GENASIS Basics: Object-oriented utilitarian functionality for large-scale physics simulations

Christian Y. Cardall^{a,b,*}, Reuben D. Budiardja^c

^a*Physics Division, Oak Ridge National Laboratory, Oak Ridge, TN 37831-6354, USA*

^b*Department of Physics and Astronomy, University of Tennessee, Knoxville, TN 37996-1200, USA*

^c*National Institute for Computational Sciences, University of Tennessee, Knoxville, TN 37996, USA*

Abstract

Aside from numerical algorithms and problem setup, large-scale physics simulations on distributed-memory supercomputers require more basic utilitarian functionality, such as physical units and constants; display to the screen or standard output device; message passing; I/O to disk; and runtime parameter management and usage statistics. Here we describe and make available Fortran 2003 classes furnishing extensible object-oriented implementations of this sort of rudimentary functionality, along with individual ‘unit test’ programs and larger example problems demonstrating their use. These classes compose the **Basics** division of our developing astrophysics simulation code GENASIS (*General Astrophysical Simulation System*), but their fundamental nature makes them useful for physics simulations in many fields.

Keywords: Simulation framework; Object-oriented programming; Fortran 2003

PROGRAM SUMMARY

Manuscript Title: GENASIS Basics: Object-oriented utilitarian functionality for large-scale physics simulations

Authors: Christian Y. Cardall and Reuben D. Budiardja

Program Title: SineWaveAdvection, SawtoothWaveAdvection, and RiemannProb-

*Corresponding author.

E-mail addresses: cardallcy@ornl.gov (C.Y. Cardall), reubendb@utk.edu (R.D. Budiardja).

Preprint submitted to Computer Physics Communications

June 1, 2015

lem (fluid dynamics example problems illustrating GENASIS Basics); ArgonEquilibrium and ClusterFormation (molecular dynamics example problems illustrating GENASIS Basics)
Journal Reference:
Catalogue identifier:
Licensing provisions: Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International
Programming language: Fortran 2003 (tested with gfortran 4.9.2, Intel Fortran 15, NAG Fortan 5.3.1, Cray Compiler 8.2.5)
Computer: PC, cluster, supercomputer
Operating system: Linux, Unix
RAM: For example problems, depends on user-specified problem size and number of processes. The fluid dynamics problems with 128^3 cells on 8 processes use about 300 MB per process. The molecular dynamics problems with 6912 particles on 12 processes use about 20 MB per process.
Number of processors used: User-specified
Keywords: Simulation framework; Object-oriented programming; Fortran 2003
Classification: 4.14, 6.5, 20
External routines/libraries: MPI [1] and Silo [2]
Nature of problem: By way of illustrating GENASIS Basics functionality, solve example fluid dynamics and molecular dynamics problems.
Solution method: For fluid dynamics examples, finite-volume. For molecular dynamics examples, leapfrog and velocity-Verlet integration.
Unusual features: The example problems named above are not ends in themselves, but serve to illustrate our object-oriented approach and the functionality available through GENASIS Basics. In addition to these more substantial examples, we provide individual unit test programs for each of the classes comprised by GENASIS Basics.
Additional comments: A version of the GENASIS Basics source code is available from the CPC program library with this publication, and minor revisions will be maintained at <http://astro.phys.utk.edu/activities/genasis>.
Running time: For example problems, depends on user-specified problem size and number of processes. The fluid dynamics problems with 128^3 cells on 8 processes take about ten minutes of wall clock time on a Cray XC30. The molecular dynamics problems with 6912 particles for 10000 time steps on 12 processes take a little over an hour on a Cray XC30.

References

- [1] <http://www.mcs.anl.gov/mpi/>
- [2] <https://wci.llnl.gov/simulation/computer-codes/silo>

1. Introduction

The size, scope, and realism of simulations of physical systems continue to grow with increases in computing power, but not without new challenges for developers of simulation codes. Increases in computing capability seem only to come with such additional burdens as distributed memory and distributed (and, more recently, heterogeneous) processing capacity. From the perspective of working physicists, it can seem that the physics itself recedes ever further away as their efforts are channeled towards developing and tailoring codes to the specific features of these increasingly complex high-performance machines. In this environment, the availability of well-designed codes with broadly applicable physics capabilities is increasingly valuable to researchers.

Even physicists working directly on code development prefer to focus on numerical algorithms, problem setup and testing, shepherding production runs to completion, and data analysis and interpretation; but large-scale physics simulations on distributed-memory supercomputers require more basic utilitarian functionality as well. Examples include handling of physical units and constants; display to the screen or standard output device; message passing; I/O to disk; and runtime parameter management and usage statistics. A distributed-memory message passing environment complicates many of these tasks. While necessary for production simulations, this sort of functionality is not central to the physics problem to be simulated, and therefore is often treated in an ad-hoc way as an afterthought or annoying nuisance—sometimes with inconsistent results and time-consuming consequences.

Like other developers of physics simulation codes, we have had to implement such utilitarian functionality in our code GENASIS (*General Astrophysical Simulation System*), a new code under development for the simulation of core-collapse supernovae on the world’s leading capability supercomputers. Initial capabilities of GENASIS for a refinable mesh and hydrodynamics from an astrophysics perspective, with emphasis on test results rather than code features, have been reported elsewhere [1]. ‘General’ denotes the capacity of the code to include and refer to multiple algorithms,

solvers, and physics and numerics choices with the same abstracted names and/or interfaces. In GENASIS this is accomplished with features of Fortran 2003 that support the object-oriented programming paradigm (e.g. [2]). ‘Astrophysical’ roughly suggests—over-broadly, at least initially—the types of systems at which the code is aimed, and the kinds of physics and solvers it will include. ‘Simulation System’ indicates that the code is not a single program, but a collection of modules, organized as classes, that can be invoked by a suitable driver program set up to characterize and initialize a particular problem. While we are initially and primarily developing and using GENASIS for astrophysics problems, our goal is to address even the most utilitarian parts of the code with enough care and generality to hopefully make them useful for physics simulations in other fields.

The object-oriented approach we apply to even the basic utilitarian parts of the code facilitates the generality, extensibility, and maintainability that potentially make portions of GENASIS useful beyond our initial target application. Object-oriented programming involves several interrelated principles. *Abstraction* identifies the major concepts required by a program, without specifying the details of implementation. *Encapsulation* bundles together and controls access to the data (or *members*) and actions (or *methods*) associated with a particular concept into a self-contained unit—a *class*. (The class itself is just a kind of template or glorified type specification; an *instance* of the class, basically a variable declared with that type specification, is called an *object*. In terms of an analogy with intrinsic data types, a class is to an object as the intrinsic data type `real` is to a variable declared as `real`.) Done well, abstraction and encapsulation lead to *decoupling*—the separation of code into building blocks that are as independent and reusable as possible. Reusability is further enhanced by *polymorphism*, which enables multiple versions or implementations of the same basic concept to be referred to and used interchangeably. Closely related is *inheritance*, which allows new (*child*) classes to be formed from prior (*parent*) classes: existing members and methods are retained or modified, and new members and methods can be added.

The high-level structure of the core of GENASIS is sketched in Fig. 1. Solid lines outline relationships in the source code directory hierarchy, and dashed arrows indicate compilation dependencies. **Modules** and **Programs** are the two highest-level divisions shown. **Modules** comprises the classes forming the central functionality of GENASIS. Each class is defined in a single Fortran **module**. Fortran **programs**—drivers that invoke these classes, and begin the execution of some particular computational task—are collected

Figure 1: High-level structure of the core of GENASIS. Solid lines outline the directory hierarchy, and dashed arrows indicate compilation dependencies.

under **Programs**.

Modules, and the divisions **UnitTests** and **Examples** under **Programs**, all contain divisions labeled **Physics**, **Mathematics**, and **Basics**. On the **Modules** side, these categories contain class implementations. Dual to these under **UnitTests** on the **Programs** side, in almost complete one-to-one correspondence, are ‘unit tests’ that exercise the capabilities and provide example usage of the individual classes. The programs in the **Physics**, **Mathematics**, and **Basics** divisions under **Examples** integrate the use of many classes in their respective divisions. For instance, the example problems discussed in Sec. 2 are found in the **Basics** division of **Examples**. As indicated by the dashed dependency arrows, the unit tests and integrative examples first depend upon their corresponding classes: **Basics** examples and unit tests depend on **Basics** classes; **Mathematics** examples and unit tests depend on **Mathematics** classes, and through these also on **Basics** classes; and so on.

Programs also has another division—**Applications**—intended for drivers aimed at purposes beyond unit tests and integrative examples, ranging from the solution of simple physical test problems to the execution of production-scale multiphysics research simulations. These ultimately depend on all **Modules**—**Physics**, **Mathematics**, and **Basics**.

Figure 2: *Left:* Structure of **Basics**. *Middle Left:* Substructure of **FileSystem**. *Middle Right:* Substructure of **MessagePassing**. *Right:* Substructure of **VariableManagement**. *All:* Solid lines outline the directory hierarchy. Boxes framed with thinner linewidths denote ‘leaf’ divisions of the code with no additional subdirectories. The compilation order is from bottom to top; thus dependencies essentially flow in reverse, from top to bottom.

Our main purpose in this paper is to describe and make available the **Basics** division of GENASIS, which contains some utilitarian functionality for large-scale simulations on distributed-memory supercomputers. (For the place of **Basics** in the overall scheme of GENASIS, refer again to Fig. 1.) Its content, as illustrated in the left diagram of Fig. 2, includes the divisions **VariableManagement**, **Display**, **MessagePassing**, **FileSystem**, and **Runtime**. Two of these—**Display** and **Runtime**—are framed with boxes of thinner linewidth; these are ‘leaf’ divisions in the sense that they contain no further subdirectories, but only individual files defining classes. The right diagram in Fig. 2 shows the structure within **VariableManagement**, which includes the leaf divisions **Specifiers**, **ArrayOperations**, **ArrayArrays**, and **VariableGroups**. The middle right diagram in Fig. 2 shows the structure within **MessagePassing**, which includes the leaf divisions **MessagePassingBasics**, **PointToPoint**, and **Collective**. The middle left diagram in Fig. 2 shows the structure within **FileSystem**, which includes the leaf divisions **FileSystemBasics**, **GridImageBasics**, **CurveImages**, **StructuredGridImages**, **UnstructuredGridImages**, and **PointGridImages**.

In these illustrations the compilation order is from bottom to top; thus the dependencies essentially flow in reverse, from top to bottom.

Before proceeding further we say a bit more about the way in which the classes in GENASIS are arranged and accessed. In discussing the organization of GENASIS—sketched in part in Figs. 1 and 2—we refer to the boxed entities as ‘divisions’ of the code. They are not classes, but subdirectories in the source code tree; thus they comprise and compose hierarchically arranged groups of classes. However, even though these divisions are not classes per se, we nevertheless mirror their hierarchical structure in Fortran `modules` that make groups of classes more convenient to access, as follows. In each directory there is a file defining a Fortran `module` whose name corresponds to the division or directory, and which contains all the classes (for a leaf division) or groups of classes (for a non-leaf division) within that directory. For instance, in the `Specifiers` directory (see the right diagram in Fig. 2) there is a file `Specifiers.f90` defining a `module Specifiers` containing Fortran `use` statements invoking each individual class in that directory. Then the classes in the `ArrayOperations` directory (the next to be compiled) need only the statement `use Specifiers` to access all the classes in that grouping. Similarly, the directory `VariableManagement` contains a file `VariableManagement.f90` as follows:

```
1 module VariableManagement
2   use Specifiers
3   use ArrayOperations
4   use ArrayArrays
5   use VariableGroups
6 end module VariableManagement
```

Then, in the classes grouped under, for example, `Display` (see the left diagram in Fig. 2), the statement `use VariableManagement` suffices to access everything contained within that division.

We describe the functionality available through `Basics` in the context of example problems, which also allow us to illustrate our object-oriented approach utilizing features of Fortran 2003. These examples and our object-oriented approach are presented in Section 2. `Basics` functionality is illustrated more particularly and systematically in Section 3. Section 4 provides instructions for compiling and building the examples and unit test programs. We conclude in Section 5.

2. Example Problems and Object-Oriented Solutions

We present two categories of nontrivial example problems for which solutions are built upon GENASIS Basics functionality: fluid dynamics and molecular dynamics. These are fundamentally different models, requiring the solution of different equations, using different techniques and different parallelization strategies. Nevertheless, GENASIS Basics serves as an excellent basis for coding solutions in both cases. In developing these examples we foreshadow future releases of the Mathematics and Physics portions of GENASIS by illustrating in a simple way the object-oriented mechanisms of inheritance and polymorphism that we use to separate lower-level coding of generic, reusable solvers from higher-level coding for specific physical systems.

In the case of the first example problem described below we present some explicit code. We do not show every detail, but only selected snippets that illustrate some points we wish to make about our object-oriented approach and, in Section 3, the functionality available in GENASIS Basics classes. All example programs are included in the accompanying submission to the CPC Program Library.

2.1. Fluid Dynamics

We present two kinds of fluid dynamics problems. The first is the periodic advection of a plane wave in mass density in one, two, or three position space dimensions (1D, 2D, 3D). The second is a Riemann problem (also 1D, 2D, 3D) which ventures beyond periodic advection to full fluid evolution with shocks and reflecting boundary conditions.

A few underlying classes provide a generic mathematical foundation for both types of fluid dynamics problems. These are: (a) a mesh—a discretization of space into cubic ‘cells’—that can be domain-decomposed into ‘bricks’ assigned to the several processes of a distributed-memory parallel program (`DistributedMeshForm`); (b) a template for a set of conserved fields, associated primitive and auxiliary fields, and some basic routines associated with them (`ConservedFieldsTemplate`); (c) a finite-volume conservation law stepper (`ConservationLawStepForm`); and (d) a template for the evolution of a set of conserved fields, for many time steps over a finite time (`ConservationLawEvolutionTemplate`). We implement each of these with a Fortran module defining a ‘class’ as understood in the object-oriented programming paradigm. Subsequent classes extend and use these to flesh out

the details of particular physical models, and set up and solve particular problems; see Fig. 3.

As it would be outside our main scope and purpose here—which is to introduce `GENASIS Basics` functionality—we do not discuss these fluid dynamics classes or the numerical methods they implement in any detail, though some snippets will be shown in Sec. 3 as we illustrate in more systematic and bottom-up fashion the functionality made available in `Basics`. The full example codes are included in the accompanying submission to the CPC Program Library. By way of summary, we note that the heart of the finite-volume solver, `ConservationLawStepForm`, uses piecewise linear reconstruction with an MC (monotonized central) slope limiter to get cell face values; an HLL (Harten-Lax-van Leer) Riemann solver to obtain fluxes from the left and right values at cell interfaces; and a second-order Runge-Kutta time step. Information and references on these aspects can be found in Ref. [1].

