.2.0 with *-O2 -fopenmp* compilation flags.

The Euler conservation laws are integrated for 30 time steps by means of the TVD RK(5,4) solver: the measured CPU time used for computing the scaling efficiencies is the average of the 30 integrations, thus representing the mean CPU time for computing one time step integration.

For the strong scaling, the benchmark has been conducted with 240,000 finite volumes. Figure 10a summarizes the strong scaling analysis: it shows that FOODIE-based code scales similarly to the baseline code without FOODIE.

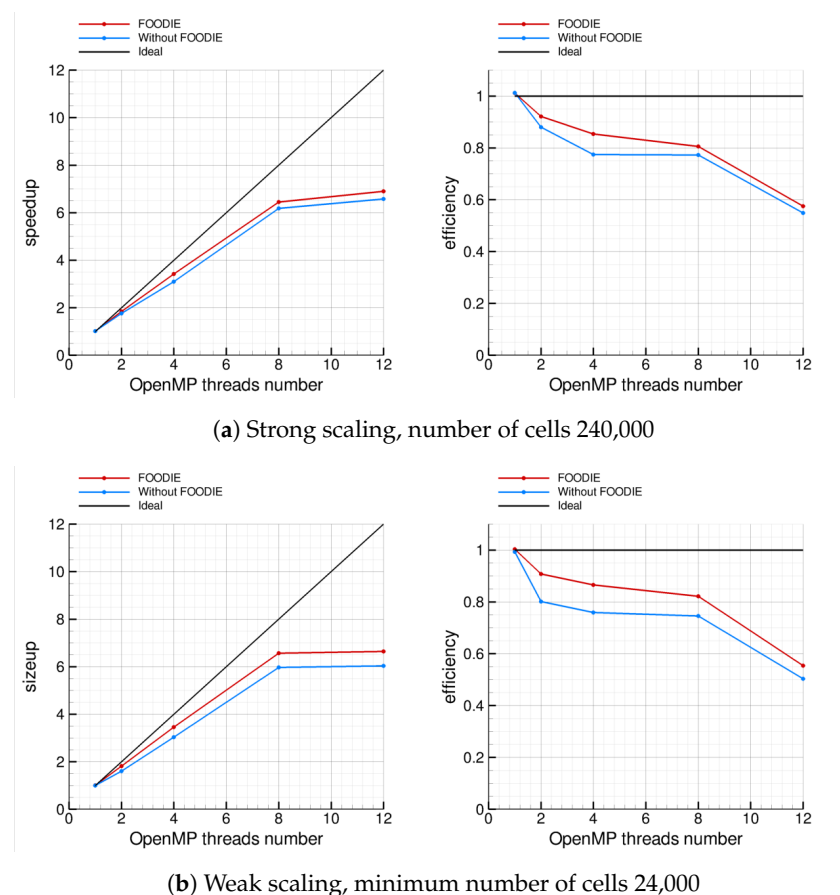


Figure 10. Scaling efficiency with OpenMP programming model.

For the weak scaling the minimum size is 24,000 finite volumes and the size is scaled linearly with the OpenMP threads, thus $N_{12} = 288,000$ cells. Figure 10b summarizes the weak scaling analysis and it essentially confirms that FOODIE-based code scales similarly to the baseline code without FOODIE.

Both strong and weak scaling analysis point out that for the computing architecture considered the parallel scaling is reasonable up to 8 cores: using 12 cores the measured efficiencies become unsatisfactory, reducing below the 60%.

To complete the comparison, the absolute CPU-time consumed by the two families of codes (with and without FOODIE) must be considered. Table 16 summarizes the benchmarks results. As shown, procedural and FOODIE-aware codes consume a very similar CPU-time for both the strong and the weak benchmarks. The same results are shown in Figure 11. These results prove that the abstraction of FOODIE environment does not degrade the computational efficiency.

Table 16. OpenMP benchmarks results.

(a) Strong Benchmarks, Number of Cells 240,000					(b) Weak Benchmarks, Minimum Number of Cells 24,000					
Number of OpenMP threads	CPU time for 1 time step integration				Number of OpenMP threads	Number of Cells	CPU time for 1 time step integration			
	FOODIE Serial	FOODIE Parallel	Procedural Serial	Procedural Parallel			FOODIE Serial	FOODIE Parallel	Procedural Serial	Procedural Parallel
1	3.3466	3.3076	3.1252	3.0873	1	24,000	0.3171	0.3162	0.3089	0.3111
2	/	1.8166	/	1.7765	2	48,000	/	0.3492	/	0.3854
4	/	0.9798	/	1.0085	4	96,000	/	0.3666	/	0.4069
8	/	0.5192	/	0.5055	8	192,000	/	0.3862	/	0.4142
12	/	0.4847	/	0.4748	12	288,000	/	0.5727	/	0.6142

