Terschanski *et al.*
Computational Science and Engineering (2026) 3:9
https://doi.org/10.1007/s44207-026-00015-w

Computational Science and Engineering

SOFTWARE Open Access

Bryne: sustainable prototyping of finite element models

Benjamin Terschanski[1*], Robert Klöfkorn[2], Andreas Dedner[3] and Julia Kowalski[1*]

*Correspondence:
benjamin.terschanski@rwth-aachen.de; kowalski@mbd.rwth-aachen.de

[1] Chair of Methods for Model-based Development in Computational Engineering, Faculty of Mechanical Engineering, RWTH Aachen, Eilfschornsteinstraße 18, 52062 Aachen, Germany
[2] Center for Mathematical Sciences, Lund University, Box 117, 221 00 Lund, Sweden
[3] Mathematics Institute, University of Warwick, Coventry, Warwick CV4 7AL, United Kingdom

Abstract

Open-source simulation frameworks are evolving rapidly to provide accessible tools for the numerical solution of partial differential equations. Modern finite element (FEM) software such as FEniCS, Firedrake, or DUNE-FEM alleviates the need for modelers to recode the discretization and linear solver backend for each application and enables rapid prototyping of solvers. However, while it has become easier to build prototype FEM models, creating a solver reusable beyond its specific initial simulation setup remains difficult. Moreover, simulation setups typically cover an ample input parameter space, and tracking complex metadata on research project time scales has become a challenge. This implies the need to supplement model development with a coding-intensive complementary workstream, seldom developed for sustainable reuse. To address these issues, we introduce our open-source Python package bryne. Bryne is an object-oriented framework for FEM solvers built with the DUNE-FEM Python API. In this article, we describe how it helps to evolve rapid-prototyping solver development into sustainable simulation building. First, we show how to translate a minimal DUNE-FEM solver into a bryne FEM model to build human-readable, metadata-enriched simulations. Bryne then offers a simulation driver and model coupling interfaces to combine implemented solvers in operator-split multiphysics simulations. The resulting reproducibility-enabled infrastructure allows users to tackle complex simulation setups without sacrificing backend flexibility. We demonstrate the workflow on a convection-coupled phase-change simulation, where a discontinuous Galerkin flow solver is coupled with a solver for solidification phase change.

Keywords: Research software engineering, Multiphysics, Reproducible science, Finite element method, Solidification phase change

1 Introduction

Modern open-source numerics frameworks provide modelers with powerful tools to build complex simulations. For simulations based on finite element (FEM) discretization a variety of open-source codes such as deal.II [1], Elmer [2], MFEM [2], FEniCS [3], Firedrake [4], DUNE-FEM [5, 6] are under active development, many of which provide a Python API.

Notably, FEniCS, Firedrake, and DUNE-FEM allow users to input the weak form of the partial differential equation (PDE) via the Unified Form Language (UFL) [7]. Alleviating the need to re-implement discretization and linear solvers, they allow for

© The Author(s) 2026. **Open Access** This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made. The images or other third party material in this article are included in the article's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit http://creativecommons.org/licenses/by/4.0/.

Springer

rapid prototyping of models and numerical methods and shift the focus from low-level debugging to numerical modeling. The respective backend then handles the assembly of the linear system and the operator Jacobian via automatic code generation. For performance reasons, the linear solves are typically performed in a lower-level language such as C++. This approach allows for easy experimentation with fundamental simulation components, such as different approximation spaces, solver setups, preconditioners, or even completely new PDE terms. Subsequently, Python-API-driven open-source codes had a significant impact on the computational science community.

While the advent of Python API-based FEM solvers has facilitated initial prototype creation, it inadvertently created a new reproducibility challenge. Here, we introduce the term "sustainable prototyping" to describe the process of moving from a draft solver to a piece of numerical software with two re-user perspectives in mind:

- For the simulation model developer, the code that implements a solver for an FEM model has to be maintainable and extensible.
- For users and other researchers, simulation setups should be fully transparent in a way that does not require deep knowledge of the solver backend.