2.1.1. Plane Wave Advection

In order to simulate the periodic advection of a plane wave in mass density, for two different waveforms, we need to implement classes further specifying the necessary physics and setting up these particular problems. Referring again to Fig. 3, and continuing the enumeration above, these are (e) a realization of `ConservedFieldsTemplate` for the specific case of a pressureless fluid (`PressurelessFluidForm`); (f) an extension of `ConservationLawEvolutionTemplate` to a template for the evolution of a generic plane wave (`PlaneWaveAdvectionTemplate`); and (g) realizations of `PlaneWaveAdvectionTemplate` for two different waveforms (`SineWaveAdvectionForm` and `SawtoothWaveAdvectionForm`).

We gain perspective with a top-down discussion of this example, beginning with the `program` outlined in Listing 1, which performs the advection of a sine wave. The statement `use Basics` in line 3 gives access to all the classes in the code divisions pictured in Fig. 2. In line 7 we declare an object `SWA`¹ of class `SineWaveAdvectionForm`,² whose definition is available

¹The identifier `SWA` is an acronym for `SineWaveAdvection`. Such acronymic naming, especially for objects, is a very common practice of ours.

²Our convention is to use the suffix `Form` in class names. We normally define one class per Fortran `module`, with the suffix preceded by an underscore (`_Form`) in the `module` name to avoid a collision with the corresponding derived `type` name.

Figure 3: Classes built on GENASIS Basics classes for fluid dynamics example problems, categorized as ‘Mathematics’ (red), ‘Physics’ (green), and ‘Problem’ definition (blue). Light dotted lines indicate compilation dependencies. Heavy dash-dot lines denote class inheritance by type extension. The boxes with heavy outlines—`SineWaveAdvectionForm`, `SawtoothWaveAdvectionForm`, and `RiemannProblemForm`—represent fully defined problems ready for execution by a very short and simple driver program, such as `SineWaveAdvection` in Listing 1. The generic mathematical structure of conservation laws is separated from the specifics of particular problems: `ConservedFieldsTemplate` is extended to `PressurelessFluidForm`, and subsequently to `PolytropicFluidForm`, by way of defining specific sets of physical variables; and `ConservationLawEvolutionTemplate` is extended to `PlaneWaveAdvectionTemplate` (which is further extended to two different waveforms) and `RiemannProblemForm` in order to set up specific systems governed by conservation laws.

Listing 1: Outline of program `SineWaveAdvection`.

```

1 program SineWaveAdvection
2
3   use Basics
4   use SineWaveAdvection_Form   !– See Listing 2
5   implicit none
6
7   type ( SineWaveAdvectionForm ) :: SWA   !– See Listing 2
8
9   !– Program initialization omitted; see Section 3.5
10
11  call SWA % Initialize ( )   !– See Listings 2 and 6
12  call SWA % Evolve ( )      !– See Listing 4
13
14  !– Program finalization omitted
15
16 end program SineWaveAdvection

```

through the `use` statement in line 4. The `subroutine` calls in lines 11 and 12 are invocations of methods belonging to the object `SWA`.³

Evidently we must consider the class `SineWaveAdvectionForm` defined in the `module` sketched in Listing 2 to begin to see how the highest-level tasks—initializing and evolving the problem—are implemented. Two members of `SineWaveAdvectionForm` are declared in line 10, the `Offset` and `Amplitude` of the sine wave. (An offset is included because the plane wave is in the density, and our `PressurelessFluidForm` methods expect the density to be positive.) `Offset` and `Amplitude` are objects of class `MeasuredValueForm`; as we will discuss in Section 3.1.1, this allows them to have a unit of measure associated with them. Lines 12–14 are declarations related to two of the methods of `SineWaveAdvectionForm`.⁴ The `generic`

³The syntax `[variable] % [procedure]` to reference methods is new to Fortran 2003. Procedures—either `subroutines` or `functions`—are attached to derived `types` through one of two new mechanisms: `procedure` pointer components, or, as we shall see for instance in Listing 2, type-bound `procedure` declarations (e.g. [2]). By default the object that owns the method is the first—and so-called *passed*—argument to the invoked procedure, giving that procedure access to the object’s members. Additional arguments follow in parentheses. In lines 11 and 12 of Listing 1 the parentheses following the method names are empty because the object—in this case `SWA`—is the only argument to those `subroutines`.

⁴The procedure named in line 12 has the `private` attribute, which means it is not accessible outside this `module`. The default, which we typically affirm explicitly as in line 14, is `public` availability. The methods in lines 12 and 14 also have the `pass` attribute, which affirms the default behavior that the object owning the method is passed as the

Listing 2: Outline of module `SineWaveAdvectionForm`. Used at line 4 of Listing 1.

```

1 module SineWaveAdvection_Form
2
3   use Basics
4   use PlaneWaveAdvection_Template    !- See Listing 3
5   implicit none
6   private
7
8   type, public, extends ( PlaneWaveAdvection_Template ) &
9     :: SineWaveAdvectionForm
10    type ( MeasuredValueForm ) :: Offset , Amplitude
11  contains
12    procedure, private, pass :: Initialize_SWA    !- See Listing 6
13    generic, public :: Initialize => Initialize_SWA
14    procedure, public, pass :: Waveform
15    final :: Finalize
16  end type SineWaveAdvectionForm
17
18  contains
19
20    !- Definitions of subroutines Initialize_SWA and Finalize, and function Waveform, omitted
21
22 end module SineWaveAdvection_Form

```

statement in line 13 provides a publicly available alias `Initialize` for the private method `Initialize_SWA` declared in line 12; as we shall see later, the method `Initialize` will be *overloaded* to alias other subroutines as well. The `final` routine declared in line 15 is automatically executed when an object of `SineWaveAdvectionForm` goes out of existence; such routines are typically used for ‘garbage collection’ (for instance, to `deallocate` any allocatable members, in order to avoid memory leaks).

Of the two methods invoked in lines 11 and 12 of Listing 1, only one—`Initialize`—is made publicly available in Listing 2 (via line 13); where is `Evolve`? Consider the indication in line 8 of Listing 2 that this class `extends` the class `PlaneWaveAdvection_Template` outlined in Listing 3. In turn, note in line 9 of Listing 3 that `PlaneWaveAdvection_Template` `extends` the class `ConservationLawEvolution_Template`, outlined in Listing 4.

The sequential extensions of `ConservationLawEvolution_Template` to `PlaneWaveAdvection_Template` to `SineWaveAdvectionForm` provide an initial look at the object-oriented principles of inheritance and polymorphism. The idea in this particular example is that

first argument [e.g. 2].

Listing 3: Outline of module `PlaneWaveAdvectionTemplate`. Used at line 4 of Listing 2.

```

1  module PlaneWaveAdvection_Template
2
3  use Basics
4  use ConservationLawEvolution_Template
5  use PressurelessFluid_Form
6  implicit none
7  private
8
9  type, public, extends ( ConservationLawEvolutionTemplate ), abstract :: &
10     PlaneWaveAdvectionTemplate
11  contains
12     procedure, private, pass :: Initialize_PWA      !- See Listing 5
13     generic, public :: Initialize => Initialize_PWA
14     procedure ( WaveformInterface ), public, pass, deferred :: Waveform
15  end type PlaneWaveAdvectionTemplate
16
17  !- Declaration of the interface for function WaveformInterface omitted
18
19  contains
20
21  !- Definition of subroutine Initialize_PWA omitted
22
23 end module PlaneWaveAdvection_Template

```

`ConservationLawEvolutionTemplate` contains members and methods needed to evolve any set of conserved fields. The extension `PlaneWaveAdvectionTemplate` adds members and methods useful to solve the advection of a plane wave for any waveform, while `SineWaveAdvectionForm` adds only the elements necessary to specify that the waveform be a sine wave. `PlaneWaveAdvectionTemplate`, and in turn `SineWaveAdvectionForm`, are said to *inherit* all the members declared and suggested by lines 11-14 of Listing 4. `SineWaveAdvectionForm` also inherits the methods declared in lines 12-14 of Listing 3 and lines 16-18 of Listing 4.

This inheritance is what allows, for instance, the object `SWA` of class `SineWaveAdvectionForm` declared in line 7 of the program in Listing 1 to invoke, in line 12, the method `Evolve` of its ancestor `ConservationLawEvolutionForm`. The outline of `Evolve` is shown in lines 25-28 of Listing 4. The declaration of the argument `CLE` with the `class` keyword instead of the `type` keyword in line 26 makes `CLE` a polymorphic variable, which means that this argument can be any extension (or extension

Listing 4: Outline of module ConservationLawEvolution_Template. Used at line 4 of Listing 3.

```

1  module ConservationLawEvolution_Template
2
3      use Basics
4      use DistributedMesh_Form
5      use ConservedFields_Template
6      use ConservationLawStep_Form
7      implicit none
8      private
9
10     type, public, abstract :: ConservationLawEvolution_Template
11         !- Declaration of several members omitted
12         type ( DistributedMesh_Form ) :: DistributedMesh
13         class ( ConservedFields_Template ), allocatable :: ConservedFields
14         type ( ConservationLawStep_Form ) :: ConservationLawStep
15     contains
16         procedure, private, pass :: Initialize_CLE
17         generic, public :: Initialize => Initialize_CLE
18         procedure, public, pass :: Evolve
19     end type PlaneWaveAdvection_Template
20
21 contains
22
23     !- Definition of subroutine Initialize_CLE omitted
24
25     subroutine Evolve ( CLE )
26         class ( ConservationLawEvolution_Template ), intent ( inout ) :: CLE
27         !- Declaration of local variables and other statements omitted
28     end subroutine Evolve
29
30 end module ConservationLawEvolution_Template

```

of extension, and so on) of `ConservationLawEvolutionForm`.⁵ This is what allows a `SineWaveAdvectionForm` object—ultimately, `SWA` declared in line 7 of Listing 1—to be the first argument of `Evolve`.

In this particular example, `PlaneWaveAdvectionTemplate` and `ConservationLawEvolutionTemplate` are abstract, as specified in lines 9 and 10 of Listings 3 and 4 respectively.⁶ In the case of `PlaneWaveAdvectionTemplate` this is required by the fact that its method `Waveform` is deferred (as declared in line 14 of Listing 3). This means that, unlike `Initialize_PWA` declared in line 12—which is defined or ‘fleshed out’ in this class definition, as suggested by line 21—the method `Waveform` is only given an interface, or specification of its argument list, as suggested by line 17. In this example the method `Waveform` is not fleshed out until `PlaneWaveAdvectionTemplate` is extended to `SineWaveAdvectionForm` in Listing 2, in which the previous deferred declaration is said to be *overridden* by the new declaration in line 14, and the method is defined as suggested in line 20. (The importance of procedure overriding for a powerful manifestation of polymorphism will be emphasized below.)

`ConservationLawEvolutionTemplate` in Listing 4 is declared abstract for a different reason. In line 13, the allocatable member `ConservedFields` is declared with the `class` keyword rather than the `type` keyword. We saw one use of the `class` keyword previously, in connection with a subroutine argument. The `class` keyword is also allowed in the declaration of an allocatable variable, which means that it can be allocated as the base type given in the declaration *or any extension thereof*. In this particular case, the member `ConservedFields` in line 13 cannot be allocated as the declared base type `ConservedFieldsTemplate`, because this type happens to be abstract—having deferred methods—and concrete instances of abstract types are not allowed. In the present example we use the extension `PressurelessFluidForm` of `ConservedFieldsTemplate`, which extends a generic template for a set of conserved fields to one with

⁵That the `class` keyword denotes a polymorphic variable in Fortran 2003 must not be confused with our (standard in the object-oriented context) use of the term ‘class’ to refer to a construct that bundles together members and methods.

⁶Our convention is to use the suffix `Template` for an abstract class instead of the suffix `Form` we use for typical classes. We emphasize that inheritance by type extension does not only occur in going from an abstract class to a non-abstract one, but between any classes at all—and indeed through any number of sequential extensions.

Listing 5: subroutine Initialize_PWA. Fits in line 21 of Listing 3. Called at line 13 of Listing 6.

```

1  subroutine Initialize_PWA ( PWA, DensityUnit )
2
3      class ( PlaneWaveAdvectionTemplate ), intent ( inout ) :: PWA
4      type ( MeasuredValueForm ), intent ( in ) :: DensityUnit
5
6      allocate ( PressurelessFluidForm :: PWA % ConservedFields )
7      !- Initialization of PWA % ConservedFields omitted
8
9      !- Local variable declarations and assignments omitted
10
11     !- N is a real array containing density values in the mesh cells, and X, Y, and Z are real
12     ! arrays containing the Cartesian coordinates of the mesh cells; whereas K is the wave vector
13     ! of the plane wave, a three-element real array
14
15     N = PWA % Waveform ( K ( 1 ) * X + K ( 2 ) * Y + K ( 3 ) * Z )
16
17 end subroutine Initialize_PWA

```

mass density, momentum density, and velocity fields. Allocation of the ConservedFields member as PressurelessFluidForm does not occur in ConservationLawEvolutionTemplate, which does not have access to any non-abstract extensions of ConservedFieldsTemplate. Instead, this allocation occurs in an extension of ConservationLawEvolutionTemplate that *does* have access to PressurelessFluidForm; see line 6 of Listing 5 for Initialize_PWA, which fits in line 21 of Listing 3 for PlaneWaveAdvectionTemplate. We are not *required* to declare ConservationLawEvolutionTemplate as abstract, because it does not have any deferred methods; but we *choose* to do so in order to force the creation of an extension (in this case PlaneWaveAdvectionTemplate), one with access to a non-abstract extension of ConservedFieldsTemplate (in this case PressurelessFluidForm), which can therefore be responsible for allocating and initializing the ConservedFields member.