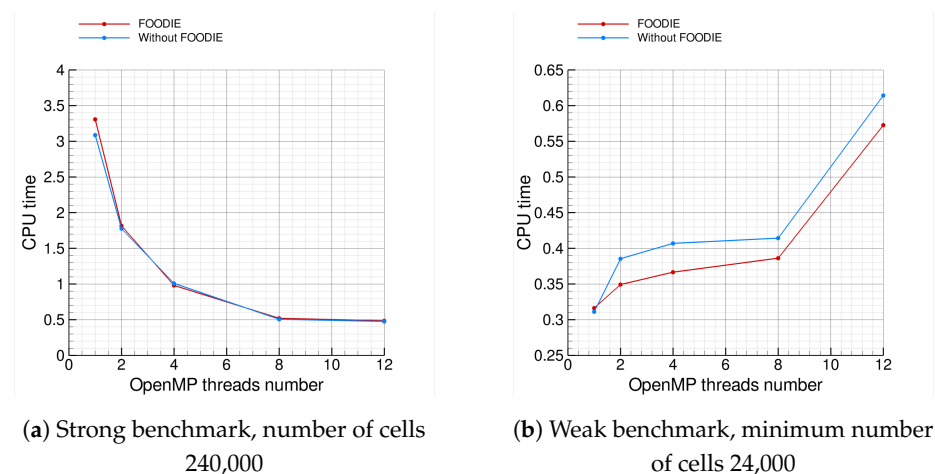


Figure 11. CPU time consumed with OpenMP programming model.

6. Concluding Remarks and Perspectives

The present manuscript provides detailed analysis of the implementation and tests of a software framework for the numerical solution of Ordinary Differential Equations (ODEs) for evolutionary (dynamic) problems. The numerical solution of general, non linear differential equations system of the form $U_t = R(t, U)$, $U_0 = F$ (where U is the vector of state variables being a function of the time-like independent variable t , $U_t = \frac{dU}{dt}$, R is the (vectorial) residual function, it could be a non linear function of the solution U itself and F is the (vectorial) initial conditions function.) is an ubiquitous mathematical representation for many dynamic phenomena. As a consequence, the development of new mathematical and numerical methods for solving ODEs is of paramount interest for mathematicians and physicists: in particular, it is crucial to minimize implementation errors, to maximize source code clearness and conciseness and to speed-up the rapid implementation of new ideas while preserving computational efficiency. Such goals are often in contrast with hard-coded ODEs solvers which implementations are often very

different from the mathematical description of the solvers themselves. As demonstrated in this work, the exploitation of Fortran Object Oriented Programming capabilities had let us to implement a very powerful ODEs solver library, FOODIE framework, that allows mathematicians and physicists to implement novel solvers in a very clear, concise and less-errors-prone than the hard-coded way due to the high abstraction level of the library itself. In particular, by means of the Abstract Calculus Pattern (ACP), FOODIE library allows to express the solvers formulae with a very high-level language, it being close as much as possible to their *natural* mathematical formulations which scientists are familiar to, i.e., the presented approach allows the implementation of novel methods with the same low-efforts of Computer Algebra System (CAS). However, differently from common CAS, the presented approach is implemented in pure Fortran programming language, allowing also for High Performance Computing (HPC) problems. As a matter of facts, the presented tests and parallel benchmarks have proved that FOODIE (and ACP approach in general) does not decrease the parallel computing efficiency.

Future works on FOODIE will concern the implementation of new ODEs solvers as well as its applications to some non linear, dynamic PDEs system, in particular concerning the Computational Fluid Dynamics (CFD) field. Current exascale superpc are currently focused on GPU accelerators: the closest perspective of FOODIE extensions will concern the exploitation of parallel GPU computing power by means of OpenMP GPU offloading.

Author Contributions: Conceptualization, implementation, validation, formal analysis, investigation, writing, supervision, S.Z.; investigation, writing, C.A. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Data Availability Statement: FOODIE library and all data related is freely available at <https://github.com/Fortran-FOSS-Programmers/FOODIE>, accessed on 20 August 2023.

Conflicts of Interest: The authors declare no conflict of interest.

Appendix A. Euler 1D Parallel Tests API

In Sections 5.1 and 5.2 it has been proved that FOODIE usage does not penalize the parallel scaling of an equivalent procedural code implemented without FOODIE. To this aim, we have solved the Euler's conservation laws (in one dimension) by means of FOODIE: as a matter of fact, Euler 1D PDEs constitutes a complex test retaining many difficulties of real applications, but it is still simple enough to serve as benchmark test. In this section we report the implementation details of the codes developed to solve (with serial and parallel models) the Euler 1D PDEs system.

