Scientific Programming 13 (2005) 189–203
IOS Press

Dynamic Memory De-allocation in Fortran 95/2003 derived type calculus

Damian W.I. Rouson[a], Karla Morris[b] and Xiaofeng Xu[c]
[a]*US Naval Research Laboratory, 4555 Overlook Ave. SW, Washington, DC 20375, USA*
Tel.: +1 202 767 6965; Fax: +1 815 572 8203; E-mail: damian.rouson@nrl.navy.mil
[b]*Department of Mechanical Engineering, The Graduate Center of the City University of New York, 365 Fifth Avenue, New York, NY 10016, USA*
Tel.: +1 212 650 7134; Fax: +1 212 650 8013; E-mail: karla morris@hotmail.com
[c]*Department of Fire Protection Engineering, University of Maryland, College Park, MD 20742, USA*
Tel.: +1 571 215 2413; Fax: +1 301 405 9383; E-mail: xxf@umd.edu

Abstract. Abstract data types developed for computational science and engineering are frequently modeled after physical objects whose state variables must satisfy governing differential equations. Generalizing the associated algebraic and differential operators to operate on the abstract data types facilitates high-level program constructs that mimic standard mathematical notation. For non-trivial expressions, multiple object instantiations must occur to hold intermediate results during the expression's evaluation. When the dimension of each object's state space is not specified at compile-time, the programmer becomes responsible for dynamically allocating and de-allocating memory for each instantiation. With the advent of allocatable components in Fortran 2003 derived types, the potential exists for these intermediate results to occupy a substantial fraction of a program's footprint in memory. This issue becomes particularly acute at the highest levels of abstraction where coarse-grained data structures predominate. This paper proposes a set of rules for de-allocating memory that has been dynamically allocated for intermediate results in derived type calculus, while distinguishing that memory from more persistent objects. The new rules are applied to the design of a polymorphic time integrator for integrating evolution equations governing dynamical systems. Associated issues of efficiency and design robustness are discussed.

1. Introduction

A central activity in object-oriented software design involves constructing abstract data types (ADTs) appropriate for a given application domain. In computational science and engineering (CS&E), these ADTs often represent physical entities whose state variables admit a standard calculus. One typically uses this calculus to formulate differential equations that govern the system dynamics. Implementing the associated algebraic and differential operators in a high-level language facilitates semantics mimicking standard mathematical notation. As the formulas expressed become increasingly complicated, one frequently finds the need to allocate temporary storage for intermediate results. To avoid memory leaks, a strategy must be adopted for de-allocating storage once it is no longer needed. As with any task performed ubiquitously in a given application domain, one expects common idioms to arise for expressing the associated algorithms. Since the addition of dynamic memory management to Fortran lagged that of many other popular languages, there has been less time for such algorithms and idioms to be promulgated. This paper presents an attempt to fill this gap and discusses the attendant issues of efficiency and design robustness.

The design patterns to be presented have been heavily influenced by a series of papers published by Decyk, Norton and Szymanski [4–7]. They outlined techniques for object-oriented programming (OOP) in Fortran 90/95, including strategies for implementing encapsulation, information hiding, inheritance, static polymorphism and run-time polymorphism. An additional influence on the current work has been the recent text

ISSN 1058-9244/05/$17.00

on OOP in Fortran 90/95 by Akin [1]. In particular, he inspired our design of object constructors as wrappers for Fortran's intrinsic constructors. Akin also gave a particularly lucid illustration of emulating run-time polymorphism via dynamic dispatching. A third influence is the work of Leftantzi, Ray and Najm [9] in developing reacting flow simulation codes using the Common Component Architecture (CCA). In integrating stiff partial differential equations forward in time, they find it useful to separate algorithms from physics. For example, they create essentially stateless, polymorphic time integrators for marching forward more stateful modules containing physical data. The current paper attempts to resolve memory management issues encountered in the development of a polymorphic time integrator for several ADTs in Fortran 95/2003.

The Fortran 95 standard provided numerous constructs useful in creating ADTs [9,13]. These include modules, derived types and private data. ADTs can be implemented in Fortran 95 as modules that encapsulate derived types with public procedures for operating on the types' private data. Operator overloading facilitates performing arithmetic on the derived types. Generic function interfaces (module procedures) facilitate generalizing that arithmetic to a more feature-rich calculus with polymorphic integration and differentiation functions. We will refer to these functions as differential operators by analogy with the mathematical operators they implement.

When overloaded arithmetic and differential operators are strung together in a single expression, the call tree follows a pattern wherein the result of each operator of higher precedence gets passed up the call tree to an operator of lower precedence, typically terminating with a call to the assignment operator. Each operator result is an instance of the given derived type. The memory that must be allocated for each result falls into two categories: derived type components whose size can be determined at compile-time and those whose size must be determined at run-time.

The Fortran 95 standard provided only one mechanism for derived type components whose memory requirements are not known at compile-time. That mechanism was pointer components. Given the havoc pointers potentially wreak on optimizing compilers and the resulting performance penalty, this mechanism offered limited utility for the class of CS&E programs in which long loops over fine-grained data objects form the dominant activity (cf. [15]). This deficiency was recognized soon after the publication of the Fortran 95 standard, and the standards committee promised in a 1998 technical report to include allocatable components in derived types in its next standard [10]. Allocatable components are now officially a feature of Fortran 2003 [11].

Due to its importance for CS&E applications, compiler vendors began providing this feature in advance of the publication of the Fortran 2003 standard (cf. [17]), albeit without adding what Metcalfe, Reid and Cohen [13] indicate was one of the primary motivations for its inclusion: facilitating automatic de-allocation of memory allocated to hold intermediate results in derived type arithmetic. Metcalfe, Reid and Cohen point out that automatic de-allocation for pointer components is infeasible due to the difficulty of determining whether the associated memory is the target of another pointer [10]; whereas allocatable components apparently circumvent this difficulty at least in the isolated case of derived type arithmetic.

Until compiler vendors provide for automatic de-allocation, developers must supply this capability. Section 2 presents a case study on a class of CS&E applications for which dynamic de-allocation of intermediate results is critical: defining a polymorphic time integrator class for evolving dynamical systems. Section 3 proposes a bottom-up de-allocation scheme in which results are marked as temporary upon creation and freed at the immediately higher level in the call tree. A simple convention facilitates freeing all temporary storage while allowing chosen objects to persist. Section 4 discusses relevant trade-offs between execution time, memory usage and design robustness.