We now return to the overloading of the Initialize method of SineWaveAdvectionForm, called in line 11 of the main program in Listing 1. While not apparent in Listing 1, there is a difference behind the scenes between the two method invocations in lines 11 and 12. We have seen that the subroutine Evolve is declared as a method and defined in SineWaveAdvectionForm's ancestor ConservationLawEvolutionTemplate in Listing 4. In contrast, there is no subroutine named Initialize in SineWaveAdvectionForm or its ancestors. Instead, the method name

Listing 6: subroutine `Initialize_SWA`. Fits in line 19 of Listing 2. Called at line 11 of Listing 1.

```

1  subroutine Initialize_SWA ( SWA )
2
3  class ( SineWaveAdvectionForm ), intent ( inout ) :: SWA
4  type ( MeasuredValueForm ) :: OffsetUnit , AmplitudeUnit
5
6  OffsetUnit    = UNIT % IDENTITY
7  AmplitudeUnit = UNIT % IDENTITY
8  SWA % Offset  = 2.0_KDR * OffsetUnit
9  SWA % Amplitude = 1.0_KDR * AmplitudeUnit
10  !- Statements overriding these defaults with values read from parameter file
11  ! or command line omitted
12
13  call SWD % Initialize ( AmplitudeUnit ) !- See Listing 5
14
15 end subroutine Initialize_SWA

```

`Initialize` is overloaded by virtue of the `generic` statements in line 13 of Listing 2, line 13 of Listing 3, and line 17 of Listing 4, making `Initialize` an alias for three different subroutines named `Initialize_SWA`, `Initialize_PWA`, and `Initialize_CLE`. This is possible because these three routines have different argument lists, which allows the system to *resolve* any particular call (i.e. route it to the correct `subroutine`) by finding the aliased routine with the appropriate argument list.⁷ To see how this works, consider the subroutines `Initialize_SWA` and `Initialize_PWA` in Listings 6 and 5, which would fall in line 20 of Listing 2 of `SineWaveAdvectionForm` and line 21 of Listing 3 of `PlaneWaveAdvectionTemplate` respectively. Notice that `Initialize_SWA` (Listing 6) has only one argument while `Initialize_PWA` (Listing 5) has two. This difference allows the call in line 11 of the program in Listing 1 to be resolved to `Initialize_SWA` (with the object `SWA` to the left of the `%` symbol passed as the only argument), and the call in line 13 of Listing 6 to be resolved to `Initialize_PWA` (with `SWA` as the first argument and `AmplitudeUnit` as the second argument).⁸

⁷Procedure overloading is sometimes called ‘ad-hoc polymorphism.’ But some prefer to use the term ‘polymorphism’ only in connection with inheritance and method overriding, to refer to different behaviors of the same method name depending on which extension of a parent type the method is owned by, rather than different behavior obtained with different argument lists.

⁸Procedure overloading can occur both within a single `type` definition and, as in this case, across the definitions of a `type` and its extensions.

Finally, we return to the deferred method `Waveform` of `PlaneWaveAdvectionTemplate` in Listing 3 in order to get a glimpse of the power of polymorphism manifest by procedure overriding in association with class inheritance. Line 15 of Listing 5 for subroutine `Initialize_PWA` calls the method `Waveform` that sets the plane wave values in the mesh cells (see also the comment in lines 11-13). As far as `PlaneWaveAdvectionTemplate` is concerned, this function could return anything; its definition is deferred, with only an interface specified (lines 14 and 17 of Listing 3). But when the argument `PWA` is an instance of `SineWaveAdvectionForm`, the function call resolves to the overriding version of `Waveform` specified by lines 14 and 20 of Listing 2 of `SineWaveAdvectionForm`, returning values of a sine wave. And similar to the sine wave case in Listings 1 and 2, we have also implemented a very short and simple driver program `SawtoothWaveAdvection` and module `SawtoothWaveAdvection_Form`. In this case, when the argument `PWA` in Listing 5 is an instance of `SawtoothWaveAdvectionForm`, that class' method `Waveform` is called and returns the values of a sawtooth wave.

This simple example illustrates that the principles of inheritance and polymorphism—embodied here in the mechanisms of `type` extension and method overriding—make it much easier to allow lower-level code (in this example, `PlaneWaveAdvectionTemplate`) to access higher-level code (in this example, the `Waveform` routines specific to `SineWaveAdvectionForm` and `SawtoothWaveAdvectionForm`).

The reason this language functionality is important for our purposes in GENASIS is that it greatly facilitates, for instance, the separation of the `Mathematics` and `Physics` divisions of GENASIS. For example, solvers for generic classes of equations can be written in `Mathematics`, and then invoked later by a range of different systems whose details are specified in `Physics`. This tremendously enhances the ease and transparency with which one can develop (for example) versatile and widely-applicable solvers.

In fact we do just this sort of thing with the conservation law solver in this plane wave advection example. To this point—taking a top-down perspective—we have only discussed driver programs (Listing 1 for the sine wave case), an `abstract` class for the initialization and periodic advection of a plane wave of arbitrary waveform (Listing 3), and specific implementations of this template class (Listing 2 for the sine wave case). The latter two are classes (f) and (g) in the list of classes in the first paragraph of this subsection. Continuing backwards through this list, classes (b)-(e) use the mechanism of polymorphism through inheritance to separate lower-level

generic code from higher-level specific code. In particular, (c) and (d) constitute a solver for a generic system governed by conservation laws, which draws on certain properties specified in the **abstract** template (b)—for instance, indices of ‘primitive’ and ‘conserved’ variables, and **deferred** methods for converting between these sets of variables, computing raw fluxes, setting boundary conditions, and so on. Class (e) is a higher-level realization of template (b) for a specific system, a pressureless fluid. It specifies mass density and velocity as primitive variables, and fills in routines that were **deferred** (specified only by interface) in template (b).

We now show results from **SineWaveAdvection** and **SawtoothWaveAdvection** in 1D, 2D, and 3D. In the left panels of Figs. 4 and 5 we plot the mass density of the initial conditions of **SineWaveAdvection** and **SawtoothWaveAdvection**, respectively. In the right panels we plot the relative error δ against the initial condition after one period of evolution, where $\delta = (\chi_t - \chi_{t0}) / \chi_{t0}$ for every cell, and χ_t and χ_{t0} are the values of the density at the initial initial and final state, respectively. For these figures we have used a resolution of 128 cells in each spatial dimension. For **SawtoothWaveAdvection**, we have set the number of wavelengths to two, overriding the default value of one in the driver program via a command line option (see Section 3.5 for this feature; see also Section 4 for examples on how these programs were run).

Problems with smooth flows like the **SineWaveAdvection** example can be used to check that the order of convergence of the numerical implementation is as expected for the numerical method employed. The error of the solution as a function of mesh resolution yields the convergence rate of the numerical solution. Here we use the L_1 -norm relative error, given by

$$L_1(\chi) = \frac{\sum_{i,j,k} |\chi(x_i, y_j, z_k) - \chi_0(x_i, y_j, z_k)|}{\sum_{i,j,k} |\chi_0(x_i, y_j, z_k)|}, \quad (1)$$

where χ_0 is the ‘known’ value used as a standard for comparison. For the **SineWaveAdvection** program, χ_0 and χ are the mass density at initial conditions and after one period of evolution, respectively. In Eq. 1 the summation is over all cells on the computational domain. Thus the L_1 norm gives a single number as a quantitative error measure for a particular mesh resolution. The left panel of Fig. 6 shows that **SineWaveAdvection** converges to second order as expected.

Figure 4: Plots of density for the 1D (upper), 2D (middle), and 3D (lower) versions of the SineWaveAdvection initial conditions (one wavelength, left) and their relative errors against the initial conditions after one period of evolution (right) at a resolution of 128 cells in each dimension. The 1D plot of density (upper left) shows both the initial (red) and final (green) states, which visually overlap almost perfectly.

Figure 5: Plots of density for the 1D (upper), 2D (middle), and 3D (lower) versions of the SawtoothWaveAdvection initial conditions (two wavelengths, left) and their relative errors against the initial conditions after one period of evolution (right) at a resolution of 128 cells in each dimension. The 1D plot of density (upper left) shows both the initial (red) and final (green) states, revealing the degradation in accuracy resulting from the drop to first order at discontinuities.

Figure 6: *Left:* L_1 norm relative error versus mesh resolution of 2D (blue circles) and 3D (red squares) **SineWaveAdvection**. The dot-dashed and dashed lines are reference lines proportional to first and second-order convergence, respectively. *Right:* Weak scaling of the **SineWaveAdvection** program on a Cray XC30 supercomputer with 64^3 cells per process. The black curve with square symbols shows the parallel efficiency of the code with the program's elapsed time with eight processes as the baseline. The red curve with circle symbols shows the parallel efficiency of the code when run with Hyper-Threading—Intel's implementation of simultaneous multithreading to let every physical core be used as two logical cores—enabled so that we can run to higher process counts. For this run, we have used the elapsed time at twenty seven processes as our baseline since this is the minimum number of processes that utilized almost all logical cores on a compute node. The first points correspond to elapsed wall clock time of 1.72 and 3.52 seconds on the black and red curves, respectively.

Finally, we consider the scaling behavior of `SineWaveAdvection` as the number of processes is increased. The weak scaling of a parallel program is the extent to which it maintains its parallel efficiency as the number of processes is increased, while keeping the amount of work per process constant. Parallel efficiency is the ratio of program's elapsed time with a particle number of processes over the elapsed time with a baseline number of processes. In the right panel of Fig. 6 we show the weak scaling of the `SineWaveAdvection` example on Darter [3], a Cray XC30 supercomputer at JICS.⁹ For this test every process was assigned 64^3 cells. The total elapsed time is normalized by the number of time steps required to complete one period of evolution. The total elapsed time of this example program is output to the screen at the program's conclusion as provided by a class in the `Runtime` division of `GENASIS Basics` (see Section 3.5 for this feature).

2.1.2. Riemann Problem

In solving the Riemann problem we reuse most of the code used to solve the plane wave advection problem discussed in Sec. 2.1.1 (see also Fig. 3). The Riemann problem takes place in a box with reflecting boundaries, and features initial conditions that consist of two regions with different densities and pressures separated by a discontinuity. The classes listed in the second paragraph of Sec. 2.1—(a) `DistributedMeshForm`, (b) `ConservedFieldsTemplate`, (c) `ConservationLawStepForm`, and (d) `ConservationLawEvolutionTemplate`—are sufficiently generic and ‘pure mathematics’ in nature to be used directly without modification.

We also reuse one of the classes mentioned in the first paragraph of Sec. 2.1.1. In particular, we extend (e) `PressurelessFluidForm`—which includes mass density and velocity—to `PolytropicFluidForm`, which adds a few variables such as pressure, internal energy, and parameters associated with a polytropic equation of state. `PolytropicFluidForm` also overrides many of the methods of `PressurelessFluidForm`. But even these overriding methods are able to reuse code in most cases, by first calling the overridden methods of `PressurelessFluidForm` to handle the portions relevant to its variables (mass density, velocity, etc.), and then separately addressing only the variables new to `PolytropicFluidForm` (pressure, energy, etc.).

⁹Joint Institute for Computational Sciences <http://www.jics.tennessee.edu>

On the other hand, the code defining the specific problem must be replaced, but this is a small line count compared to the code as a whole. In particular, (f) `PlaneWaveAdvectionTemplate` and its extensions (g) `SineWaveAdvectionForm` and `SawtoothWaveAdvectionForm` are completely replaced by a new class `RiemannProblemForm`, which specifies reflecting boundary conditions and sets the initial conditions for this problem.

We now show results from `RiemannProblem` in 1D, 2D, and 3D. The default initial conditions for the `RiemannProblem` are those typically known as the Sod shock tube problem [4] for 1D, which we have also extended to 2D and 3D. Figure 7 plots the initial (left panels) and final state (right panels) of the fluid at $t = 0.25$.

2.2. Molecular Dynamics

We present two kinds of molecular dynamics problems. The first is the equilibrium of bulk argon; we examine the solid, liquid, and gas phases with a periodic box as proxy for an infinite system, using the Lennard-Jones potential. The second is the gravitational collapse of an isolated system of particles (e.g. ‘stars’) which evolves to a state satisfying the virial theorem.

A few underlying classes provide a generic mathematical foundation for both types of molecular dynamics problems. These are: (a) a function that returns a random number drawn from a Gaussian distribution (`NormalRandomNumberFunction`); (b) a class that provides for storage of the positions and velocities of a set of particles ‘owned’ by a particular process (`ParticlesForm`); (c) a class whose members include an instance of `ParticlesForm`, along with members and methods associated with some basic observables associated with a collection of particles, with the expectation that the totality of particles is divided among the several processes of a distributed-memory parallel program (`DistributedParticlesForm`); and (d) a template for the evolution of an instance of `DistributedParticlesForm` or one of its extensions, whose methods include integration of Newton’s laws of motion for all particles (including a parallelized computation of the total force on each particle), and a deferred method specifying a particular force law (`ParticleDynamicsTemplate`). Subsequent classes extend and use these to flesh out the details of particular physical models, and set up and solve particular problems; see Fig. 8.

As it would be outside our main scope and purpose here—which is to introduce `GENASIS Basics` functionality—we do not discuss these molecular dynamics example classes or the numerical methods they implement in any

Figure 7: Density in 1D (upper), 2D (middle), and 3D (lower) versions of RiemannProblem at $t=0$ (left) and $t=0.25$ (right) at a resolution of 128 cells in each dimension. For 1D, pressure (green cross symbols) and velocity (blue plus symbols) are also plotted.

Figure 8: Classes built on GENASIS Basics classes for molecular dynamics example problems, categorized as ‘Mathematics’ (red), ‘Physics’ (green), and ‘Problem’ definition (blue). Light dotted lines indicate compilation dependencies. Heavy dash-dot lines denote class inheritance by type extension. The boxes with heavy outlines—`LennardJonesDynamicsForm` and `GravitationalDynamicsForm`—represent problems ready for execution by short and simple driver programs, such as `ArgonEquilibrium` and `ClusterFormation` respectively. The generic mathematical structure of a system of particles governed by Newton’s laws of motion is separated from the specifics of particular problems: `DistributedParticlesForm` is extended to `LatticeParticlesForm`, and separately to `ClusterParticlesForm`, by way of setting up either a periodic lattice or an isolated cluster respectively; and `ParticleDynamicsTemplate` is extended to `LatticeDynamicsTemplate` and `LennardJonesDynamicsForm` to specify observables relevant to atomic-scale problems and then a particular force law, and separately to `ClusterDynamicsTemplate` and `GravitationalDynamicsForm` to do the same for an isolated self-gravitating cluster.

1
2
3
4
5
6
7
8
9 detail. The full example codes are included in the accompanying submission
10 to the CPC Program Library. Here we provide only a brief overview.