Appendix A.1. Euler 1D Baseline API

The 1D Euler PDEs system is a non linear, hyperbolic (inviscid) system of conservation laws for compressible gas dynamics, that reads

$$U_t = R(U) \Leftrightarrow U_t = F(U)_x$$

$$U = \begin{bmatrix} \rho \\ \rho u \\ \rho E \end{bmatrix} \quad F(U) = \begin{bmatrix} \rho u \\ \rho u^2 + p \\ \rho u H \end{bmatrix} \quad (A1)$$

where ρ is the density, u is the velocity, p the pressure, E the total internal specific energy and H the total specific enthalpy. The PDEs system must completed with the proper initial and boundary conditions. Moreover, an ideal (thermally and calorically perfect) gas is considered

$$\begin{aligned} R &= c_p - c_v \\ \gamma &= \frac{c_p}{c_v} \\ e &= c_v T \\ h &= c_p T \end{aligned} \quad (A2)$$

where R is the gas constant, c_p and c_v are the specific heats at constant pressure and volume (respectively), e is the internal energy, h is the internal enthalpy and T^* is the temperature. The following addition equations of state hold:

$$\begin{aligned} T &= \frac{p}{\rho R} \\ E &= \rho e + \frac{1}{2} \rho u^2 \\ H &= \rho h + \frac{1}{2} \rho u^2 \\ a &= \sqrt{\frac{\gamma p}{\rho}} \end{aligned} \quad (A3)$$

An extension of the above Euler system is considered allowing the modelling of a multi-fluid mixture of different gas (with different physical characteristics). The well known Standard Thermodynamic Model is used to model the gas mixture replacing the density with the density fraction of each specie composing the mixture. This led to the following system:

$$\begin{aligned} U_t &= R(U) \Leftrightarrow U_t = F(U)_x \\ U &= \begin{bmatrix} \rho_s \\ \rho u \\ \rho E \end{bmatrix} \quad F(U) = \begin{bmatrix} \rho_s u \\ \rho u^2 + p \\ \rho u H \end{bmatrix} \quad \text{for } s = 1, 2, \dots, N_s \\ \rho &= \sum_{s=1}^{N_s} \rho_s \\ c_p &= \sum_{s=1}^{N_s} \frac{\rho_s}{\rho} c_{p,s} \quad c_v = \sum_{s=1}^{N_s} \frac{\rho_s}{\rho} c_{v,s} \end{aligned} \quad (A4)$$

where N_s is the number of initial species composing the gas mixture.

Appendix A.1.1. Memory Organization

The finite volume, Godunov's like approach is employed. Essentially, the method of Lines is used to decouple the space operator from the time one. Firstly, the space operator (the residual function of Equation (A1)) is numerically solved in order to reduce the original PDEs system to a system of ODEs that is then integrated over time by means of FOODIE solvers. Here we omit the details of the numerical models, interested readers can see [22,23]. On the contrary, some details on the memory organization is necessary to explaining the implemented API.

The conservative variables are co-located at the cell center. The cell and (inter)faces numeration is as shown in Listing A1.

cell										(inter) faces									
v										v									
	----		----		----		----		----		----		----		----		----		----
	1-Ng		2-Ng		...		-1		0		1		2		...		Ni		Ni+1
	----		----		----		----		----		----		----		----		----		----
0-Ng										-1	0	1	2			Ni-1	Ni		Ni+Ng

Listing A1. Numerical grid organization.

In Listing A1 N_i is the number of finite volumes (cells) used for discretizing the domain and N_g is the number of ghost cells used for imposing the left and right boundary conditions (for a total of $2N_g$ cells). For each cell the conservative variables must be stored: this is done by means of of rank 2 array where the first index refers to the conservative variables (densities, momentum or energy) while the second index refers to the space location, namely the cell index.

The most CPU time consuming part of a finite volume scheme is the fluxes computation across the cells interfaces. Such a computation corresponds to a loop over all the cells interfaces. Listing A2 shows a pseudo-code example of such a computation.

Listing A2. Pseudo-code example of fluxes computation.

```
do i=0, Ni
```

```

F(:, i) = compute_fluxes(U(:, i), U(:, i+1))
enddo

```

In the pseudo-code of Listing A2 it has been emphasized that the fluxes across an interface depends on the cells at left and right of the interface itself. The key point for the parallelization of such an algorithm is to compute the fluxes concurrently using as much as possible the parallel resources provided by the running architecture. As a consequence, the above showed loop over the whole domain is split into sub-domains (explicitly or implicitly accordingly to the parallel model adopted) and the fluxes of each sub-domain are computed concurrently.