2. Case study: A polymorphic time integrator class

2.1. Physics

Consider a set of ADTs embodying the state and behavior of various lower-dimensional objects immersed in a moving three-dimensional (3D) fluid. Two examples of current interest include zero-dimensional point masses and one-dimensional line vortices. More specifically, we are interested in simulating clouds of water droplets being transported through turbulent combustion reactants for purposes of fire suppression. We abstract from this the idealized case of point masses immersed in a turbulent flow with buoyancy driven by temperature gradients. The motion of sufficiently small droplets is governed by Stokes' drag law:

$$\begin{aligned}
&\frac{d\mathbf{v}_{\text{droplet}}}{dt} = \frac{1}{St}[\mathbf{v}_{\text{gas}}(\mathbf{r}_{\text{droplet}}, t) - \mathbf{v}_{\text{droplet}}], \\
&t \in (0, T] \\
&\frac{d\mathbf{r}_{\text{droplet}}}{dt} \equiv \mathbf{v}_{\text{droplet}}, \quad \mathbf{r}_{\text{droplet}}(0) = \mathbf{r}_0, \\
&\mathbf{v}_{\text{droplet}}(0) = \mathbf{v}_0
\end{aligned} \tag{1}$$

where $\mathbf{r}_{\text{droplet}}$, $\mathbf{v}_{\text{droplet}}$ and St are the droplet position, velocity and Stokes number, respectively; and $\mathbf{v}_{\text{gas}}$ is the local gas velocity. The gas velocity is interpolated from velocity fields produced by a Navier-Stokes solver. The Stokes number is a dimensionless measure of a droplet's dynamic response time. The droplet state space is thus 7-dimensional, including three components each of $\mathbf{r}_{\text{droplet}}$ and $\mathbf{v}_{\text{droplet}}$ plus one scalar St.

Since we will typically simulate millions of droplets, a large number of objects are needed to represent the full problem state space, and the droplet abstraction can be considered fine-grained according to the definition of Rouson and Xiong [15]. It will prove convenient, however, to gather large collections of droplets into a "cloud" abstraction, at the heart of which lies a droplet array or linked list. At this level, the data structure is coarse-grained and each instantiation occupies substantial amounts of memory.

As mentioned above, we are also interested in tracking the motion of complicated networks, or "tangles", of quantized vortices in superfluid liquid helium (^{4}He). Below a critical transition temperature of 2.17 K, helium behaves as a two-fluid mixture of interpenetrating normal fluid and superfluid. The superfluid is characterized by vortices swirling around evacuated cores of approximately 1 Angstrom in diameter, so in the context of classical physics simulations, they can be represented quite well by curvilinear, one-dimensional objects that induce a velocity in the surrounding fluid according to the Biot-Savart law. The equation of motion for the vortex lines themselves is [16]:

$$\begin{aligned}
\frac{d\mathbf{S}}{dt} &= \mathbf{v}_s + \mathbf{v}_i + \alpha\mathbf{S}' \otimes (\mathbf{v}_n - \mathbf{v}_s - \mathbf{v}_i) \\
&\quad -\alpha'\mathbf{S}' \otimes [\mathbf{S}' \otimes (\mathbf{v}_n - \mathbf{v}_s - \mathbf{v}_i)], \\
\mathbf{S}(\xi, 0) &= \mathbf{S}_0(\xi), \quad t \in (0, T]
\end{aligned} \tag{2}$$

where $\mathbf{v}_s$ is the superfluid velocity imposed by boundary and initial conditions; $\mathbf{v}_i$ is the velocity induced by other vortex segments; $S(\xi, t)$ is the position of a point on the vortex filament; S' is the first derivative of $\mathbf{S}$ with respect to vortex filament arc-length ξ; α and α' are temperature-dependent constants; and finally $\mathbf{v}_n$ is the local velocity of the normal fluid. Neglecting point connectivity information, the vortex point state space is thus three-dimensional, comprising the three components of $\mathbf{S}$, all other variables being provided by other ADTs.

As with droplets, it proves convenient to gather large collections of vortex points into a vortex tangle ADT. Again if we desire to simulate millions of vortex points, the data structure is coarse-grained at the tangle level, and memory management will prove paramount. (See [15] for more detail on the tangle data structure design.)

In both the droplet and superfluid problems, a "Fluid" ADT provides the local velocities at the droplet and vortex core locations via interpolations on a velocity field produced by solving the Navier-Stokes equations:

$$\begin{aligned}
\frac{\partial \mathbf{u}}{\partial t} + \mathbf{u} \cdot \nabla \mathbf{u} &= -\nabla p + \frac{1}{Re} \nabla^2 \mathbf{u} + \mathbf{f}, \\
\nabla \cdot \mathbf{u} &= 0 \\
&\text{on } [0, 2\pi)^3 \times (0, T]
\end{aligned} \tag{3}$$

supplemented by the initial conditions

$$\mathbf{u}(\mathbf{x}, 0) = \mathbf{u}_0(\mathbf{x}), \quad p(\mathbf{x}, 0) = p_0$$

and periodic boundary conditions

$$\begin{aligned}
&\mathbf{u}(x_i \pm 2\pi n, t) = \mathbf{u}_0(x_i) \forall n, \\
&p(x_i \pm 2\pi n, t) = p(x_i) \forall n, \\
&i = 1, 2, 3
\end{aligned}$$

where $\mathbf{u}$ generically represents $\mathbf{v}_{\text{gas}}$ or $\mathbf{v}_n$; p is the fluid pressure; Re is the Reynolds number; and the body force, $\mathbf{f}$, generically represents buoyancy in hot gases, collective drag reactions from droplets, or collective mutual friction between normal fluid and superfluid.

Since we are interested in high-fidelity representations of fluid turbulence, we model the fluid state variables with globally smooth Fourier basis functions. This builds the desired periodic boundary conditions into our basis. We also recast Eq. (3) in a form that eliminates the pressure term while identically guaranteeing the divergence-free condition (cf. [14]). Time advancement takes place in Fourier space; whereas the nonlinear terms in Eq. (3) are computed pseudospectrally in physical space [3]. Transformation between these two representations via 3D Fast Fourier Transform (FFT) dominates the simulation's operation count. Since these transforms are most easily performed in shared memory, our simulations require gigabyte-sized arrays. Thus, our fluid representation is also coarse-grained and economy receives high priority.

2.2. Numerical algorithms

Each of the above problems requires modeling the evolution of a dynamical system governed by a coupled system of nonlinear ordinary differential equations of the form

$$\frac{d\mathbf{Y}}{dt} = \mathbf{F}(\mathbf{Y}, t), \quad \mathbf{Y}(0) = \mathbf{Y}_0, \quad t \in (0, T], \tag{4}$$

where $\mathbf{Y}$ is the global state vector, $\mathbf{Y}_0$ is the vector of initial conditions, and T is the final value of the time variable t. For droplets, $\mathbf{Y}$ contains all droplet positions, velocities and Stokes numbers. For vortex points, it contains position vectors. For a fluid, it contains 3D Fourier coefficient vectors. (Recall that time advancement for our fluids takes place in Fourier space.) For mixtures, $\mathbf{Y}$ contains all of the above.