11
12 For purposes of this example we take the simple-minded approach of com-
13 puting the pairwise forces between all particles, such that the work grows as
14 the square of the number of particles. As a reflection of this, for the results
15 presented below we chose numbers of particles and processes that gave sim-
16 ilar wall-clock runtimes for each run (up to about 1.2 hours on Darter, the
17 Cray XC30 at NICS): 2048, 6912, 16384, and 32000 particles, with 1, 12, 64,
18 and 250 processes respectively. The parallelization strategy involves giving
19 each process its equal share of particles, all of which may be anywhere in the
20 computational domain (a periodic box in the case of argon equilibrium, and
21 an ‘infinite’¹⁰ box in the case of cluster formation by gravitational collapse).
22 The parallelization of the force computation involves a ‘ring algorithm,’ in
23 which each process has storage for ‘guest particles’ from another process in
24 addition to the local particles it ‘owns.’ The force computation begins with
25 each process copying the positions of its local particles to the guest particle
26 storage, so that the contribution to the forces on each of its local particles by
27 all the other local particles can be computed. Then each process sends the
28 positions in its guest particle storage to the ‘next’ process, and their contribu-
29 tions are added to the forces on the local particles. This is repeated until all
30 processes have computed the contributions from all the other processes’ par-
31 ticles. We have implemented the ‘leapfrog’ and ‘velocity-Verlet’ approaches
32 as options for integrating the particles’ equations of motion; once the forces
33 are computed, these position and velocity updates are trivially parallel and
34 comparatively inexpensive. In the case of a periodic box, the ‘minimum im-
35 age convention’ is employed, in which only the particle or its nearest periodic
36 image is used in computing the force components. In the case of the gravita-
37 tional force, which has no repulsion, we use potential softening at small scales
38 by way of regularization of close encounters (i.e. avoiding very tight binaries,
39 the detailed following of which would cause debilitatingly small time steps).
40 For further information on molecular dynamics and N-body problems, see
41 for instance Refs. [5, 6, 7].
42
43
44
45
46
47
48
49
50

51
52 ¹⁰Technically, `huge ( 1.0_KDR )`, where the meaning of KDR is discussed in Sec. 3.1.1.
53
54
55
56
57
58

Table 1: Parameters for `ArgonEquilibrium` runs

Phase	Density ($\text{\AA}^{-3}$)	Temperature (K)
Solid	3.040×10^{-2}	59.90
Liquid	2.026×10^{-2}	119.8
Gas	7.599×10^{-3}	359.4

2.2.1. Argon Equilibrium

In order to simulate the solid, liquid, and gas phases of argon, we need to implement classes further specifying the necessary physics and setting up this particular problem. Referring again to Fig. 8, and continuing the enumeration above, these are (e) an extension of `DistributedParticlesForm` that sets up particles on a Bravais lattice (in this case FCC, i.e. face-centered cubic) with a Maxwell-Boltzmann velocity distribution corresponding to a chosen temperature (`LatticeParticlesForm`); (f) an extension of `ParticleDynamicsTemplate` that adds the setup, computation, and recording of observables relevant to an atomic system (`LatticeDynamicsTemplate`); and (g) an extension of `LatticeDynamicsTemplate` that finally specifies a particular force law (`LennardJonesDynamicsForm`). The short and simple driver program `ArgonEquilibrium` initializes and evolves an instance of `LennardJonesDynamicsForm`, and an associated parameter file includes the force parameters appropriate for argon (see Sec. 3.4.1).

We now show results obtained with `ArgonEquilibrium`. We ran simulations of the solid, liquid, and gas phases using values for number density and temperature given in Table 1. Each simulation ran for 10000 time steps. The initial and final configurations of runs with 2048 particles are shown in Fig. 9. From this figure it is evident that only the solid phase retains its initial FCC lattice configuration.

Figure 10 shows that the simulations begin with an equilibration phase. Because the initial conditions do not represent equilibrium, there is an almost immediate exchange of kinetic and potential energy apparent in the earliest moments in left panel of Fig. 10. The temperature—computed from the kinetic energy via the equipartition relation, and shown for the gas phase in the right panel of Fig. 10—accordingly departs from the input ‘target’ temperature value. (The right panel also illustrates the reduction in fluctuations that comes by increasing the number of particles in the simulation.) The

Figure 9: Initial (left) and final (right) states of simulations of the solid (top), liquid (middle), and gas (bottom) phases of argon. This figure shows runs with 2048 particles, whose positions are given by points and whose velocities are indicated by vectors.

Figure 10: *Left:* Kinetic (red), potential (green), and total (blue) energies for simulations of the solid (solid), liquid (dashed), and gas (dotted) phases of argon with 32000 particles. *Right:* Temperature of argon gas phase simulations with 2048 (red), 6912 (green), 16384 (blue), and 32000 (yellow) particles.

first third of each simulation is devoted to achieving equilibrium at the target temperature by periodically rescaling the velocities so as to reflect the target temperature. This process is visible as ‘steps’ in the energies plotted in the left panel, including the total energy. It is apparent that the system has settled into equilibrium at the desired target temperature well within the first third of the simulation, after which no further velocity rescalings are performed. Thermodynamic averages can then be computed as time averages of the data generated in the last two-thirds of the simulation.

While the fact that only the solid phase retains the lattice structure of its initial condition can be directly apprehended from Fig. 9, the correlation functions shown in Fig. 11 distinguish all three phases from one another. The correlation function is essentially the probability of finding a particle at a given distance from another particle. It is time-averaged from the final two-thirds of the simulations. The correlation functions for all three phases are zero out to a distance corresponding to the repulsive hard core of the Lennard-Jones potential. All three phases then feature a peak—albeit reduced in amplitude as one goes from solid to liquid to gas—corresponding to the bottom of the attractive portion of the Lennard-Jones potential. Beyond this, the correlation function for the solid phase shows pronounced peaks corresponding to the distances between particles in the lattice structure. In the liquid phase there are a few additional peaks, more evenly spaced, less pronounced, and decaying more quickly with distance. The gas phase has very little structure beyond the first peak, as the random motions of particles less influenced by the potential quickly result in an average density. The cor-

Figure 11: Correlation functions for the solid (upper left), liquid (upper right), and gas (lower left) phases of argon, as computed from simulations with 6912 (blue), 16384 (green), and 32000 (red) particles. The lower right panel shows the mean square displacement of particles from their initial positions for the solid (blue), liquid (green), and gas (red) phases in simulations with 32000 particles. The linear behavior is characteristic of diffusion, with the slope as diffusion constant; the early quadratic behavior shown in insets represents initial free particle motion for times shorter than the typical collision time.

relation function is computed to larger distances for simulations with more particles, because a larger number of particles at a given density corresponds to a larger periodic box.

The lower right panel of Fig. 11, showing the mean square displacement of particles from their initial positions, also distinguishes between the phases. The linear behavior is characteristic of diffusion, with the slope as diffusion constant; not surprisingly, particles in the gas phase diffuse more rapidly than those in the liquid phase, while particles in the solid phase do not diffuse at all, showing zero mean square displacement around their original lattice positions. The early quadratic behavior of the gas and liquid phases shown in insets represents initial free particle motion for times shorter than the typical collision time.

2.2.2. Cluster Formation

In order to simulate the gravitational collapse of an isolated system of particles, we implement some alternative extensions to the generic classes listed in the second paragraph of Sec. 2.2. Continuing the enumeration there, and referring again to Fig. 8, these are (e) an extension of `DistributedParticlesForm` that sets up particles with a Gaussian spatial distribution, as well as a Maxwell-Boltzmann velocity distribution corresponding to an initial kinetic energy much smaller than the initial potential energy (`ClusterParticlesForm`); (f) an extension of `ParticleDynamicsTemplate` that adds the setup, computation, and recording of observables relevant to an isolated cluster (`ClusterDynamicsTemplate`); and (g) an extension of `ClusterDynamicsTemplate` that finally specifies a particular force law (`GravitationalDynamicsForm`). The short and simple driver program `ClusterFormation` initializes and evolves an instance of `GravitationalDynamicsForm`.

We now show results for `ClusterFormation`. Unlike `ArgonEquilibrium`, in which dimensioned parameters corresponding to the real world were used, we set up an idealized dimensionless problem. The initial particle position components were drawn from a Gaussian distribution with a standard deviation of 0.5. The particle mass was chosen such that the total mass was 1.0. The standard deviation of the velocity components was chosen such that the initial kinetic energy was roughly 3% of the initial potential energy. Each simulation ran for 10,000 time steps.

Figures 12-14 illustrate and summarize our `ClusterFormation` runs. Fig-

Figure 12: Initial and final states of a `ClusterFormation` run with 2048 particles, with particle positions shown at $t = 0.0$ (left) and $t = 10.0$ (right).

Figure 12 shows the initial and final conditions of a run with 2048 particles. The initial cluster of particles (Fig. 12 left), having much less kinetic than potential energy, undergoes gravitational collapse such that by the end of the simulation there is a dense core, but also a number of particles that seem to be unbound and escaping to infinity (Fig. 12 right). The box framing the inner part of the domain is the same size as that in the magnified view of Fig. 13, which shows a time sequence of particle positions.

The sequence of selected frames in Fig. 13 can be examined alongside the time series in Fig. 14. In the initial state, most of the particles (2σ worth, with position components in Gaussian distributions) are within a radius of 1.0 (Fig. 13 upper left). This is only within a factor of order unity of the ‘cluster size’ $-GM^2/U$ (where G is the gravitational constant, M is the sum of all the particles’ masses, and U is the gravitational potential energy of the system) at $t = 0$ in the left panel of Fig. 14; this is because this cluster length scale is only a figure of merit intended to convey changes in the size of the cluster during the simulation. Collapse is well underway by $t = 0.75$ (Fig. 13 upper right). Maximum compression has almost been achieved at $t = 1.0$ (Fig. 13 middle left), but it is clear that this configuration ‘overshoots’ the final situation. The middle right panel of Fig. 13 at $t = 1.5$ still shows a dense core, but also a significant ‘halo’ of outward-moving particles that have rebounded from the initial collapse. After this initial rebound, the nature of the system does not change much between $t = 3.0$ (Fig. 13 lower left) and $t = 10$ (Fig. 13 lower right).

The values of kinetic energy T and potential energy U at the end of

Figure 13: Closer view of a time sequence of a `ClusterFormation` run with 2048 particles, with particle positions shown at $t = 0.0$ (upper left), $t = 0.75$ (upper right), $t = 1.0$ (middle left), $t = 1.5$ (middle right), $t = 3.0$ (lower left), and $t = 10.0$ (lower right).

Figure 14: Left panel: Plots of $-GM^2/U$, a rough estimate of the cluster size, as a function of time, for runs with 2048 (orange), 6912 (blue), 16384 (green), and 32000 particles (red). Right panel: Kinetic (red), potential (green), and total (blue) energies as a function of time for runs with 2048 (dotted), 6912 (dashed), 16384 (dot-dashed), and 32000 particles (solid).

the simulation in the right panel of Fig. 14 seem consistent with the virial theorem $2T = -U$ for a gravitationally bound system. The total energy stays constant as expected, with the only minor visible deviation occurring at the most dynamic moment of maximum compression. There is not a great deal of variation between the curves in Fig. 14 for runs with different numbers of particles. One perhaps notable exception is that the curves for the run with the largest number of particles (the red curve in the left panel, for instance) seem to be capturing a regular ‘ring down’ oscillation between $t = 2$ and $t = 4$ that the runs with fewer numbers of particles may not be able to capture. There may be a slight slope to the kinetic and potential energy curves between $t = 3$ and $t = 10$; this might reflect a slow settling to final equilibrium, but we also note that the virial theorem applies to a bound system, whereas the energy values calculated here apparently include a relatively small but perhaps non-negligible number of unbound particles escaping to infinity.

3. Illustrations of Basics Functionality

With the example problems in Sec. 2 in mind, including the more detailed top-down discussion of the `PlaneWaveAdvection` example, we describe some of the functionality available in `Basics`. In particular we discuss in bottom-up fashion (i.e. in order of compilation) the code divisions appearing in Fig. 2.

3.1. *VariableManagement*

VariableManagement is the first code division appearing in the diagram on the left side of Fig. 2. It is one of the **Basics** divisions that is not a leaf division. Its subdivisions **Specifiers**, **ArrayOperations**, **ArrayArrays**, and **VariableGroups** are shown in the rightmost diagram of Fig. 2. We discuss each in turn.

3.1.1. *Specifiers*

The first division of **VariableManagement** is **Specifiers**, which contains classes used in the specification of number and character variables. This includes some default parameters to be used as **kind** and **len** specifiers for Fortran intrinsic data types. It also includes mathematical and physical constants and a means of dealing with units.

Consider for example the precision and range of numerical variables, which can be specified with a Fortran **kind** parameter. Note for instance the declaration of some of the members of **ConservationLawEvolutionTemplate**,¹¹ which falls in line 11 in Listing 4:

```
real ( KDR ) :: CourantFactor , StartTime , FinishTime , ...
```

where ... represents additional members not listed here. The parameter **KDR** is a nickname, provided by **Specifiers**, for **KIND_DEFAULT % REAL**, one of the members of the object **KIND_DEFAULT** declared and initialized in **Specifiers**.¹² Fortran **kind** parameters also specify the precision and range of explicit numbers, as in the **2.0_KDR** in line 8 of Listing 6. Normally **KIND_DEFAULT % REAL** (and therefore its nickname **KDR**) is set to correspond to double precision; but if one wanted to execute a program in, say, single precision (and assuming all real variables were declared to be of kind **KDR**), changing this single parameter would accomplish the task.

¹¹As a member of **ConservationLawEvolutionTemplate**, in statements at for instance line 27 of Listing 4, these variables would be referenced as e.g. **CLE % CourantFactor**, etc.

¹²By convention, when we write in all caps a class name that is not an acronym (for instance **KIND_DEFAULT**) it refers to a *singleton*, a special type of object intended to be the only instance of that class ever declared. When we write a class member (e.g. the **REAL** in **KIND_DEFAULT % REAL**) in all caps it denotes a parameter that is intended to remain unchanged throughout program execution. All-caps class members remaining unchanged is at the mercy of convention, except in the case of a singleton object declared with the **parameter** attribute, in which case all of its members automatically remain unchanged.

`Specifiers` also provides means of dealing with units, primarily for initialization and output. These means include the class `MeasuredValueForm` and the singleton object `UNIT`. The class `MeasuredValueForm` has a `real` member `Number` and a `character` member `Unit` that specifies a unit of measure (e.g. 'm' for meters).¹³ The members of the singleton `UNIT` are objects of `MeasuredValueForm` for many units of measure.

There is an additional wrinkle when instances of `MeasuredValueForm` are used in particular as units of measure, as opposed to typical measured values. Units of measure make use of an additional `character` member `Label`, not used in typical measured quantities. Consider for instance `UNIT % CENTIMETER` as an example. The need for an extra member `Label` (relative to the primary members `Number` and `Unit` of `MeasuredValueForm`) arises from that fact that in its internal computations on plain `real` variables, `GENASIS` assumes all numbers are given in terms of a single fundamental unit. To be specific, a geometrized system of units is currently employed in which the meter is fundamental. In terms of this fundamental unit, $1 \text{ cm} = 10^{-2} \text{ m}$. The right-hand side of this expression is reflected in the `MeasuredValueForm` members `Number` and `Unit`, i.e. `UNIT % CENTIMETER % Number = 1.0e-2_KDR` and `UNIT % CENTIMETER % Unit = 'm'`; while the `Label` member is associated with the unit in question, i.e. `UNIT % CENTIMETER % Label = 'cm'`. The members `Number` and `Unit` of `MeasuredValueForm` are used to convert a number into or out of `GENASIS`' fundamental unit on input or output, while the `Label` member is used to label the output of a number in the given unit.