Appendix A.1.2. The Integrand API

The conservative variables of 1D Euler's system can be easily implemented as a *FOODIE integrand field* defining a concrete extension of the *FOODIE integrand* type. Listing A3 reports the implementation of such an integrand field that is contained into the tests suite shipped within the *FOODIE* library.

Listing A3. Implementation of the *Euler 1D integrand* type.

```

type, extends(integrand) :: euler_1D
private
integer(I_P)                :: ord=0      ! Space accuracy formal order.
integer(I_P)                :: Ni=0       ! Space dimension.
integer(I_P)                :: Ng=0       ! Number of ghost cells for boundary conditions handling.
integer(I_P)                :: Ns=0       ! Number of initial species.
integer(I_P)                :: Nc=0       ! Number of conservative variables, Ns+2.
integer(I_P)                :: Np=0       ! Number of primitive variables, Ns+4.
real(R_P)                   :: Dx=0._R_P ! Space step.
type(weno_interpolator_upwind) :: weno    ! WENO interpolator.
real(R_P), allocatable       :: U(:, :)   ! Integrand (state) variables, whole physical domain [1:Nc,1:Ni].
real(R_P), allocatable       :: cp0(:)    ! Specific heat cp of initial species [1:Ns].
real(R_P), allocatable       :: cv0(:)    ! Specific heat cv of initial species [1:Ns].
character(:), allocatable    :: BC_L      ! Left boundary condition type.
character(:), allocatable    :: BC_R      ! Right boundary condition type.
integer(I_P)                 :: me=0       ! ID of this_image().
integer(I_P)                 :: we=0       ! Number of CAF images used.
contains
! auxiliary methods
procedure, pass(self), public :: init
procedure, pass(self), public :: destroy
procedure, pass(self), public :: output
procedure, pass(self), public :: dt => compute_dt
! ADT integrand deferred methods
procedure, pass(self), public :: t => dEuler_dt
procedure, pass(lhs), public  :: local_error => euler_local_error
procedure, pass(lhs), public  :: integrand_multiply_integrand => euler_multiply_euler
procedure, pass(lhs), public  :: integrand_multiply_real => euler_multiply_real
procedure, pass(rhs), public  :: real_multiply_integrand => real_multiply_euler
procedure, pass(lhs), public  :: add => add_euler
procedure, pass(lhs), public  :: sub => sub_euler
procedure, pass(lhs), public  :: assign_integrand => euler_assign_euler
procedure, pass(lhs), public  :: assign_real => euler_assign_real
! private methods
procedure, pass(self), private :: primitive2conservative
procedure, pass(self), private :: conservative2primitive
procedure, pass(self), private :: synchronize
procedure, pass(self), private :: impose_boundary_conditions
procedure, pass(self), private :: reconstruct_interfaces_states
procedure, pass(self), private :: riemann_solver
final
final                               :: finalize
endtype euler_1D

```

Serial, CAF enabled and OpenMP versions of Euler test share the same integrand API. In the serial version the cells fluxes are computed serially, whereas in CAF and OpenMP versions they are computed in parallel by the number of CAF images or OpenMP threads used, respectively.