Advancing Eq. (4) from the kth time step, t_k, to time $t_{k+1} \equiv t_k + \Delta t$ requires calculating

$$\mathbf{Y}_{k+1} = \mathbf{Y}_k + \int_{t_k}^{t_{k+1}} \mathbf{F}(\mathbf{Y}, t) dt \tag{5}$$

Each choice of quadrature scheme for the above integral yields a corresponding marching scheme. For example, choosing the rectangle rule yields the forward Euler method:

$$\mathbf{Y}_{k+1} = \mathbf{Y}_k + \mathbf{F}(\mathbf{Y}_k, t_k)\Delta t \tag{6}$$

For stiff systems, it is often useful to split the time derivative in Eq. (4) into a linear operator, L, and a nonlinear operator, N. Assuming L and N are time-invariant, we write:

$$\frac{d\mathbf{Y}}{dt} = \mathbf{F}(\mathbf{Y}, t) = L(Y) + N(Y) \tag{7}$$

For the fluid, the stiff terms lie in L. The above splitting facilitates advancing this term with an implicit scheme to circumvent the otherwise stringent stability restrictions, while N can be advanced by an explicit scheme to avoid iteration. Spalart et al. [19] proposed just such a marching algorithm, choosing trapezoidal quadrature for L and a third-order Runge-Kutta (RK3) method for N. Their method can be written in the three-step predictor-corrector form:

$$\begin{aligned}
\mathbf{Y}' &\equiv \mathbf{Y}_k + \Delta t[\alpha_1 L(\mathbf{Y}_k) + \beta_1 L(\mathbf{Y}') \\
&\quad + \gamma_1 N(\mathbf{Y}_k)] \\
\mathbf{Y}'' &\equiv \mathbf{Y}' + \Delta t[\alpha_2 L(\mathbf{Y}') + \beta_2 L(\mathbf{Y}'') \\
&\quad + \gamma_2 N(\mathbf{Y}') + \zeta_2 N(\mathbf{Y}_k)] \\
\mathbf{Y}_{k+1} &\equiv \mathbf{Y}'' + \Delta t[\alpha_3 L(\mathbf{Y}'') + \beta_3 L(\mathbf{Y}_{k+1}) \\
&\quad + \gamma_3 N(\mathbf{Y}'') + \zeta_3 N(\mathbf{Y}')]
\end{aligned} \tag{8}$$

which we write in a slightly modified form for clarity. Comparing a Taylor expansion of Eq. (7) to Eq. (8) yields 11 nonlinear constraints on the coefficient vectors $\boldsymbol{\alpha}, \boldsymbol{\beta}, \boldsymbol{\gamma}$ and $\boldsymbol{\zeta}$ that must be satisfied to achieve the desired order of accuracy. Our coefficient values were provided by Alan Wray of NASA Ames Research Center (private communication). In what follows, we will explore the dynamic memory de-allocation requirements of both the explicit Euler and the Wray RK3 scheme.

2.3. Software architecture

Figure 1 depicts a typical class diagram using the Unified Modeling Language (UML) architectural description standard [2]. At the top of the diagram is a representation of our `Integrand` class. This essentially stateless, polymorphic class is implemented as a `MODULE` containing the derived type `Integrand`, whose only components are pointers to instances of the derived types to be integrated. The relationship between the `Integrand` class and each of its pointer targets is represented as UML association.

Just below the `Integrand` class in Fig. 1 are three coarse-grained ADTs: `Cloud`, `Mixture`, and `Fluid`. The first and last of these are built up from two more fine-grained ADTs. A `Cloud` contains a very fine-grained array of `Droplets`; whereas a `Fluid` contains an array of three `Fields`, each of which contains one component of the fluid velocity vector field. Grouping fine-grained instances of an ADT into a more coarse-grained collection ADT facilitates changing the relationship between each instance. For example, an alternative implementation of a `Cloud` might create a linked list by adding pointer components to the `Droplets`. Alternatively, without affecting the `Droplet` definition, one might insert individual `Droplet` instances into linked list ADT as described by Akin [1]. Although the linked list abstraction would be somewhat artificial for the `Droplet` dispersion simulations, it proves quite natural in our quantum vortex tangle simulations. In the latter, the corresponding class diagram is analogous to that of Fig. 1 with a `Tangle` class replacing the `Cloud` class and a `Vortex Point` class replacing the `Droplet` class.

Although the `Cloud/Droplet` and `Tangle/Vortex Point` relationships are not inheritance hierarchies in the strictest sense, they share some of the properties of such hierarchies. Decyk et al. [5, 7], express inheritance relationships in Fortran 90/95