We are now in a position to see how initializations with units can be coded directly. (Examples of initialization with units when reading numbers from parameter files or as command line options, and of output with units to the screen or to a file, will be discussed in connection with the `FileSystem`, `Runtime`, and `Display` portions of `Basics` in Sections 3.4, 3.5, and 3.2 respectively.) Note the declaration of the members `Offset` and `Amplitude` in line 10 of Listing 2 for `SineWaveAdvectionForm`. These members, of type `MeasuredValueForm`, are initialized in lines 6-11 of Listing 6. Default, dimensionless values are assigned in lines 6-9. Default values with units could

¹³We use 'unit' in two ways here: first, as the name of a `character` member of `MeasuredValueForm`; and second, as the name of the singleton object `UNIT`. Which sense is meant in a particular expression should be clear in context.

be implemented with some other member of the `UNIT` singleton instead of `UNIT % IDENTITY`; as it is, non-default values, including units if desired, can be set through a parameter file or by command line options as noted in lines 10-11. Arithmetic and assignment operations involving instances of `MeasuredValueForm`, such as in lines 8 and 9, behave in expected ways: the explicit `real` number multiplies the `Number` member on the right-hand side.¹⁴ The multiplication of an explicit number by a member of the `UNIT` singleton puts the result into GENASIS' fundamental working unit, so that it can properly be used in subsequent calculations. (In the assignment of a `MeasuredValueForm` to a `real`, the `Unit` member is simply ignored.)

Finally, we note that mathematical and physical constants can be accessed through the singleton `CONSTANT`. For instance, the number π is referenced by the expression `CONSTANT % PI` in the `Waveform` function hinted at by line 20 of Listing 2 for `SineWaveAdvectionForm`.

3.1.2. *ArrayOperations*

The second division of `VariableManagement` is `ArrayOperations`, which includes some basic operations on arrays. Some of these—such as `Clear` and `Copy`—are merely overloaded names for `subroutines` whose arguments are intrinsic data types stripped of any potentially complicating contexts or attributes that might inhibit compiler optimization.¹⁵ The purpose of these wrappers is to expose elementary variables to the compiler so that it might use fast methods (such as a system call to set values in memory) it might not otherwise employ if, say, the array has the `pointer` attribute, is deeply embedded in a complicated expression involving a sequence of derived `type`

¹⁴These and like results are achieved by overloading of the arithmetic and assignment operators.

¹⁵The modules defining these two operations are `Clear_Command` and `Copy_Command`. We use the suffix `Command` instead of `Form` in naming a module when it is more convenient to simply implement an action, as opposed to a class with both members and methods. Slavish adherence to object-oriented principles would implement such a case as a class with a (perhaps overloaded) method and no members. In the case of pure actions this is more cumbersome than necessary, leading us to revert to a more procedural rather than object-oriented perspective for `Commands`. Therefore the mechanism of overloading in this case is not the Fortran 2003 `generic` statement used in connection with type-bound procedures; instead it is the use, possible since Fortran 90, of a generic `interface` block to define a generic name for a set of specific procedures. Nevertheless the basic idea of the overloading is still the same, i.e. use of a single name to refer to several possible routines, distinguished by and resolved according to their distinct argument lists.

member references, etc. Other array operations have a more substantial purpose, such as `Sort` and `Search`.¹⁶ The `Search` operation, for instance, is used in `PressurelessFluidForm`—an extension of `ConservedFieldsTemplate`—in the fluid dynamics examples in Sec. 2.1. `ConservedFieldsTemplate` has a rank-1 integer array member `iaConserved` containing indices denoting conserved variables. The extension `PressurelessFluidForm` has an integer member `CONSERVED_DENSITY` which is an index or flag whose value identifies that particular variable. In the expression

```
call Search ( CF % iaConserved , CF % CONSERVED_DENSITY, iDensity )
```

these members of an instance `CF` of `PressurelessFluidForm` are the first two arguments—the inputs—of the `Search` command. Upon return, the last argument `iDensity` is set to the index of `CF % iaConserved` for which `CF % iaConserved == CF % CONSERVED_DENSITY`. (In this case equality is sought, but for a general `real` or `integer` array `A` and a corresponding scalar `Value` as the first two arguments, the last argument `iValue` is set upon return such that `A ( iValue ) <= Value < A ( iValue + 1 )`.)

3.1.3. *ArrayArrays*

The next division of `VariableManagement` is `ArrayArrays`, which contains classes that can be used to form arrays of arrays. One application is the construction of so-called ‘ragged arrays.’ In the rank-two case a ragged array can be conceptualized (following the Fortran row-major perspective) as a matrix whose columns have different numbers of elements. This can be useful when one wants an indexed collection of rank-one arrays that are not all the same length. Even if they are all the same length, one might want to conceptualize the data primarily as a collection of rank-one objects rather than as a rank-two array.

For instance, in the fluid dynamics examples in Sec. 2.1, the class `DistributedMeshForm` has a member

```
type ( ArrayReal_1D_Form ), dimension ( 3 ) :: Edge
```

which is a three-element array of 1D arrays, each of which contains the coordinate values of the cell edges in one of the three spatial dimensions. Having an ‘array of arrays’ allows for loops over dimension while simultaneously accommodating different numbers of cells in each dimension, should the user so desire. The class `ArrayReal_1D_Form` has an allocatable rank-one `real`

¹⁶These are defined in the modules `Sort_Command` and `Search_Command`.

array member `Value`; thus, for an instance `DM` of `DistributedMeshForm`, the expression

```
DM % Edge ( iD ) % Value ( iE )
```

is the edge value `iE` in dimension `iD`. The method `Initialize` of `ArrayReal_1D_Form` is overloaded; possible alternative arguments include an array shape, an explicit array to be copied, or another `ArrayReal_1D_Form` to be copied. In addition to `ArrayReal_1D_Form`, `ArrayArrays` includes analogous classes for arrays of other types and ranks.

3.1.4. *VariableGroups*

The final division of `VariableManagement` is `VariableGroups`, which contains some infrastructure we use extensively in handling collections of variables, especially sets of related physical fields. In particular, the class `VariableGroupForm` includes both metadata about the variables (names, units, etc.) and storage for the variable data itself. The data storage is a rank-two `real` array member `Value` of `VariableGroupForm`; the first dimension indexes separate points, and the second dimension indexes individual variables. The ‘points’ enumerated by the first index can mean very different things in different problems. In the fluid dynamics examples in Sec. 2.1, each row represents a cell in the mesh, with the columns representing different physical fields on the mesh. In the molecular dynamics examples in Sec. 2.2, each row represents a different particle, and each column represents a position or velocity component of the particle.

An example from the fluid dynamics classes in Sec. 2.1 is the member

```
type ( VariableGroupForm ) :: Position
```

of class `DistributedMeshForm`. The call

```
call DM % Position % Initialize &
( [ DM % nProperCells + DM % nGhostCells, 3 ], &
  NameOption = 'Position', &
  VariableOption = [ 'Center.X', &
                    'Center.Y', &
                    'Center.Z' ], &
  UnitOption = DM % CoordinateUnit, ClearOption = .true. )
```

initializes this member of an instance `DM` of class `DistributedMeshForm`. The first argument is an array with two elements specifying the shape of the data storage. The first element gives the total number of cells in the section of the distributed mesh assigned to a given process: the number of ‘proper’ cells on which computation is performed, plus the number of ‘ghost’ cells containing data from neighboring processes (needed for differences, derivatives,

etc.). The second element specifies that there be three variables or fields in this variable group. The second, third, and fourth arguments are optional metadata specifications: the name of the variable group, the names of the variables (in this case the position components of the cell centers), and an array of type `MeasuredValueForm` specifying the units associated with each of the three variables. The last optional argument instructs that the data array should be ‘cleared’ (initialized to 0.0).

It is often convenient to use aliases to access the variable data in an instance of `VariableGroupForm`. Consider the following Fortran 2003 `associate` construct¹⁷ in `PlaneWaveAdvectionTemplate` from the examples in Sec. 2.1.1:

```

associate &
  ( X => DM % Position % Value ( :, 1 ), &
    Y => DM % Position % Value ( :, 2 ), &
    Z => DM % Position % Value ( :, 3 ), &
    VX => PF % Value ( :, PF % VELOCITY ( 1 ) ), &
    VY => PF % Value ( :, PF % VELOCITY ( 2 ) ), &
    VZ => PF % Value ( :, PF % VELOCITY ( 3 ) ), &
    N => PF % Value ( :, PF % COMOVING_DENSITY ) )

    !- Statements reference X ( : ), Y ( : ), Z ( : ), etc. as 1D arrays

end associate
```

We encountered the variable group `DM % Position` in the previous paragraph; here the aliases `X`, `Y`, and `Z` can be referenced as 1D arrays for the components of the mesh cell positions. The other aliases correspond to physical fields in an instance `PF` of `PressurelessFluidForm`; this is an extension of `ConservedFieldsTemplate`, which in turn is an extension of `VariableGroupForm`. Thus `PressurelessFluidForm` retains all the functionality of `VariableGroupForm`, and can be directly used in a call to any procedure with a polymorphic `VariableGroupForm` argument, while also adding members and methods pertaining to this particular group of physical variables. The members `PF % VELOCITY` and `PF % COMOVING_DENSITY` are integers whose values identify the columns in an instance of `PressurelessFluidForm` containing those variables. Use of the aliases given above for geometric and physical fields greatly simplifies expressions used in setting initial conditions in `PlaneWaveAdvectionTemplate`.

¹⁷This aliasing mechanism, new to Fortran 2003, avoids the need to declare pointer variables to be used as shorthand names and allows the compiler to produce more efficient code.

An instance of `VariableGroupForm` can have its own storage, as in the `Position` member of `DistributedMeshForm` above; or point to existing storage, perhaps for the purpose of selecting only a subset of variables from an existing variable group. For instance, in the fluid dynamics examples in Sec. 2.1, the size of output files can be reduced by only writing a few select fluid variables rather than all of them. Consider the initialization

```
call PF % Output % Initialize &
    ( PF, SelectedOption = [ PF % COMOVING_DENSITY, PF % VELOCITY ], &
      VectorOption = [ 'Velocity' ], &
      VectorIndicesOption = VectorIndices )
```

of a `VariableGroupForm` member `Output` of an instance `PF` of `PressurelessFluidForm`. As we have seen previously, overloading of the `Initialize` method allows variation in behavior. In this case the object `PF % Output` uses `PF` itself (the first argument) as the basis of its initialization, and in fact will set a pointer to `PF`'s existing data storage rather than allocate its own and perform a copy operation. But as the second argument suggests, only the listed 'primitive' variables are 'selected' in the new variable group `PF % Output`; conserved variables and any auxiliary variables in `PF` are to be omitted for output purposes. The GENASIS I/O classes (see Sec. 3.4) to which `PF % Output` is ultimately sent will then act only on the 'selected' variables. (This example also illustrates the optional specification of 'vectors' in a variable group, through arguments specifying the names and index triplets of the variables composing the vector components, for the purpose of generating vector field visualizations.)

One may wonder why the trouble should be taken to use a class like `VariableGroupForm` rather than work directly with explicitly declared arrays of variables. One reason is that individual variable storage for a field on an irregular, unstructured, and/or adaptive multidimensional mesh cannot reside in a rank-2 or rank-3 array isomorphic to a logically Cartesian grid, as would be possible with a simple single-level rectangular mesh. Another reason is that our simple `VariableGroupForm` construct greatly simplifies and renders more generic many tasks. For instance, as just mentioned, the file system facilities described later are written in terms of variable groups, so that a user need not program such operations in detail for the many sets of variables that may arise in different problems. Other examples discussed in future papers include the exchange of ghost cell data in multidimensional meshes, and the prolongation and restriction of data in multilevel meshes, which also are handled in terms of variable groups. In writing versatile and

reusable code it is much better to be able to simply set up a variable group rather than have to continually rewrite (for example) explicit ghost exchange or I/O library calls for the many different variables that may appear in, or be added to or removed from, various applications.

Finally, we mention a couple of other classes in the code division `VariableGroups`. A class `PackedVariableGroupForm` can be used to collect selected rows and columns of a `VariableGroupForm` into contiguous memory for more efficient processing. Another class, `VariableGroupArrayMetadataForm`, contains metadata about an array of variable groups, for instance the numbers of variables in each group as well as the total number of variables in all groups.

3.2. *Display*

After `VariableManagement` comes the leaf division `Display` (see the diagram on the left side of Figure 2), which contains infrastructure for displaying messages and variables to the standard output in a uniform and orderly way. The two central features available through `Display` are the singleton object `CONSOLE`, which has members that are used to control output to the standard display ('stdout'); and the heavily overloaded routine `Show`,¹⁸ which manages the hassles of displaying various data types with their respective formatting.

One issue that arises in a distributed-memory parallel computing environment is that, absent intervention by the programmer, output emerges from all processes in an asynchronous manner. This typically results in unordered output of unwanted redundancy. The use of `if` statements can avoid this by allowing only a particular process to output to the screen, but a proliferation of `if` statements with every `print` statement becomes tedious (and difficult to change for debugging purposes if a global variable specifying the process to be displayed is not used). Thus `CONSOLE` has a member `DisplayRank` used by the `Show` command to display output only from the process of the chosen rank, without the programmer having to continually supply `if` statements. The `DisplayRank` member of the `CONSOLE` singleton can be set through its method `SetDisplayRank`. Alternatively, instead of calling `SetDisplayRank` one can supply an optional argument `DisplayRankOption` to the `Show` command specifying, for that call only, which process should be visible.

Another issue related to output is that different pieces of information are

¹⁸Defined in the module `Show_Command`.

of different levels of interest. In a production run one might want minimal output, while more verbose output may be useful for debugging. `CONSOLE` has a member `Verbosity` that addresses this need. It can be set to one of several levels denoted, in descending order of importance, by the `CONSOLE` members `ERROR`, `WARNING`, `INFO_1`, ..., `INFO_7`. In a call to `Show` one of these same members can be given as an optional argument `IgnorabilityOption`; if the importance is thus tagged as less than `Verbosity`, the output is suppressed. In the absence of `IgnorabilityOption` an importance of `INFO_1` is assumed.