References

1. Ince, E.L. *Ordinary Differential Equations*; Courier Corporation: Chelmsford, MA, USA, 1956.

2. Galán–García, J.L.; Rodríguez–Cielos, P.; Galán–García, M.Á.; Le Goff, M.; Padilla–Domínguez, Y.; Rodríguez–Padilla, P.; Atencia, I.; Aguilera–Venegas, G. SODES: Solving ordinary differential equations step by step. *J. Comput. Appl. Math.* **2023**, *428*, 115127. <https://doi.org/10.1016/j.cam.2023.115127>.
3. Owoyele, O.; Pal, P. ChemNODE: A neural ordinary differential equations framework for efficient chemical kinetic solvers. *Energy AI* **2022**, *7*, 100118. <https://doi.org/10.1016/j.egyai.2021.100118>.
4. Nagy, D.; Plavec, L.; Hegedus, F. The art of solving a large number of non-stiff, low-dimensional ordinary differential equation systems on GPUs and CPUs. *Commun. Nonlinear Sci. Numer. Simul.* **2022**, *112*, 106521. <https://doi.org/10.1016/j.cnsns.2022.106521>.
5. Nascimento, R.G.; Fricke, K.; Viana, F.A. A tutorial on solving ordinary differential equations using Python and hybrid physics-informed neural network. *Eng. Appl. Artif. Intell.* **2020**, *96*, 103996. <https://doi.org/10.1016/j.engappai.2020.103996>.
6. Fang, J.; Nadeem, M.; Habib, M.; Akgül, A. Numerical Investigation of Nonlinear Shock Wave Equations with Fractional Order in Propagating Disturbance. *Symmetry* **2022**, *14*, 1179. <https://doi.org/10.3390/sym14061179>.
7. Backus, J.W.; Beeber, R.J.; Best, S.; Goldberg, R.; Haibt, L.M.; Herrick, H.L.; Nelson, R.A.; Sayre, D.; Sheridan, P.B.; Stern, H.; et al. The FORTRAN automatic coding system. In *IRE-AIEE-ACM '57 (Western): Papers Presented at the February 26-28, 1957, Western Joint Computer Conference: Techniques for Reliability*; New York, NY, USA, published by National Joint Computer Committee NJCC 1957; pp. 188–198. <https://doi.org/10.1145/1455567.1455599>.
8. Draft International Standard for Fortran 2003. Published by the International Fortran standards committee, ISO/IEC JTC1/SC22, 2004, technical report available at the Fortran Working Group (ISO/IEC JTC1/SC22/WG5) official Web site, <https://wg5-fortran.org/N1601-N1650/N1601.pdf>, accessed August 20, 2023.
9. Rouson, D.; Xia, J.; Xu, X. *Scientific Software Design: The Object-Oriented Way*; Cambridge University Press: Cambridge, UK, 2011.
10. Butcher, J. *Numerical Methods for Ordinary Differential Equations*, 2nd ed.; Wiley: Hoboken, NJ, USA, 2008; p. 482.
11. Gottlieb, S.; Ketcheson, D.I.; Shu, C.W. High order strong stability preserving time discretizations. *J. Sci. Comput.* **2009**, *38*, 251–289.
12. Maury, J., Jr.; Segal, G. Cowell Type Numerical Integration as Applied to Satellite Orbit Computation. Technical Report, 1 April 1969, Goddard Space Flight Center, NASA. Available online: <https://ntrl.ntis.gov/NTRL/dashboard/searchResults/titleDetail/N6926703.xhtml> (accessed 20 August 2023).
13. Shu, C.W. Total-variation-diminishing time discretizations. *SIAM J. Sci. Stat. Comput.* **1988**, *9*, 1073–1084.
14. Williamson, J.H. Low-storage runge-kutta schemes. *J. Comput. Phys.* **1980**, *35*, 48–56.
15. Carpenter, M.H.; Kennedy, C.A. Fourth-order 2N-storage Runge-Kutta schemes. Technical Report, 1 June 1994, Langley Research Center Hampton, NASA, VA, United States <https://ntrs.nasa.gov/citations/19940028444> (accessed 20 August 2023).
16. Allampalli, V.; Hixon, R.; Nallasamy, M.; Sawyer, S.D. High-accuracy large-step explicit Runge–Kutta (HALE-RK) schemes for computational aeroacoustics. *J. Comput. Phys.* **2009**, *228*, 3837–3850.
17. Niegemann, J.; Diehl, R.; Busch, K. Efficient low-storage Runge–Kutta schemes with optimized stability regions. *J. Comput. Phys.* **2012**, *231*, 364–372.
18. Robert, A.J. The integration of a low order spectral form of the primitive meteorological equations. *J. Meteorol. Soc. Jpn. Ser. II* **1966**, *44*, 237–245.
19. Asselin, R. Frequency filter for time integrations. *Mon. Weather Rev.* **1972**, *100*, 487–490.
20. Williams, P.D. A Proposed Modification to the Robert–Asselin Time Filter. *Mon. Weather Rev.* **2009**, *137*, 2538–2546. <https://doi.org/10.1175/2009MWR2724.1>.
21. Williams, P.D. The RAW filter: An improvement to the Robert–Asselin filter in semi-implicit integrations. *Mon. Weather Rev.* **2011**, *139*, 1996–2007.
22. Zaghi, S.; Di Mascio, A.; Favini, B. Application of WENO-Positivity-Preserving Schemes to Highly Under-Expanded Jets. *J. Sci. Comput.* **2016**, *69*, 1033–1057. <https://doi.org/10.1007/s10915-016-0226-5>.
23. Zaghi, S. OFF, Open source Finite volume Fluid dynamics code: A free, high-order solver based on parallel, modular, object-oriented Fortran API. *Comput. Phys. Commun.* **2014**, *185*, 2151–2194. <https://doi.org/10.1016/j.cpc.2014.04.005>.

Disclaimer/Publisher’s Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.