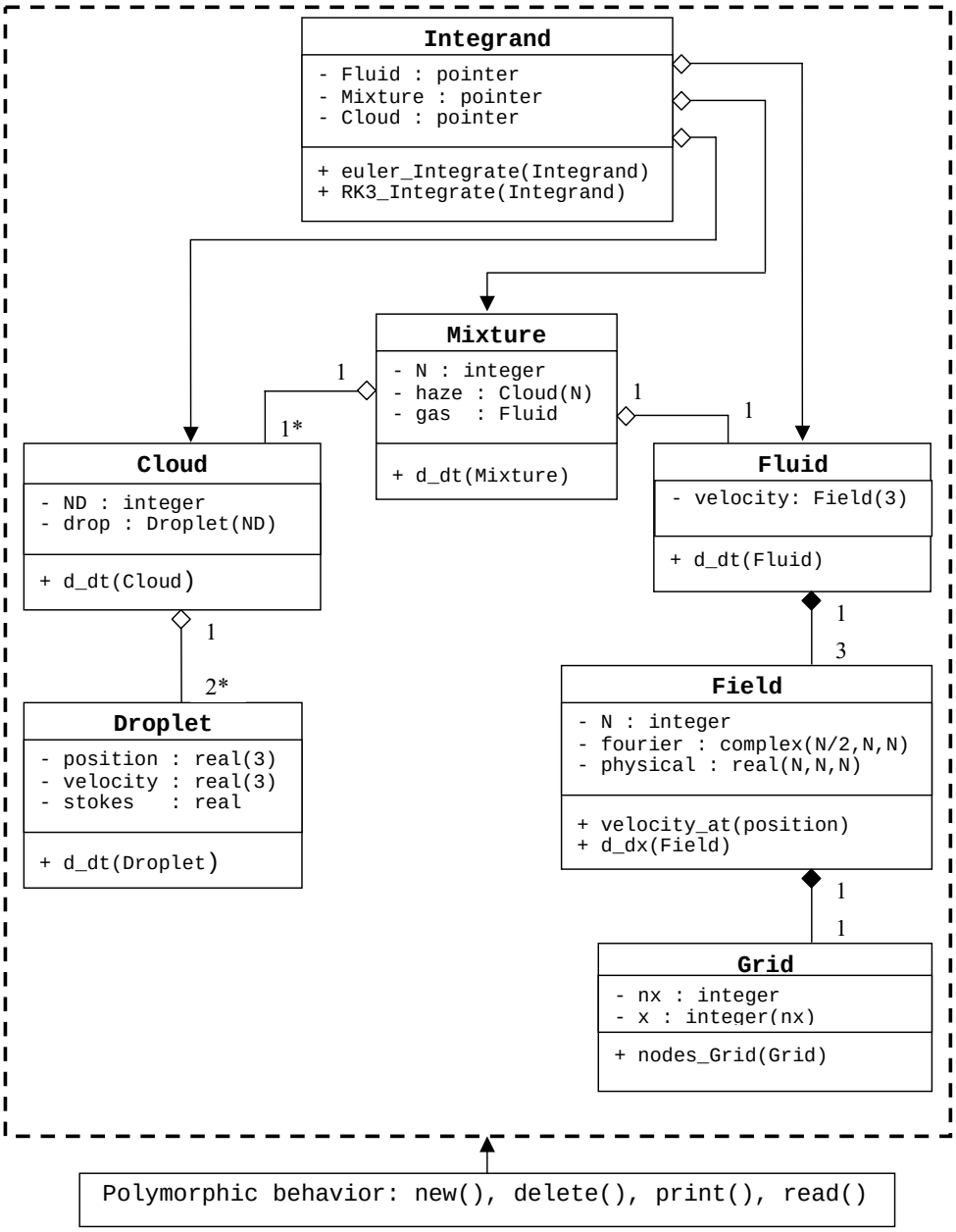


Fig. 1. Class diagram for droplet-laden Navier-Stokes solver with polymorphic time integration.

by including exactly one copy of the base class inside the subclass and delegating operations on the subclass to the base class via calls to homonymous methods in the base class. Unlike an inheritance relationship, `Clouds` and `Tangles` contain multiple instances of their related classes, but like an inheritance relationship, many operations on these coarse-grained classes are delegated to operations on each instance of their fine-grained counterparts. Each relationship connector in Fig. 1 is labeled according to the multiplicity of the relationship. Thus, a `Cloud` contains two or more `Droplets`, but each `Droplet` is associated with only one `Cloud`. Finally, open connectors indicate composition, i.e. classes composed of quantities that could exist on their own. Closed connectors indicate aggregation relationships, i.e. classes aggregating quantities that would not be useful independently.

The `Fluid` class on the right side of Fig. 1 aggre-

gates three instances of the `Field` class. As mentioned above, our use of globally smooth basis functions makes it less natural to decompose the fluid state into the sort of fine-grained data structures that might be more typical of, say, a finite element code. Hence, the Fluid could be implemented as a traditional inheritance hierarchy as defined in [5,7] – that is, the three component `Fields` could be aggregated into a single `Vector Field` abstraction, exactly one copy of which would be stored in a `Fluid`. Since inheritance relationships are frequently described as "is a" relationships, we could then state that a `Fluid` is a `Vector Field` that satisfies the Navier-Stokes equations. For our purposes, however, the `Vector Field` class would be too simplistic to justify the additional layer of abstraction. Furthermore, the relationships would be more complicated because `Fluids` would still need direct access to individual `Fields` for storing scalar quantities such as temperature.

Note that a `Field` abstracts a purely mathematical construct. Following a design pattern similar to Lefantzi et al. [12], we have separated physics from data. The `Fluid` function `d dt()` calculates the right-hand side of the Navier-Stokes equations using the differentiation function `d dx()` provided by its `Fields`. The differentiation details and the associated grid and basis functions are not exposed to the `Fluid`. Neither are the equations being solved exposed to the `Fields`. This design enhances the reusability of both modules. One could reuse the `Fluid` class, while replacing the Fourier representation in `Field` with a finite element method. Likewise, one could reuse the `Field` class, while replacing the Navier-Stokes equations in `Fluid` with an acoustic wave equation.

As shown in Fig. 1, another service a `Field` object provide its `Fluid` is the interpolation algorithm `velocity at(position)`, which returns a 3D velocity vector at the location of a passed 3D `position` vector. This yields $\mathbf{v}_{\rm gas}$ at the droplet positions in Eq. (1) and $\mathbf{v}_n$ at vortex point locations in Eq. (2). Finally, note that all state variables are private as indicated by the "$-$" symbols; whereas public methods are designated by "$+$"; and all classes implement the polymorphic constructor `new()`, destructor `delete()`, and input/output procedures `print()` and `read()`.

2.4. Derived type calculus

We can now specify the requirements for an ADT calculus using Fortran 95/2003 derived types. We seek to develop a polymorphic time integrator class for approximating the solution to Eq. (5) using the algorithms in Eqs (6) and (8). Such a class must be agnostic with respect to the private implementation details described in the previous section. The class must be able to operate on any physical object in the architecture of Fig. 1. (Here we distinguish between "physical" objects for which there exists an evolution equation and purely mathematical abstractions such as the `Field` class.) For example, it must be possible to integrate a `Cloud` object without knowing that a `Cloud` is composed of `Droplets`, nor whether those `Droplets` are organized into an array or a linked list.

We further require the syntax of the resulting derived type calculus to map naturally from the mathematical notation of Eqs (4)–(8). For example, we expect that after a Cloud instantiation of the form

```
TYPE(Cloud) :: silver_lined
CALL new(silver_lined)
```

one could access the explicit Euler method in the proposed Integrand class as follows:

```
DO k=1,num_steps
  CALL euler_Integrator(silver_lined,
  dt)
END DO
```

resulting in a call to a subroutine of the form

```
SUBROUTINE euler_Integrator(this,dt)
  ...
  this = this + dt*d_dt(this)
  ...
END SUBROUTINE euler_Integrator
```

where "`this`" is pseudocode for a generic pointer to an object of any one of several desired classes. Since no such generic pointers exist in Fortran 95, the actual code differs in ways that will be discussed below. Note that, following Decyk, Norton and Szymanski [5], our procedures mimic C++ by making the first argument in all ADT methods an object named "`this`" whose type is the derived type defined in the same module.

Successful compilation of the above code requires overloading the $+$, * and $=$ operators and implementing the differentiation function `d dt()`. More significantly, it also requires run-time polymorphism. Decyk, Norton and Szymanski [7] first outlined a strategy for expressing run-time polymorphism via dynamic dispatching in Fortran 90/95. Our `Integrand` class implementation follows an example of their technique from Akin [1] as explained next.

An `Integrand` must be instantiated using the `Integrand` constructor as follows:

```
USE Cloud_Class
USE Integrand_Class
TYPE(Cloud), TARGET  :: purple_haze
TYPE(Integrand)      :: kernel
INTEGER ,PARAMETER   :: num_drops=1.
REAL    ,PARAMETER   :: St=1.
purple_haze = Cloud_(num_drops,St)
kernel = Integrand_(purple_haze)
```

whereupon `purple haze` can now be marched forward in time by repeated calls to `euler Integrator(kernel)` or `RK3 Integrator(kernel)`. Above we have made use of the constructor convention proposed by Akin [1], whereby the `Integrand ()` constructor calls the Fortran 90/95 default constructor `Integrand()`, the latter being inaccessible from outside the `Cloud MODULE` due to the privacy of the `Integrand` data members (see [1] for more detail). Ultimately, the results can be output by a call to `print(purple haze)`, after which the results can later be recovered by a call to `read(purple haze)`.