The `Show` command provides a very simple interface for a wide range of formatted and labeled output. There are provisions for intrinsic data types, objects of `MeasuredValueForm`, and arrays of all these. An object of `MeasuredValueForm` can be supplied to modify the units (including the unit label itself) in which `real` or `MeasuredValueForm` numbers are displayed. Consider for instance the statements

```
call Show ( DM % MinCoordinate, 'MinCoordinate' )
call Show ( DM % MaxCoordinate, 'MaxCoordinate' )
```

which would display, as pure numbers (albeit with the given string labels), the `real` members `MinCoordinate` and `MaxCoordinate` of an instance `DM` of `DistributedMeshForm`. But we saw in Sec. 3.1.4 that `DistributedMeshForm` has a member `CoordinateUnit` of type `MeasuredValueForm`; this can be used in the calls

```
call Show ( DM % MinCoordinate, DM % CoordinateUnit, 'MinCoordinate', &
           CONSOLE % INFO_3 )
call Show ( DM % MaxCoordinate, DM % CoordinateUnit, 'MaxCoordinate', &
           CONSOLE % INFO_3 )
```

in order that the output be converted and appropriately labeled according to the unit specified by `DM % CoordinateUnit`. In these latter `Show` statements we have also added an ignorability argument. The output will only appear from the process of rank `CONSOLE % DisplayRank`, and only if `CONSOLE % Verbosity` is high enough to include `INFO_3`.¹⁹

3.3. MessagePassing

The next division of `Basics` is `MessagePassing` (see the diagram on the left side of Fig. 2), which contains some classes that abstract the data and

¹⁹The setting of `CONSOLE % DisplayRank` and `CONSOLE % Verbosity` to something other than their defaults is done via a user-specified parameter file (see Section 3.4) and/or command-line arguments (see Section 3.5).

1
2
3
4
5
6
7
8
9 methods useful for working in a message passing parallel computing envi-
10 ronment. As shown in the middle right diagram in Fig. 2, it has three
11 subdivisions: `MessagePassingBasics`, `PointToPoint`, and `Collective`.
12

13 Before describing the leaf divisions under `MessagePassing` we note that
14 one may ask why we bother to create and use message passing classes
15 when MPI (Message Passing Interface) [8] is well-established, widely used,
16 and seems likely to remain the dominant library for message passing in
17 distributed-memory applications. Our message passing classes do indeed use
18 MPI ‘under the hood,’ but in principle could be given other back ends (such as
19 the lower-level Cray GNI application programming interface) without mod-
20 ifying their references in higher-level code. Our interfaces are streamlined
21 relative to direct MPI calls while providing methods for common use cases
22 relevant to our system, and thus provide a simpler experience for the user.
23 In terms of object-oriented design patterns [9], our `MessagePassing` classes
24 provide a ‘façade’ that shields the user from concerns such as the kind and
25 size of data types being sent and received (e.g. single precision vs. dou-
26 ble precision, and beyond, in the case of `real` data) and declaring many
27 variables—handles, buffers, tags, and so on—for use in MPI calls. In our
28 approach many of these needed variables are encapsulated in the classes we
29 define. Finally, we note that the MPI library is mostly procedural in orien-
30 tation and uses Fortran 77 semantics for its routines, sometimes requiring
31 more arguments than necessary if the equivalent modern Fortran interfaces
32 were available (though the MPI-3 standard, which is not yet in wide use at
33 this writing, may improve on this). Our interfaces relieve the user of these
34 burdens.
35
36
37
38
39
40
41

42 *3.3.1. MessagePassingBasics*

43 We mention two classes in this division: `CommunicatorForm` and
44 `PortalHeaderForm`.
45

46 Inspired (obviously) by the notion of an MPI communicator, an instance
47 of `CommunicatorForm` connects a group of processes in a distributed-memory
48 parallel program. The members of `CommunicatorForm` include its `Handle`,
49 `Size`, and `Rank`. Its method `Initialize` is overloaded to provide for ei-
50 ther starting the message passing system, i.e. initializing MPI and thus
51 creating the ‘global’ or ‘world’ communicator; or for creating a ‘subcommu-
52 nicator’ that connects only a subset of the processes connected by a parent
53 communicator. The method `Synchronize` pauses all processes until they
54 reach the call to this method (i.e. an MPI ‘barrier’). Another method
55
56
57
58
59
60
61
62
63
64
65

of `CommunicatorForm` is `Abort`, which provides an emergency exit from distributed-memory program execution. Normal termination of the message passing system (i.e. finalization of MPI) occurs when the `CommunicatorForm` object representing the global communicator goes out of existence.

The purpose of `PortalHeaderForm` is to have an object that handles some basic information about data exchange with a set of processes, such as the numbers of source and target processes; arrays containing the ranks of those processes; and arrays containing the numbers of ‘chunks’ of data to be received from and sent to those processes.

3.3.2. *PointToPoint*

A single message from one process to another is a ‘point-to-point’ communication. The division `PointToPoint` provides classes for single incoming and outgoing messages of different intrinsic data types, as well as for arrays of such messages. Here we illustrate methods for arrays of messages, used in the exchange of ghost cell data in the `DistributedMeshForm` class of the fluid dynamics examples in Sec. 2.1. This class includes the members

```
type ( IncomingMessageArrayRealForm ), allocatable :: &
    IncomingPrevious, IncomingNext
type ( OutgoingMessageArrayRealForm ), allocatable :: &
    OutgoingPrevious, OutgoingNext
```

used to receive data from and send data to processes owning the ‘previous’ and ‘next’ bricks of cells in each dimension.

By way of illustration, the call

```
call DM % IncomingPrevious % Initialize &
    ( DM % Communicator, spread ( TAG_IN_PREV, 1, PHP % nSources ), &
      PHP % Source, PHP % nChunksFrom * VM % nVariablesTotal )
```

initializes the first of these message array members. The first argument is an object of `CommunicatorForm`. The second argument is an array of tags labeling incoming messages from ‘previous’ processes, and the third argument is an array of the ranks of the processes from which messages are to be received; in both of these members of an instance `PHP` of `PortalHeaderForm` (see Sec. 3.3.1) are used. The final argument is an array containing the number of values to be received from each process, calculated from another member of `PHP` and a member of an instance `VM` of `VariableGroupArrayMetadataForm` (Sec. 3.1.4). Here `PHP % nChunksFrom` is an array of the numbers of ghost cells from each source process, and the scalar `VM % nVariablesTotal` is the total number physical variables in the array of variable groups to be exchanged. The call

```

9      call DM % IncomingPrevious % Receive ( )

```

posts receives for all the incoming messages from ‘previous’ processes. Similar calls for **DM % IncomingNext** initialize and post receives for the incoming messages from ‘next’ processes.

An example of the initialization of an outgoing message array is

```

16     call DM % OutgoingPrevious % Initialize &
17         ( DM % Communicator, spread ( TAG.OUT.PREV, 1, PHP % nTargets ), &
18           PHP % Target, PHP % nChunksTo * VM % nVariablesTotal )

```

which is analogous to the above example for an incoming message array, but with targets replacing sources, etc. Element **iE** of the rank-1 buffer to be sent to a target process associated with the index **iD** is set with a statement of the form

```

25     DM % OutgoingPrevious % Message ( iD ) % Value ( iE ) = ...

```

where ... is an expression for the scalar value to be sent. With all elements of the buffer set, the call

```

29     call DM % OutgoingPrevious % Send ( iD )

```

sends the individual message **iD**. (All messages in an outgoing array can be sent by a similar call without a target index, but it is typically advantageous to load the buffer for a single message and then send it right away before moving to the next message.) Similar calls for **DM % OutgoingNext** initialize and send messages to the ‘next’ processes.

A ‘wait’ must be executed before data can be retrieved from a message for which a receive has been previously posted. In this case, the call

```

40     call DM % IncomingPrevious % Wait ( iD )

```

waits for the message from the ‘previous’ process specified by the index **iD**. (In many cases it will be advantageous to use a different overloading we provide for the **Wait** method, the ‘wait any’ variant which returns the index of the first message whose data is available to read from its buffer.) Upon return from a wait call, buffer values are stored into appropriate variables with a statement of the form

```

50     ... = DM % IncomingPrevious % Message ( iD ) % Value ( iE )

```

where ... represents a scalar variable to which is to be assigned element **iE** of the incoming buffer from message **iD**. Similar calls for **DM % IncomingNext** wait for and extract values from messages from the ‘next’ processes. Finally, perfunctory waits

```

9      call DM % OutgoingPrevious % Wait ( )
10     call DM % OutgoingNext % Wait ( )

```

for the outgoing messages to finish sending complete the exchange, here in the version that waits for all messages at once.

3.3.3. *Collective*

A communication that involves a group of processes is said to be ‘collective’. The division **Collective** contains classes that provide for collective communications involving different intrinsic data types. The collective communication class for a given data type is generic in the sense that its members are sufficient for broadcast, gather, scatter, and combined scatter-gather²⁰ communications, and for collection operations (i.e. reductions of data); thus all of these are simply implemented as methods of a single class. For the gather and reduction methods, the absence or presence of the optional argument **RootOption** in the initialization of a collective communication object determines whether or not the gathered or reduced value is distributed to all processes.²¹ This option must also be used in connection with broadcast and scatter calls: by definition, these communications involve a ‘root’ process that is the source of data to be sent to all the processes in the group.

As an example of a collective communication in our fluid dynamics examples in Sec. 2.1, the class **ConservationLawEvolutionTemplate** executes a global reduction in determining the time step. The call

```

37     call CO % Initialize ( C, nOutgoing = [ 1 ], nIncoming = [ 1 ] )

```

initializes an instance **CO** of **CollectiveOperationRealForm**. The first argument is an instance of **CommunicatorForm**. The second and third arguments specify the sizes of the outgoing and incoming buffers. (Here only a single value will be involved in the reduction; arrays are used for the **nOutgoing** and **nIncoming** arguments to allow for the possibility of scatter and gather operations with different numbers of elements sent to or received from different processes.) The absence of the **RootOption** means that all processes will receive the reduced value. The statement

```

49     CO % Outgoing % Value ( 1 ) = TimeStepLocal

```

²⁰‘AllToAll’ routines in MPI.

²¹In MPI this corresponds to **MPI_Allgather** vs. **MPI_Gather**, or **MPI_Allreduce** vs. **MPI_Reduce**.

sets the outgoing buffer value as the minimum time step as determined from the data local to a particular process. The global reduction is accomplished by

```
call CO % Reduce ( REDUCTION % MIN )
```

in which the argument specifies, via reference to a member of our REDUCTION singleton, the type of reduction operation to be performed. Finally, the statement

```
CLE % TimeStep = CO % Incoming % Value ( 1 )
```

copies the reduced value to the member TimeStep of the instance CLE of ConservationLawEvolutionTemplate.

3.4. *FileSystem*

As seen in the diagram on the left side of Fig. 2, the next division of Basics is FileSystem, whose classes handle I/O to disk. The middle left diagram in Fig. 2 shows divisions FileSystemBasics, GridImageBasics, CurveImages, StructuredGridImages, UnstructuredGridImages, and PointGridImages. The major classes in FileSystem used to create and interact with files of various kinds have the term ‘stream’ in their names, connoting a source or sink of data flow. The names of classes used to store grids and data intended for visualization and/or checkpoint/restart include the term ‘image,’ which evokes both colloquial and computing connotations (respectively ‘picture’ and ‘a computer file capturing the contents of some computing element—hard drive, system memory, etc.’).

3.4.1. *FileSystemBasics*

In addition to some lower-level functionality, the classes in this division include two kinds of streams.

ParametersStreamForm is used to read scalar or one-dimensional array parameters of various datatypes from a text file, each line of which contains an entry of the form [name]=[value] or [name]=[value1],[value2],.... Units may be attached to real values with a tilde (~) followed by an all-caps string denoting one of the units available in UNIT_Singleton (see Section 3.1.1). For example, a parameter file associated with the ArgonEquilibrium program of Sec. 2.2.1 contains the entries

```
ParticleMass=39.95~ATOMIC_MASS_UNIT
EnergyParameter=1.032e-2~ELECTRON_VOLT
LengthParameter=3.405~ANGSTROM
```

which specify the particle mass and Lennard-Jones potential parameters appropriate for argon. Parameters need not appear in the file in the order searched for by the program; indeed the absence of a parameter, or even the absence of the parameter file, will not in themselves cause a program exit or crash (though the lack of sensible default values for absent parameters certainly might). Our example programs in Sec. 2 do not directly invoke objects of `ParametersStreamForm`, but instead use a higher-level parameter search facility described in Section 3.5 that looks both in a parameter file and at command line options.

A second type of stream in `FileSystemBasics` is `TableStreamForm`, which reads simple two-dimensional tables of data from a text file.

3.4.2. *GridImageBasics*

The remaining divisions in `FileSystem`, beginning with `GridImageBasics`, provide a façade facilitating interaction with sophisticated I/O libraries. In principle different libraries may be implemented as ‘back ends.’ For this initial release we have implemented the Silo library²² as the back end (see Sec. 4), which is convenient for use with the visualization package VisIt.²³

The major class of interest in this division is `GridImageStreamForm`, which handles some basic aspects of I/O that are the same regardless of the type(s) of grid(s) to be written. Its members and methods allow one to, for instance, open (for reading or writing) or close a file; make a directory (virtual within a file or actual in the file system, which may depend on the I/O library back end); list an existing file’s contents; and automatically track filename numbering for consecutive write operations (e.g. for periodic output in time-dependent evolution). For example,

```
type ( GridImageStreamForm ) :: GridImageStream
```

is one of the members of the class `DistributedMeshForm` used by our fluid dynamics example problems in Sec. 2.1. The method `SetImage` of `DistributedMeshForm` contains the lines

```
associate ( GIS => DM % GridImageStream )
call GIS % Initialize &
( Name, CommunicatorOption = DM % Communicator, &
  WorkingDirectoryOption = OutputDirectory )
call GIS % Open ( GIS % ACCESS_SET_GRID )
```

²²<https://wci.llnl.gov/codes/silo>

²³<https://wci.llnl.gov/codes/visit>

in which `DM` is the passed `DistributedMeshForm` object. Initialization of this stream for parallel I/O includes a name for a file or series of files, an instance of `CommunicatorForm` (see Sec. 3.3.1), and optionally a string specifying an output directory. Once initialized, an object of `GridImageStreamForm` can be opened for various purposes, with the argument used in calling the `Open` method specifying the mode of access to the stream object. In this case the purpose is only to set properties of the grid (see the following subsections); no files are to be written, created, or read with this access mode. With the mode `ACCESS_CREATE`, a file number counter associated with the stream object is incremented and appended to the `Name` argument to form the filename, and then a new file is created with this filename. This access mode also implies read and write access to the newly created file. Other possible access modes are `ACCESS_WRITE`, which opens an existing file for writing by appending new data to the end of the file; and `ACCESS_READ`, which opens an existing file for reading only.²⁴ Once the tasks associated with the stream are finished, the call

```
call GIS % Close ( )
end associate !- GIS
```

flushes any data associated with the stream²⁵ and closes the file.