Appendix A shows an excerpt of the code with complete details for the explicit Euler and RK3 integration of `Fluid` and `Cloud` objects. Note that `Integrand` instantiations take place through assignment procedures that guarantee that at any given time, all but one of the pointers points to `NULL()` as suggested by Decyk et al. [7] and Akin [1]. The `euler Integrator()` and `RK3 Integrator()` methods use this fact to determine what type of object is being passed. An `IF-THEN-ELSE` construct then dispatches the appropriate code for the given object. The code blocks inside each clause are semantically equivalent, with the only differences being the class of object being manipulated.

Note that in Fortran 2003, it will likely be much easier to implement similar functionality using so-called "polymorphic entities", i.e. objects whose actual type might vary during program execution. However, commercial compiler vendors have only recently provided full Fortran 95 compliance (cf. [17,18]), and compiler projects in the public domain have not even reached that point (cf. [20,21]). It is likely there will continue to be a need for strategies to emulate this technology in Fortran 95 for the next several years. Regardless, the memory de-allocation issues discussed next exist in either version of the language – as they likely would in any language without garbage collection.

Without loss of generality, it will suffice to focus on `Cloud` objects for the remainder of this paper. At the heart of the `Cloud` clause in `euler Integrator()` is the code

```
this%cloud_ptr = this%cloud_ptr
  + dt*d_dt(this%cloud_ptr)
```

Execution of the above line results in a calling sequence in which the result of the differentiation function `d dt()` is passed to an overloaded multiplication operator, the result of which is passed to an overloaded addition operator, the result of which is passed to an overloaded assignment operator. A typical interface and function signature for one of these calls might take the form

```
INTERFACE operator(*)
  MODULE PROCEDURE scalar_times_
  Cloud
END INTERFACE

FUNCTION scalar_times_Cloud(scalar,
factor)RESULT(product)
  REAL        ,INTENT(IN) :: scalar
  TYPE(Cloud) ,INTENT(IN) :: factor
  TYPE(Cloud)             :: product
```

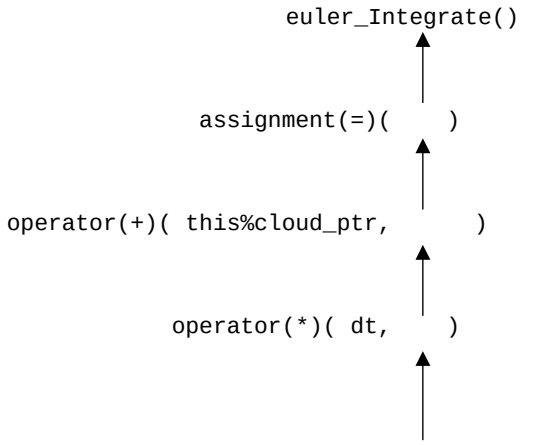
where `scalar times Cloud()` must handle any memory allocations required to store `product`. If, for example, the Cloud ADT contains an `ALLOCATABLE` array component called "`drop`" holding a collection of `Droplets`, then `scalar times Cloud()` will contain a line of the form

```
ALLOCATE( product%drop(SIZE(factor%
drop)) )
```

To avoid memory leaks, each such allocation must have a corresponding de-allocation. This issue arises four times in the call tree of the above explicit Euler expression (see Fig. 2). It arises many more times during a typical RK3 time step (see Fig. 3). The question to be addressed in the next section is where best to accomplish the requisite de-allocations.

3. Memory de-allocation rules

The software developer's vantage point frames our central dilemma. Looking from the top of the call tree down, it is clear to the designer of `euler Integrator()` which results in the explicit Euler formula can be deleted immediately after their first use. However, by the time the ultimate result has been computed, copied into the object on the left of the assignment operator, and control has been returned to `euler Integrator()`, all variable names associated with intermediate results have gone out of




```
this%cloud_ptr = this%cloud_ptr + dt*d_dt(this%cloud_ptr)
```

Fig. 2. Call tree for a typical euler Integrator() statement: tree (top), statement (bottom).

scope. Thus, the program name space contains neither `ALLOCATABLE` arrays nor `POINTER`s on which the `DEALLOCATE` intrinsic can operate to free the desired memory.

Alternatively, looking from the middle of the call tree down, it may not be clear to the designer of an overloaded arithmetic or differential operator whether the arguments passed up to it are temporary or persistent. Deleting all arguments would waste clock cycles if some arguments need to be recomputed later. For example, the RK3 code for a `Cloud` is

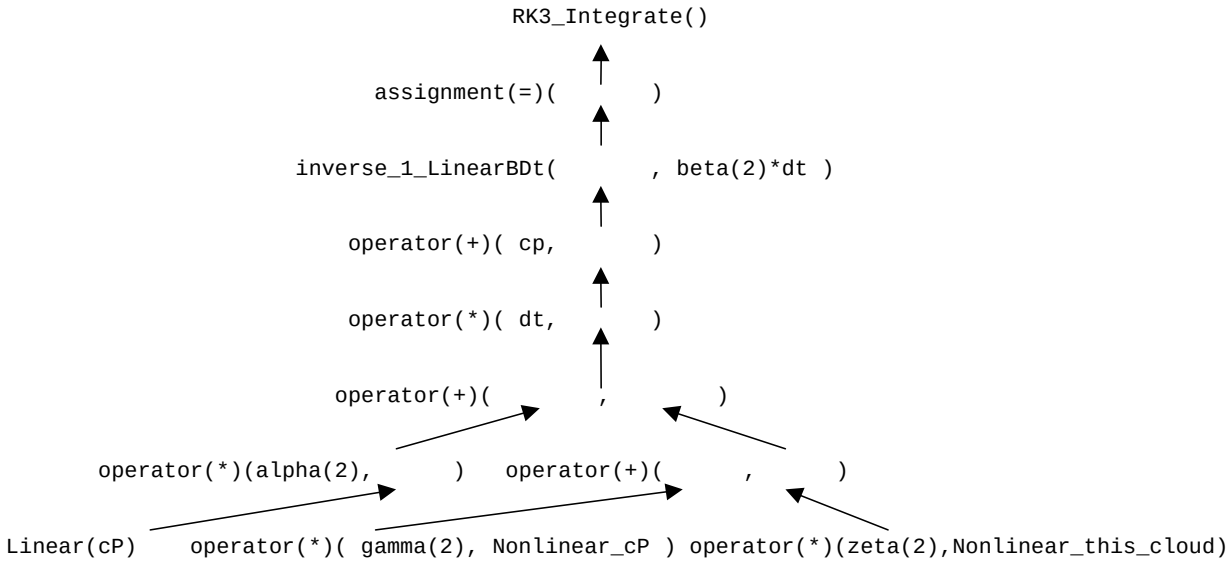
```
TYPE(Cloud):: cP, cPP, Nonlinear_this
_cloud,Nonlinear_cP

Nonlinear_this_cloud = Nonlinear
(this%cloud_ptr)
cP  = inverse_1_LinearBDt( this%
cloud_ptr+ &
  dt*( alpha(1)*Linear(this%cloud
  _ptr) +  &
    gamma(1)*Nonlinear_this_cloud ),
    beta(1)*dt &
  )
Nonlinear_cP = Nonlinear(cP)
  cPP = inverse_1_LinearBDt( cP  +
  &
         dt*(alpha(2)*Linear(cP) +
         gamma(1)*Nonlinear_cP &
          + zeta(2)*Nonlinear_this
          _cloud ),beta(2)*dt  &
       )
this%cloud_ptr =              &
  inverse_1_LinearBDt( cPP +  &
    dt*( alpha(3)*Linear(cPP) +
    gamma(2)*Nonlinear(cPP)   &
       + zeta(3)*Nonlinear_cP),
       beta(3)*dt   &
  )
```