3.4.3. *CurveImages*

The class `CurveImageForm` is used write a one-dimensional grid and data on it, which can be used to generate curves in a Cartesian xy plane. Data for the 1D plots in Figs. 4 (upper panels), 5 (upper panels), 7 (upper panels), 10, 11, and 14 were generated using this class. An example of its use is the member

```
type ( CurveImageForm ) :: CurveImage
```

of the class `DistributedMeshForm` used by our fluid dynamics example problems in Sec. 2.1. If an instance `DM` of class `DistributedMeshForm` is used as a 1D mesh, the lines

```
associate ( CI => DM % CurveImage )
call CI % Initialize ( GIS )
call CI % SetGrid &
( 'Curves', DM % Edge ( 1 ), DM % nProperCells , &
```

²⁴These access modes are parameter class members of the stream object (see also ⁹).

²⁵Depending on the I/O library and hardware, a call to the write method may only buffer the data and not actually write to the file system until the stream is closed.

```

9          oValue = DM % nGhostLayers ( 1 ), &
10         CoordinateUnitOption = DM % CoordinateUnit ( 1 ) )
11     do iVG = 1, size ( VG )
12         call CI % AddVariableGroup ( VG ( iVG ) )
13     end do
14     end associate !- CI

```

are executed between the opening and closing of the instance of `GridImageStreamForm` (aliased as GIS) just discussed in Sec. 3.4.2, in the method `SetImage` of `DistributedMeshForm`. Initialization requires an instance of `GridImageStreamForm`. The call to `CI % SetGrid` specifies a directory, gives the spatial coordinates of the mesh points, and includes an optional argument specifying the units of those coordinates. The argument `VG` is an array of class `VariableGroupForm` (see Sec. 3.1.4); sending them to the method `CI % AddVariableGroup` makes the variable data and meta-data (names, units, etc.) specified by the variable groups available to the `CurveImageForm` methods for inclusion in the file as supported by the I/O library back end, which in turn allows a visualization package that supports the format to display it directly. (For instance, the units in Fig. 10 were automatically included by the visualization package `VisIt` without intervention of the user beyond including units in the appropriate variable groups in the `ArgonEquilibrium` program.) With these preparations, all that is needed is the call

```

35     call DM % CurveImage % Write &
36     ( TimeOption = TimeOption, CycleNumberOption = CycleNumberOption )

```

which is made in the `DM % Write` method. Again, as supported by the I/O library back end and visualization package, time and cycle number can be optionally saved and displayed.

3.4.4. *StructuredGridImages and UnstructuredGridImages*

The classes `StructuredGridImageForm` and `UnstructuredGridImageForm` are used to write and read two- or three-dimensional grids and data on them, which can be used to generate various kinds of multidimensional plots. Using the member

```

49     type ( StructuredGridImageForm ) :: GridImage

```

of the class `DistributedMeshForm` used by our fluid dynamics example problems in Sec. 2.1, data were generated for the 2D and 3D plots in Figs. 4 (middle and lower panels), 5 (middle and lower panels), and 7 (middle and lower panels). A structured grid may be rectilinear or curvilinear, but is in either case ‘logically rectangular’: the quadrilateral or cuboid cells filling the

domain can be indexed with a pair or triplet of integers, with a single list of coordinates in each dimension sufficing (together with the integer values of an index pair or triplet) to specify the edge values of any particular cell. In an unstructured grid—which can be more irregular in shape—the coordinates of each individual node must be given, together with connectivity information that specifies how the nodes are arranged into cells. Use of structured and unstructured grids follows that just exemplified in Sec. 3.4.3 in connection with curves: a **SetGrid** method provides the information needed to set up a particular kind of grid, an **AddVariableGroup** method attaches dependent variables to the grid, and a **Write** method commits the data to a file.

3.4.5. *PointGridImages*

The class **PointGridImageForm** is used to write a list of positions that can be displayed as points. Using the members

```
type ( PointGridImageForm ) :: GridImagePosition , GridImagePositionBox
```

of the class **DistributedParticlesForm** used by our molecular dynamics example problems in Sec. 2.2, data were generated for the plots in Figs. 9, 12, and 13. (Member **GridImagePosition** is used to plot particles' absolute positions, while member **GridImagePositionBox** is used to plot positions restricted to a periodic box.) Like **DistributedMeshForm** discussed in connection with the curve and structured grid examples discussed in the preceding subsections, **DistributedParticlesForm** also has **SetImage** and **Write** methods. And similar to its curve, structured grid, and unstructured grid counterparts, **PointGridImageForm** has **SetGrid**, **AddVariableGroup**, and **Write** methods. One difference, however, is that **SetGrid** need only be called once in our fluid dynamics examples, in which the mesh remains unchanged throughout the simulation, whereas **SetGrid** must be called before every call to **Write** in the point mesh case because the particle positions change with every time step.

3.5. *Runtime*

The last division of **Basics** is the leaf division **Runtime** (see the diagram on the left side of Fig. 2), which provides some generic functionality associated with running programs. Included are a function that returns wall time, a command that displays memory usage, a command that initializes the random seed, and a class that handles command line options (newly accessible to programs in Fortran 2003).

The user can access most of this functionality through the object `PROGRAM_HEADER` declared in `PROGRAM_HEADER.Singleton`, which requires initialization. For instance, our example program `SineWaveAdvection` contains the lines

```
allocate ( PROGRAM_HEADER )
call PROGRAM_HEADER % Initialize ( 'SineWaveAdvection' )
```

which appear at line 9 of Listing 1. The argument to the `Initialize` method is the program name. By default, `PROGRAM_HEADER` appends a short string indicating the dimensionality (e.g. `'_1D'`, `'_2D'`, etc.) to the program name. Because many applications of GENASIS are expected to be written rather generically for anywhere from one to three space position space dimensions, the default behavior is to determine the dimensionality from a parameter file or command line option, or failing to find these, default to three dimensions. `PROGRAM_HEADER` has the global ('world') communicator as one of its members, and initialization of this singleton takes care of initialization of MPI (see Section 3.3). The `UNIT` (see Section 3.1.1) and `CONSOLE` (see Section 3.2) singletons are also initialized. Once `PROGRAM_HEADER` is initialized its method `ShowStatistics` can be called at any time to display total elapsed time, the amount of time spent in I/O, and current memory usage.

Another `PROGRAM_HEADER` method of significant interest is `GetParameter`. Our general approach to parameters²⁶ is to (1) set a reasonable default value in the code itself, (2) look for an overriding user-specified value in a parameter file, and (3) look for a higher-priority user-specified value among any command line options. The `PROGRAM_HEADER % GetParameter` method follows this philosophy: it informs the user of the existing (presumably default) value, and then sequentially looks for and resets that value if it is found in a parameter file and/or on the command line (with appropriate notifications along the way). The parameter file searched by default is one whose filename consists of the program name with the suffix `'_Program_Parameters'` appended; alternatively, a `ParametersStream` object (see Section 3.4) associated with some other parameter file can be supplied as an optional argument. Our fluid and molecular dynamics example

²⁶In this context we use the term 'parameter' to mean a user-specified value that characterizes a problem or specifies some aspect of program execution. This usage is not quite as restrictive as the precise meaning of the Fortran `parameter` keyword; but most of the time it ends up obeying the same spirit in practice, i.e. denoting some value that remains unchanged during program execution.

problems in Sec. 2 make heavy use of this functionality. For instance, there is a file `ArgonEquilibrium_3D_Program_Parameters` automatically found and read by our example program `ArgonEquilibrium`. In Sec. 3.4.1 we showed three lines from that file, specifying the particle mass and Lennard-Jones potential parameters appropriate for argon. This file also contains the default number density and temperature values

```
NumberDensity=7.599e-3~NUMBER_DENSITY_ANGSTROM
TargetTemperature=359.4~KELVIN
```

which correspond to the gas phase. The program can be run with alternate values, such as those listed in Table 1 for the liquid and solid phases, either by changing the entries in the parameter file, or by including expressions like the above as command line options.

4. Building Examples and Unit Tests

GENASIS Basics is distributed as a gzip-compressed `.tar` file. Upon uncompression and extraction, the top-level directory has `README` and `LICENSE` files and three subdirectories: `Build`, `Modules`, and `Programs`. The directory structures inside `Modules` and `Programs` are as described in Sec. 1. In particular, the example programs described in Sec. 2 and the unit test programs are available under the `Programs/Examples` and `Programs/UnitTests` subdirectories, respectively.

A machine-specific `Makefile` is needed to build GENASIS programs. Several sample `Makefiles` are provided under the subdirectory `Build/Machines`. Minor modifications of one of the provided `Makefiles` that most approximates one's computing environment is often sufficient to get started. The essential information needed includes the name of the compiler wrapper to compile MPI-based code (e.g. commonly `mpif90` for Fortran), compiler-specific flags for various debugging and optimization options, and the flags and locations to include and link with third-party libraries such as Silo.

Once the machine-specific `Makefile` is set up, the environment variable `GENASIS_MACHINE` has to be set to tell the GENASIS build system to use the corresponding `Makefile`. For example, to use the `Makefile` for the GNU compiler on a Linux machine (i.e. `Makefile.Linux.GNU`), in a Bash Unix shell one can type

```
> export GENASIS_MACHINE=Linux_GNU
```

In most common computing environments with a generic MPI library, the fluid dynamics programs described in Sec. 2.1 can then be built and executed (here with 8 MPI processes) with the following commands:

```
> cd Programs/Examples/Basics/FluidDynamics/Executables
> make PURPOSE=OPTIMIZE
> mpirun -np 8 ./SineWaveAdvection_Linux_GNU \
nCells=128,128,128
> mpirun -np 8 ./SawtoothWaveAdvection_Linux_GNU \
nCells=128,128,128 nWavelengths=2,2,2
> mpirun -np 8 ./RiemannProblem_Linux_GNU nCells=128,128,128 \
FinishTime=0.25
```

(To compile in a manner that is unoptimized but useful for debuggers, replace `PURPOSE=OPTIMIZE` with `PURPOSE=DEBUG`. Or omit it altogether; in the absence of a specification of `PURPOSE`, the Makefile in `FluidDynamics/Executables` sets `PURPOSE=DEBUG` as a default.) Note that in these examples, the optional non-default parameter values for `nCells`, `nWavelengths`, and `FinishTime`—which were used in generating the lower panels of Figures 4, 5, and 7—are passed to the programs in this case via command-line options. The 1D and 2D cases of these programs—which were used in generating the upper and middle panels of Figures 4, 5, and 7—can also be executed by specifying fewer elements for `nCells`, for example

```
> mpirun -np 2 ./RiemannProblem_Linux_GNU \
nCells=128 Dimensionality=1D FinishTime=0.25
> mpirun -np 4 ./RiemannProblem_Linux_GNU \
nCells=128,128 Dimensionality=2D FinishTime=0.25
```

where the `Dimensionality` option here is only used as an appendix to the name of the output file (it should be consistent with the number of elements given to `nCells`, which the program uses to determine the desired dimensionality of the mesh).

By default the output files are written in the directory `Output` that resides on the same level as the `Executables` directory, but this can be changed with an optional `OutputDirectory` command line option.

If the VisIt visualization package is available, animated versions of plots similar to Figs. 4, 5, and 7 can be generated using the supplied visualization script called from the `Output` directory. The script takes one argument, which

is the program name appended with the `Dimensionality` string. Assuming the executable `visit` is available, the visualization script can be called, for example, as follows:

```
> cd Programs/Examples/Basics/FluidDynamics/Output
> visit -cli -s ../PlaneWaveAdvection.visit.py \
SineWaveAdvection_3D
> visit -cli -s ../PlaneWaveAdvection.visit.py \
SawtoothWaveAdvection_2D
> visit -cli -s ../PlaneWaveAdvection.visit.py \
RiemannProblem_1D
```

The molecular dynamics programs described in Sec. 2.2 can be built and executed in a manner similar to those in `FluidDynamics`. The directory `MolecularDynamics` is also found under `Programs/Examples/Basics`. A blanket `make` command in the `Executables` subdirectory compiles both `ArgonEquilibrium` and `ClusterFormation`. For both programs, all results presented in Sec. 2.2 were obtained with parameters `nSteps=10000` and `nWrite=1000`. The various numbers of particles and processes used for different runs mentioned in Sec. 2.2 are specified in that section. In the case of `ClusterFormation`, the number of particles is directly specified by a parameter `nParticles`. For `ArgonEquilibrium` a parameter `nUnitCellsRoot` is used instead; the number of particles is $4 * (nUnitCellsRoot ** 3)$. Thus the values 8, 12, 16, and 20 for `nUnitCellsRoot` correspond to 2048, 6912, 16384, and 32000 particles respectively. Specification of the number density and temperature parameters for different phases of argon is discussed in Sec. 3.5.

Unit test programs exercising individual GENASIS classes can similarly be built and executed inside the `Executables` directory of each leaf division of the code under `Programs/UnitTests`. For example, the following commands build and execute the unit test programs for classes in the `Runtime` division (see Section 3.5):

```
> cd Programs/UnitTests/Basics/Runtime/Executables
> make
> mpirun -np 1 [program_name]
```

This blanket `make` builds all the unit test targets in the `Makefile` fragment `Programs/Basics/Runtime/Makefile_Runtime`. Individual targets of course also can be built.

GENASIS **Basics** has been tested with the following compilers: GNU Fortran (gfortran, part of GCC) 4.9.2, Intel Fortran 15, NAG Fortran 5.3.1, and Cray Compiler Environment 8.2.5. Newer versions of these compilers are likely to work as well. GENASIS **Basics** is written in full compliance with the Fortran 2003 standard to enhance portability.

5. Conclusion

Our main purpose in this paper is to describe, make available, and illustrate with examples the **Basics** division of GENASIS, which contains some utilitarian functionality for large-scale simulations on distributed-memory supercomputers. The content of **Basics** is outlined in Fig. 2. The first division, **VariableManagement**, contains several subdivisions shown in the right diagram of Fig. 2. **Specifiers** contains classes used in the specification of number and character variables, as well as mathematical and physical constants and a means of dealing with units. **ArrayOperations** includes some basic operations on arrays. **ArrayArrays** contains classes that can be used to form arrays of arrays; one application is the construction of so-called ragged arrays. **VariableGroups** contains classes we use extensively in handling collections of variables, especially sets of related physical fields. After **VariableManagement** in the left diagram of Fig. 2 comes **Display**, which contains infrastructure for displaying messages and variables to the standard output in a uniform and orderly way. The classes in **MessagePassing**, found in its subdivisions **MessagePassingBasics**, **PointToPoint**, and **Collective** (see the middle right diagram in Fig. 2), abstract the data and methods useful for working in a message passing parallel computing environment. **FileSystem** is next in the left diagram of Fig. 2; the classes in **FileSystemBasics**, **GridImageBasics**, **CurveImages**, **StructuredGridImages**, **UnstructuredGridImages**, and **PointGridImages** (see the middle left diagram in Fig. 2) handle I/O to disk, in part by providing a façade facilitating interaction with sophisticated I/O libraries. Finally, **Runtime** provides a **PROGRAM_HEADER** object giving access to such facilities such as a function that returns wall time, a command that displays memory usage, and a method of reading input from parameter files or the command line.