where `cP` and `cPP` are analogous to $\mathbf{Y}'$ and $\mathbf{Y}''$ in Eq. (8); `Nonlinear this cloud` and `Nonlinear cP` are intermediate results; and `inverse 1 Linear BDt()` represents the inverse operator $(1 - \beta_i \Delta t L)^{-1}$ and takes a right-hand side from Eq. (8) as its first argument along with $\beta_i \Delta t$ as its second argument for RK3 substep i. Note that each of the above assignments represents an implicit instantiation, including any requisite dynamic memory allocations under programmer control.

In tallying persistent results in the above code, a very simple pattern emerges: all objects on the left side of an assignment operator are used more than once after assignment. Clearly erroneous results would obtain were one to assume all operator arguments are temporary and can therefore be deleted after their first use.

It is worth mentioning here that Fortran 95 requires an arithmetic `OPERATOR` to be a Fortran `FUNCTION` taking one or two non-optional arguments declared with `INTENT(IN)`, which precludes alteration of the arguments inside the procedure. Note that there is no need to alter something that will not be used subse-




```
cPP = inverse_1_LinearBDt(                                &
                cP  + dt*( alpha(2)*Linear(cP ) +         &
                           gamma(2)*Nonlinear_cP +        &
                            zeta(2)*Nonlinear_this_cloud &
                          ),beta(2)*dt                    &
                               )
```

Fig. 3. Call tree for a typical RK3 Integrator() statement: tree (left), statement (right).

quently. By contrast, a Fortran 95 `ASSIGNMENT` operator must be a SUBROUTINE taking two non-optional arguments, the first declared with `INTENT(OUT)` or `INTENT(INOUT)` and the second with `INTENT(IN)`. Again, the only reason to modify an argument is if that modification will somehow be used subsequently. Thus, modification implies persistence.

The latter reasoning suggests a solution to the de-allocation dilemma from the bottom-up vantage point. Starting at the bottom of the call tree, one marks each object as temporary or persistent at the point of creation before it is passed up the call tree. Temporary objects are deleted immediately after their first use. To codify this approach, we introduce a simple definition and its corollary:

> *Definition:* Given an object that appears as the result of an arithmetic or differential operator, we define the object as *temporary* if it can be deleted at the termination of execution of the first subsequent operator in which it appears as an argument.
>
> *Corollary:* All objects that are not temporary are *persistent*.

Four simple rules comprise the proposed memory de-allocation scheme:

> *Rule 1:* All results of arithmetic and differential operator `FUNCTION`s are marked as temporary upon creation.
> *Rule 2:* Left-hand arguments to assignment operator `SUBROUTINE`s are marked as persistent.
> *Rule 3:* Temporary objects are deleted prior to the termination of any arithmetic or differential operator in which they appear as an argument.
> *Rule 4:* Persistent objects are deleted prior to the termination of the procedure that instantiated them.

It might be worth noting the above definition and corollary are the reverse of how the terms “temporary” and “persistent” are defined in [2] for UML. There, an object that survives beyond the procedure that creates it is termed “persistent”. An object deleted by the proce-

dure that instantiates it is termed "temporary". Again the issue is vantage point. Our definition seems more natural from the standpoint of the arithmetic operator designer, who would otherwise be required to delete "persistent" objects upon first use, while the designer of the overall time-advancement algorithm might use "temporary" objects multiple times before deletion.

To facilitate imposition of the above rules, we store information in each ADT about its persistence. The binary nature of persistence naturally lends itself to representation by a Fortran LOGICAL variable. For example, our Cloud derived type definition is

```
TYPE Cloud
  PRIVATE
    TYPE(Droplet) ,DIMENSION(:),
    ALLOCATABLE :: drop
    LOGICAL :: temporary
END TYPE Cloud
```

We can therefore mark an operator result as temporary immediately subsequent to its instantiation or direct memory allocation inside the operator. Returning to the discussion of the scalar times Cloud() operator at the end of the previous section, we would simply include the following line immediately after the ALLOCATE statement presented near the end of the previous section:

```
product%temporary = .TRUE.
```

At the end of scalar times Cloud(), we would release the memory associated with any temporary arguments before their variable names go out of scope by executing a line of the form

```
IF (factor%temporary) CALL delete
(factor)
```

We place similar statements in all arithmetic and differential operators; whereas in the assignment operator, we write

```
left%temporary = .FALSE.
```

just after the corresponding ALLOCATE. Then, just before termination of the assignment operator, we write

```
IF (right%temporary) CALL delete
(right)
```

We tested the above strategy in several of the classes described in Fig. 1. Our tests indicate that the memory usage of simulations with constant-sized objects remains constant over time.

One important caveat relates to syntax. We have found considerable variation amongst compilers with regards to one's ability to DEALLOCATE allocatable components of derived type arguments declared with INTENT(IN). Some compilers allow the DEALLOCATE command and perform it successfully for sufficiently small sets of objects. Other compilers allow it, but appear not to perform it. Still other compilers disallow it. To ensure portability, we currently define one MODULE PROCEDURE to replace each desired operator. This circumvents the requirement to declare INTENT. Since the resulting expressions are somewhat less readable, we always insert an adjacent comment with the syntax used throughout this paper. For example, we write

```
!this%cloud_ptr= this%cloud_ptr
+ dt*d_dt(this%cloud_ptr)
CALL assign( this%cloud_ptr, &
     plus( this%cloud_ptr , &
        times( dt, d_dt(this%cloud_
        ptr) &
  )  )  )
```

where the corresponding MODULE PROCEDUREs are

```
! INTERFACE assignment(=)
  INTERFACE assign
     MODULE PROCEDURE assign_Cloud
  END INTERFACE assign
! END INTERFACE assignment(=)
! INTERFACE operator(+)
  INTERFACE plus
     MODULE PROCEDURE plus_Cloud
  END INTERFACE plus
! END INTERFACE operator(+)
```

and likewise for the times operator. We hope the standards committee will eventually relax the INTENT requirement on operator arguments, in which case we can rapidly revise our code by replacing the above executable lines with the corresponding commented lines.