We provide and discuss several nontrivial example programs in order to illustrate usage of the functionality available in **Basics**, as well as our use of object-oriented design principles using the features of Fortran 2003 that

support this programming paradigm [e.g. 2, 10]. Fluid dynamics examples include plane wave advection (with two different waveforms) and a Riemann problem, in one, two and three spatial dimensions. Molecular dynamics examples include the solid, liquid, and gas phases of argon and the formation of a cluster via gravitational collapse. The object-oriented design principles we follow promote code reusability by facilitating generality, extensibility, and maintainability. In our context, an object-oriented approach enables the flexibility connoted by the ‘General’ in GENASIS—the capacity of the code to include and refer to multiple algorithms, solvers, and physics and numerics choices with the same abstracted names and/or interfaces.

By way of applying the object-oriented principles of abstraction, encapsulation, and decoupling, we have established conventions for implementing classes via Fortran **modules** and derived **types**. These conventions are illustrated in Listings 2 and 3 from the **SineWaveAdvection** example. A single **module** containing a single derived **type** definition comprises a class. The names of the module and the derived type correspond to each other (up to an extra underscore preceding the suffix in the module name). Derived types with data components have been available since Fortran 90; these data components are the *members* of the class. But derived types with procedure components are new to Fortran 2003; these procedure components are the *methods* of the class. The language allows for two kinds of procedure components: procedure pointer variables, and type-bound procedures. We generally use the latter. Type-bound procedures are declared in the **contains** section of the derived type definition, while the subroutine or functions themselves are placed in the **contains** section of the module.

Features new to Fortran 2003 also allow us to implement the principles of inheritance and polymorphism. The **extends** keyword allows a new derived type (a child) to directly inherit the members and methods of a previously defined derived type (the parent), so that these components can be referenced directly as if they had been declared in the new child class itself. The new derived type can add additional members and methods to those already available through the parent type. The child also can modify methods of the parent in two ways. First, with the new **generic** keyword it can *overload* a type-bound procedure name that is resolved to one of multiple procedures depending on the argument list. Second, it can *override* a type-bound procedure with a new subroutine or function exhibiting different behavior than the one in the parent class that it replaces. We also note the new capability of defining a derived type as **abstract**, which allows for **deferred** methods for

1
2
3
4
5
6
7
8
9 which only an interface—and not the subroutine or function itself—are provided. Objects of an abstract type cannot be instantiated (i.e. variables of an abstract type cannot be declared); only objects of an extension of the abstract type, with type-bound procedures that override the deferred methods, can be instantiated.

10
11
12
13
14
15
16 These mechanisms—which prove so useful in developing solvers and physics capabilities for GENASIS—are illustrated in a simple case in Listings 2 and 3 from the **SineWaveAdvection** example. In this problem a particular waveform is advected. But the parent class in Listing 3 is an abstract template that does not depend upon the particular functional form at all. The child class in Listing 2 serves only to specify a particular functional form. Indeed, our example problems also include **SawtoothWaveAdvection**, an alternative extension of the generic plane wave advection template that sets up and advects a different waveform.

17
18
19
20
21
22
23
24
25
26
27 Of course, in this simple example, less sophisticated techniques (such as passing the name of a function as a subroutine argument) could achieve the same purpose; but in more complex and realistic situations, trying to achieve similar generality in Fortran with pre-Fortran 2003 language features fast becomes difficult and unwieldy. Language constructs enabling object-oriented programming provide a much better means of achieving the same sort of code reusability. The point is that the principles of inheritance and polymorphism—exemplified here in the mechanisms of **type** extension and method overriding—make it much easier to allow lower-level code to access higher-level code. This language functionality greatly facilitates, for instance, the separation of the **Mathematics** and **Physics** divisions of GENASIS (to be developed in future releases). For example, solvers for generic classes of equations can be written in **Mathematics**, and then invoked later by a range of different systems whose details are specified in **Physics**. This tremendously enhances the ease and transparency with which one can develop (for example) versatile and widely-applicable solvers.

28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47 Our example programs illustrate the utility of **Basics** for nontrivial problems, and also foreshadow the utility of future releases developing the **Mathematics** and **Physics** portions of GENASIS. Figures 3 and 8 show the classes that compose our fluid dynamics and molecular dynamics examples, illustrating how computational physics problems can be broken down into separate concepts, many of which can be abstracted for reuse. Line counts are by no means a foolproof measure of code significance or human productivity, but they can provide rough initial indications. **Basics**

comprises some 18776 lines of code. Labeling the fluid dynamics example classes as ‘Mathematics’, ‘Physics’, and ‘Problem’ definition according to the color shading in Fig. 3, we have 1836, 1371, and 343/238 (`PlaneWaveAdvection/RiemannProblem`) lines of code respectively. Similarly, for the molecular dynamics example classes in Fig. 8 we have 1132, 1162, and 192/236 (`ArgonEquilibrium/ClusterFormation`) lines for ‘Mathematics’, ‘Physics’, and ‘Problem’ definition respectively. Thus in our example problems, **Basics** takes 84-88% of the line count, the ‘Mathematics’ category takes 5-8%, ‘Physics’ takes 5-6%, and the specific ‘Problem’ definition takes 1-2%. Now, the line counts of the **Mathematics** and **Physics** divisions of GENASIS ultimately will be significantly larger than the counts taken here from our example classes. But already it can be glimpsed that well-designed, reusable code can enable researchers to focus more on their problems and less on computational details.

An object-oriented approach need not ruin code performance. We typically store data in a member of an instance of `VariableGroupForm` (see Sec. 3.1.4), in such a way that reference to that data may at first look intimidating—not only to humans, but perhaps also to compilers! However, the actual operations on that data can be made to look straightforward to both of these audiences. Use of `associate` statements to label individual variables, as exemplified in Sec. 3.1.4, helps make mathematical expressions readable. But beyond this, the floating-point operations on which the bulk of computational effort is (hopefully) spent can be isolated in low-level kernels expressed in terms of elementary loops or array syntax operations on variables of intrinsic data types, which the compiler can handle well. The classes composing our example programs contain a number of such ‘kernel’ subroutines (which in fact we often label with the suffix `Kernel`).

Subsequent papers will describe and make available additions to the **Mathematics** and **Physics** divisions of GENASIS. Some of these will be more specialized towards our focus on an astrophysics application (core-collapse supernovae), but many capabilities will be applicable to other physical problems as well. Of particular initial interest in **Mathematics** is the representation of a ‘manifold’ (or space) as a collection of ‘charts’ (or coordinate patches), in which we approximate the ideal of continuity with a finite sequence of meshes which provide, as necessary, increasing refinements of the coarsest (top-level) mesh. This multilevel mesh structure is the stage upon which the physics plays out. As for solvers, capabilities for nonrelativistic hydrodynamics on a multilevel centrally-refined mesh, with emphasis on test results

rather than code features, already have been reported [1]. An earlier ‘draft version’ of GENASIS featured magnetohydrodynamics (MHD) [11], and was used to study standing accretion shock instability (SASI)-driven magnetic field amplification [12, 13]. In the current version of GENASIS we plan to implement MHD capabilities based on the HLLD solver [14, 15]. For Newtonian gravity, a multilevel Poisson solver will use a distributed FFT solver [16] for the coarsest level, in conjunction with multipole or finite-difference solves on individual levels, in a multigrid approach. Work on general relativistic gravity—a major undertaking—is also underway in GENASIS [17]. A more significant challenge for our own target application of core-collapse supernovae is neutrino transport. Building on our group’s past experience [18, 19], theoretical developments [20, 21], and initial forays into transport in the earliest version of GENASIS [22, 23], we plan to deploy a multigrid approach here as well, using a multigroup Variable Eddington Tensor formulation [24] with closures of increasing sophistication, ultimately culminating in a full ‘Boltzmann solver’ [25].

Acknowledgements

This material is based upon work supported by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research and Office of Nuclear Physics. This research used resources of the Joint Institute for Computational Sciences at the University of Tennessee.

References

- [1] C. Y. Cardall, R. D. Budiardja, E. Endeve, A. Mezzacappa, GenASIS: General Astrophysical Simulation System. I. Refinable Mesh and Non-relativistic Hydrodynamics, *Astrophys. J. Suppl. Ser.* 210 (2014) 17.
- [2] J. Reid, The new features of fortran 2003, *SIGPLAN Fortran Forum* 26 (2007) 10–33.
- [3] M. R. Fahey, R. Budiardja, L. Crosby, S. McNally, Deploying darter - a cray xc30 system, in: J. Kunkel, T. Ludwig, H. Meuer (Eds.), *Supercomputing*, Vol. 8488 of *Lecture Notes in Computer Science*, Springer International Publishing, 2014, pp. 430–439. doi:10.1007/978-3-319-07518-1_28. URL http://dx.doi.org/10.1007/978-3-319-07518-1_28

- [4] G. Sod, A survey of several finite difference methods for systems of nonlinear hyperbolic conservation laws, *Journal of Computational Physics* 27 (1) (1978) 1–31.
doi:10.1016/0021-9991(78)90023-2.
URL <http://linkinghub.elsevier.com/retrieve/pii/0021999178900232>
- [5] J. M. Thijssen, *Computational Physics*, 2nd Edition, Cambridge University Press, 2007, Ch. 8.
- [6] T. Pang, *An Introduction to Computational Physics*, 2nd Edition, Cambridge University Press, 2006, Ch. 8.
- [7] P. Bodenheimer, G. P. Laughlin, M. Rozyczka, T. Plewa, H. W. Yorke, *Numerical Methods in Astrophysics: An Introduction*, CRC Press, 2007, Ch. 3.
- [8] W. Gropp, E. Lusk, A. Skjellum, *Using MPI - 2nd Edition: Portable Parallel Programming with the Message Passing Interface (Scientific and Engineering Computation)*, 2nd Edition, The MIT Press, 1999.
- [9] E. Gamma, R. Helm, R. Johnson, J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley Professional, 1994.
- [10] J. C. Adams, W. S. Brainerd, R. A. Hendrickson, R. E. Maine, J. T. Martin, B. T. Smith, *The Fortran 2003 Handbook: The Complete Syntax, Features and Procedures*, Springer, 2008.
- [11] E. Endeve, C. Y. Cardall, R. D. Budiardja, A. Mezzacappa, Turbulent magnetic field amplification from spiral sasi modes in core-collapse supernovae, *Journal of Physics: Conference Series* 402 (1) (2012) 012027.
- [12] E. Endeve, C. Y. Cardall, R. D. Budiardja, A. Mezzacappa, Generation of Magnetic Fields by The Stationary Accretion Shock Instability, *The Astrophysical Journal* 713 (2) (2010) 1219–1243.
- [13] E. Endeve, C. Y. Cardall, R. D. Budiardja, S. W. Beck, A. Bejnood, R. J. Toedte, A. Mezzacappa, J. M. Blondin, Turbulent Magnetic Field Amplification from Spiral SASI Modes: Implications for Core-collapse Supernovae and Proto-neutron Star Magnetization, *Astrophys. J.* 751 (2012) 26.

- [14] T. Miyoshi, K. Kusano, A multi-state HLL approximate Riemann solver for ideal magnetohydrodynamics, *Journal of Computational Physics* 208 (1) (2005) 315–344.
- [15] A. Mignone, M. Ugliano, G. Bodo, A five-wave Harten-Lax-van Leer Riemann solver for relativistic magnetohydrodynamics, *Monthly Notices of the Royal Astronomical Society* 393 (4) (2009) 1141–1156.
- [16] R. D. Budiardja, C. Y. Cardall, Parallel FFT-based Poisson solver for isolated three-dimensional systems, *Computer Physics Communications* 182 (10) (2011) 2265–2275.
- [17] P. Tsatsin, R. Budiardja, C. Cardall, E. Endeve, P. Marronetti, A. Mezzacappa, GenASiS: A full GR-RMHD simulation framework: overview, goals, and preliminary tests, in: *APS Meeting Abstracts*, 2011, p. 12006.
- [18] M. Liebendörfer, O. E. B. Messer, A. Mezzacappa, S. W. Bruenn, C. Y. Cardall, F.-K. Thielemann, A Finite Difference Representation of Neutrino Radiation Hydrodynamics in Spherically Symmetric General Relativistic Spacetime, *Astrophys. J. Suppl. Ser.* 150 (2004) 263–316.
- [19] S. W. Bruenn, A. Mezzacappa, W. R. Hix, J. M. Blondin, P. Marronetti, O. E. B. Messer, C. J. Dirk, S. Yoshida, 2D and 3D core-collapse supernovae simulation results obtained with the CHIMERA code, *Journal of Physics: Conference Series* 180 (2009) 012018.
- [20] C. Cardall, A. Mezzacappa, Conservative formulations of general relativistic kinetic theory, *Phys. Rev. D* 68 (2) (2003) 023006.
- [21] C. Y. Cardall, E. J. Lentz, A. Mezzacappa, Conservative special relativistic radiative transfer for multidimensional astrophysical simulations: Motivation and elaboration, *Phys. Rev. D* 72 (4) (2005) 043007.
- [22] C. Y. Cardall, A. O. Razoumov, E. Endeve, A. Mezzacappa, The Long Term: Six-Dimensional Core-Collapse Supernova Models, in: A. Mezzacappa, G. M. Fuller (Eds.), *Open Issues in Core Collapse Supernova Theory*, 2005, pp. 196–218.
- [23] C. Y. Cardall, An approach to neutrino radiative transfer in supernova simulations, in: G. Kanschä, E. Meinköhn, R. Rannacher, R. Wehrse

(Eds.), Numerical Methods in Multidimensional Radiative Transfer, Springer, Berlin Heidelberg, 2009, pp. 27–37.

[24] C. Y. Cardall, E. Endeve, A. Mezzacappa, Conservative 3+1 general relativistic variable Eddington tensor radiation transport equations, Phys. Rev. D87 (10) (2013) 103004.

[25] C. Y. Cardall, E. Endeve, A. Mezzacappa, Conservative 3+1 general relativistic Boltzmann equation, Phys. Rev. D88 (2) (2013) 023011.