4. Efficiency and robustness

We make no claim regarding the uniqueness of our proposed rule set. However, it appears to offer a reasonable trade-off between execution time and memory utilization. In this regard, considering two alternatives might be instructive. A bottom-up alternative would be to simply de-allocate all intermediate results, marking none as persistent. When coupled with judicious overwriting of results that can be cheaply recalculated, it might be possible to reduce the memory require-

ments at the expense of increased execution time. A top-down alternative might be to daisy-chain objects together into a linked list as they percolate up the call tree. When coupled with a mechanism for enumerating and labeling the nature of the data contained in each object, one could then recursively tunnel through the list, deleting all non-persistent data and obviating any recalculations at the expense of increased memory and additional logic for constructing the linked list.

At some level, designs based on the two latter strategies would exhibit greater robustness. The bottom-up alternative would require slightly simpler logic, eliminating the `temporary` marker. The top-down alternative would likewise eliminate this marker and gather all the de-allocation decisions into one place: the `Integrator` module, which as previously noted is the place where it is most obvious to the software engineer whether a particular object is temporary or persistent. By contrast, distributing the de-allocation decisions across all operators on all ADTs requires coordination and compliance in the development of each ADT. Verifying compliance would require exposing implementation details to a greater level than might be desirable in a team development effort. All of this implies that the best solution would be for compiler vendors to provide an automatic de-allocation capability at least for overloaded arithmetic operators.

We have investigated the capabilities of several compilers. Currently, the Fortran 95 compiler from ST Micro Portland Group [17] and Intel [18] do not perform any automatic de-allocations for intermediate results in derived type arithmetic. The Numerical Algorithms Group Fortran 95-to-C translator does perform the de-allocations, but only at the top of the call tree [22]. As mentioned, this solution is costly for coarse-grained data structures. Finally, two public domain Fortran 95 compiler projects do not appear to be far enough along to compile our code [20,21].

5. Conclusions

We have proposed a set of rules for de-allocating memory associated with temporary intermediate results of arithmetic and differential operators on abstract data types, while preserving objects desired to be persistent. Constructs have been presented for using such operators to write Fortran expressions that naturally mimic standard mathematical notation. We applied these constructs to the development of a polymorphic class for integrating equations that govern evolving dynamical systems.

As complicated expressions are evaluated, our rules work from the bottom of the call tree upwards, marking each intermediate result as temporary or persistent immediately after instantiation. Each temporary object is deleted at the immediately lower level of operator precedence (higher level on the call tree) just before the final name associated with the object goes out of scope. Doing so avoids memory leaks that would be particularly costly at the highest levels of abstraction, where coarse-grained data structures predominate. We tested the rules to ensure that simulations with constant-sized objects exhibit constant memory utilization over time. We also proposed a workaround for compilers that do not allow de-allocation of allocatable components of objects declared with `INTENT(IN)`.

Although the resulting algorithm is not unique, comparisons to alternative bottom-up and top-down approaches indicate it represents a reasonable trade-off between execution time and storage requirements. A minor compromise in robustness stems from of the required coordination and compliance by the developer of each operator for each abstract data type. Nonetheless, we believe the proposed rules fill an important need in the absence of automatic de-allocation by compilers and makes more efficient use of memory than automatic de-allocation at the top of the call tree.

Acknowledgements

This work was supported in part by Award No. 0206152 from the National Science Foundation.

Appendix A. Polymorphic time integration class

```
MODULE Integrand_Class

  USE Cloud_Class
  USE Fluid_Class

  IMPLICIT NONE

  PRIVATE                     ! Default for procedures & data
  PUBLIC :: Integrand        ! Derived type
  PUBLIC :: Integrand_        ! Constructor
  PUBLIC :: euler_Integrator ! explicit Euler
  PUBLIC :: RK3_Integrator   ! 3rd-order Runge-Kutta

  TYPE Integrand
    PRIVATE
    TYPE(Cloud)  , POINTER  :: cloud_ptr
    TYPE(Fluid)  , POINTER  :: fluid_ptr
  END TYPE Integrand

  INTERFACE Integrand_
    MODULE PROCEDURE assign_Fluid
    MODULE PROCEDURE assign_Cloud
  END INTERFACE

  CONTAINS

    FUNCTION assign_Fluid(fluid_Integrand) RESULT(family)
      TYPE(Integrand)                    :: family
      TYPE(Fluid), TARGET, INTENT(IN)   :: fluid_Integrand
      CALL nullify_Integrand(family)
      family%fluid_ptr => fluid_Integrand
    END FUNCTION assign_Fluid

    FUNCTION assign_Cloud(cloud_Integrand) RESULT(family)
      TYPE(Integrand)                  :: family
      TYPE(Cloud), TARGET, INTENT(IN) :: cloud_Integrand
      CALL nullify_Integrand(family)
      family%cloud_ptr => cloud_Integrand
    END FUNCTION assign_Cloud

    SUBROUTINE nullify_Integrand(family)
      TYPE(Integrand), INTENT(OUT)    :: family
      NULLIFY(family%fluid_ptr)
      NULLIFY(family%cloud_ptr)
    END SUBROUTINE nullify_Integrand

    ! ______  Explicit Euler time advancement _____________
    ! Each integrand class must provide a d_dt() function
    ! taking one instance of the class as its sole argument
```

```
! and returning another instance of the same class
! holding the time derivative of the argument.

SUBROUTINE euler_Integrator(this,dt)
  TYPE(Integrand), INTENT(INOUT) :: this
  REAL                           :: dt

  IF      (ASSOCIATED(this%fluid_ptr)  ) THEN

    this%fluid_ptr=this%fluid_ptr +dt*d_dt(this%fluid_ptr)

   ELSE IF (ASSOCIATED(this%cloud_ptr)  ) THEN

    this%cloud_ptr=this%cloud_ptr +dt*d_dt(this%cloud_ptr)

  END IF
END SUBROUTINE euler_Integrator

! _______ 3rd-order Runge-Kutta time advancement _________
! This integator uses the algorithm of Spalart, Moser &
! Rogers [19] with constants specified by Wray (private
! comm.).  Each integrand must provide functions Linear()
! and Nonlinear() to calculate the corresponding part of
! the differential operator, each taking as their sole
! argument one instance of the class and returning
! another instance of the same class containing its
! derivative. In addition, the inverse of
! 1-beta(i)*dt*Linear() must be provided, taking an
! instance of the integrand class as its 1st argument
! and beta(i)*dt as its 2nd argument for substep i.

SUBROUTINE RK3_Integrator(this,dt)
  IMPLICIT NONE
  TYPE(Integrand), INTENT(INOUT) :: this
  REAL           , INTENT(INOUT) :: dt
  TYPE(Droplet):: fP,fPP,Nonlinear_this_fluid,Nonlinear_fP
  TYPE(Cloud) :: cP,cPP,Nonlinear_this_cloud,Nonlinear_cP
  INTEGER,PARAMETER ::         NSTEPS=3
  REAL   ,PARAMETER, DIMENSION(NSTEPS) :: &
     alpha = (/ 4./15.,  1./15.,  1./6.  /) &
    ,beta  = (/ 4./15.,  1./15.,  1./6.  /) &
    ,gamma = (/ 8./15.,  5./12.,  3./4.  /) &
    ,zeta  = (/  0.  , -17./60., -5./12. /)

  IF      (ASSOCIATED(this%fluid_ptr)  ) THEN

    Nonlinear_this_fluid = Nonlinear(this%fluid_ptr)
    fP  = inverse_1_LinearBDt(                      &
            this%fluid_ptr +                         &
            dt*( alpha(1)*Linear(this%fluid_ptr) + &
                 gamma(1)*Nonlinear_this_fluid     &
```

```
                ),beta(1)*dt                              &
            )
        Nonlinear_fP = Nonlinear(fP)
        fPP = inverse_1_LinearBDt( &
                fP  + dt*( alpha(2)*Linear(fP ) +         &
                           gamma(2)*Nonlinear_fP +        &
                            zeta(2)*Nonlinear_this_fluid &
                          ),beta(2)*dt                     &
            )
        this%fluid_ptr = inverse_1_LinearBDt(         &
                fPP + dt*( alpha(3)*Linear(fPP) +     &
                           gamma(3)*Nonlinear(fPP) + &
                            zeta(3)*Nonlinear_fP     &
                          ),beta(3)*dt                &
            )

        CALL delete(fPP)
        CALL delete(Nonlinear_fP)
        CALL delete(fP)
        CALL delete(Nonlinear_this_fluid )

      ELSE IF (ASSOCIATED(this%cloud_ptr)  ) THEN

        Nonlinear_this_cloud = Nonlinear(this%cloud_ptr)
        cP  = inverse_1_LinearBDt(                       &
                this%cloud_ptr +                          &
                dt*( alpha(1)*Linear(this%cloud_ptr) + &
                     gamma(1)*Nonlinear_this_cloud     &
                   ),beta(1)*dt                           &
            )
        Nonlinear_cP = Nonlinear(cP)
        cPP = inverse_1_LinearBDt( &
                cP  + dt*( alpha(2)*Linear(cP ) +         &
                           gamma(2)*Nonlinear_cP +        &
                            zeta(2)*Nonlinear_this_cloud &
                          ),beta(2)*dt                     &
            )
        this%cloud_ptr = inverse_1_LinearBDt(         &
                cPP + dt*( alpha(3)*Linear(cPP) +     &
                           gamma(3)*Nonlinear(cPP) + &
                            zeta(3)*Nonlinear_cP     &
                          ),beta(3)*dt                &
            )

        CALL delete(cPP)
        CALL delete(Nonlinear_cP)
        CALL delete(cP)
        CALL delete(Nonlinear_this_cloud )

      END IF
    END SUBROUTINE RK3_Integrator
END MODULE Integrand_Class
```

References

[1] E. Akin, *Object-oriented Programming via Fortran 90/95*, Cambridge University Press, Great Britain, 2003.

[2] S.S. Alhir, *UML in a Nutshell*, O'Reilly Media, Inc., 1998.

[3] C. Canuto, M.Y. Hussaini, A. Quateroni and T.A. Zhang, *Spectral Methods in Fluid Dynamics*, Springer-Verlag, Berlin, Germany, 1988.

[4] V.K. Decyk, C.D. Norton and B.K. Szymanski, Expressing object-oriented concepts in Fortran 90, *ACM SIGPLAN Fortran Forum* **15** (1997), 13–18.

[5] V.K. Decyk, C.D. Norton and B.K. Szymanski, How to express C++ concepts in Fortran 90, *Scientific Programming* **6** (1997), 363–390.

[6] V.K. Decyk, C.D. Norton and B.K. Szymanski, High performance object-oriented programming in Fortran 90, in: *Proc. Eighth SIAM Conference on Parallel Processing for Scientific Computing March 14–17, 1997*, M. Heath et al., eds, Minnesota, 1997.

[7] V.K. Decyk, C.D. Norton and B.K. Szymanski, How to support inheritance and run-time polymorphism in Fortran 90, *Computer Physics Communications* **115** (1998), 9–17.

[8] J3 Fortran Standards Technical Committee, *ISO/IEC 1539: 1991 (E) Fortran 90 – Documentation*, International Organization for Standards/International Electrotechnical Committee, Geneva, Switzerland, 1991.

[9] J3 Fortran Standards Technical Committee, Technical Report ISO/IEC 1539-1:1997 International Organization for Standards/International Electrotechnical Committee, Geneva, Switzerland, 1997, (cited in [13]).

[10] J3 Fortran Standards Technical Committee, Technical Report ISO/IEC 15581:1998(E) International Organization for Standards/International Electrotechnical Committee, Geneva, Switzerland, 1998, (cited in [13]).

[11] J3 Fortran Standards Technical Committee, *ISO/IEC 1539 Working Draft*, International Organization for Standards/International Electrotechnical Committee, Geneva, Switzerland, 2001, (cited in [13]).

[12] S. Lefantzi, J. Ray, C. Kennedy and H. Najm, A component-based toolkit for reacting flow with high order spatial discretizations on structured adaptively refined meshes, *Progress in Computational Fluid Dynamics: An International Journal* (to appear in 2005).

[13] M. Metcalf, J. Reid and M. Cohen, *Fortran 95/2003 Explained*, Oxford University Press, Oxford, 2004.

[14] D.W.I. Rouson and J.K. Eaton, On the preferential concentration of solid particles in a turbulent channel flow, *Journal of Fluid Mechanics* **428** (2001), 149–159.

[15] D.W.I. Rouson and Y. Xiong, Design metrics in quantum turbulence simulations: How physics influences software architecture, *Scientific Programming* **12** (2004), 185–196.

[16] R.J. Donnelly, *Quantized Vortices in Helium II*, Cambridge University Press, 1991.

[17] ST Micro Portland Group, *PGI IA-32 Compilers and Tools Documentation*, (http://www.pgroup.com).

[18] Intel, Inc. Intel Fortran Compiler 8.1 for Linux, (http://www.intel.com/software/products/compilers/flin).

[19] P.R. Spalart, R.D. Moser and M.M. Rogers, Spectral methods for the Navier-Stokes equations with one infinite and two periodic directions, *Journal of Computational Physics* **6** (1991), 297–324.

[20] G95 Forgran project (http://www.g95.org).

[21] GNU Fortran 95 project (gfortran), http://gcc.gnu.org/fortran.

[22] Numerical Algorithms Group (NAG) Fortran 95 compiler, http://www.nag.com/nagware/NP.asp.