In practice, there is often a trade-off between the flexibility in rapid-prototyping of new PDE solvers and the degree of sophistication of the simulation setup infrastructure. Figure 1 illustrates a typical monolithic prototype solver. Here, all major simulation components are implemented in a single notebook or Python module. This includes the static model definition and solver logic (*model setup*), as well as the inputs specific to a particular simulation scenario (*simulation setup*). The specific simulation setup includes the mesh type, finite element basis and degree, physical coefficients, material parameters, as well as numerical and solver parameters. The input parameter space can thus become quite large even for a minimal, single-purpose PDE solver.

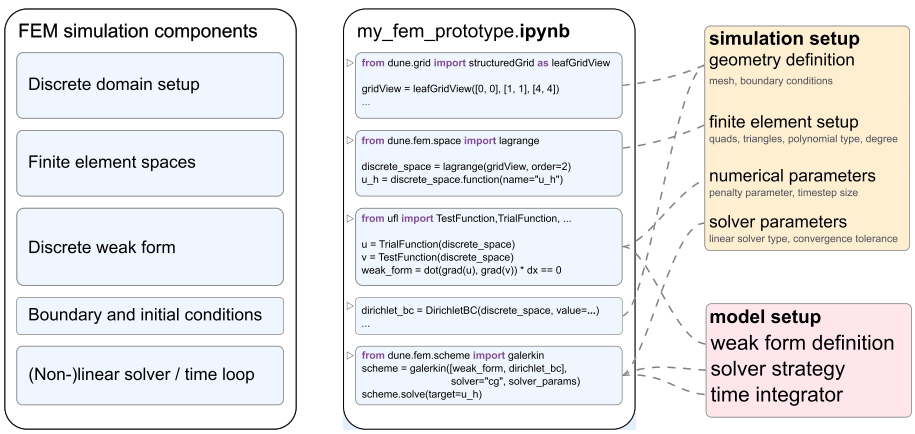

Fig. 1 Sketch of a typical Python FEM simulation notebook (*center*). In a prototyping workflow, users implement the core components of a simulation (*left*) in a single Python module or notebook (*center*). On the right, we distinguish between the *simulation setup*, containing the input parameters or "moving parts" of the simulation, and the *model setup*, which is invariant for a given model

Prototype models and showcase solvers like the one in Fig. 1 are reproducible as functional early samples to prove the feasibility of a simulation method. Reusability is, however, limited in several ways:

1. A tight coupling of solver logic and simulation setup sacrifices flexibility in changing the application scenario.
2. Models can not easily be generalized or combined in multi-physics simulations.
3. Non-standard interfaces and input handling make it hard to compare and archive models and simulation results.

It hence remains challenging to develop simulation codes on top of popular open-source numerical backends that are usable past an initial proof of concept.

Some efforts have been made to address these issues. Probably the most sophisticated project is the commercial software FEATool Multiphysics [8], which implements a simulation builder using the FEniCS backend, including a GUI. The design of FEATool integrates some of the concepts discussed in this paper, but it is not free software and the base paid version does not include the capabilities for user-defined models. Community efforts such as the FEniCS component multiphenicsx [9] focus on the implementation of solver logic for coupled multiphysics simulations, but do not provide a higher-level framework for building reusable FEM models. To the best of our knowledge, no such efforts exist for Firedrake and DUNE-FEM at the time of writing. Subsequently, application developers of open-source research solvers built on FEM backends are forced to re-implement their own simulation infrastructure, hindering the establishment of community best practices for the reproducibility of complex simulation setups. Recent examples can be found using FEniCS [10–13], Firedrake [14, 15] and DUNE-FEM [16, 17].

We therefore identify an infrastructural gap between the Python-API-based discretization tools that enable flexible solver building and the requirements for the sustainable development of complex simulations. Bridging this gap is a core motivation of this work. In this paper, we present bryne, a Python package built on the DUNE-FEM Python API and designed to

- semantically structure and systematically compose prototyped models in a modular way, and to
- improve metadata traceability and hence the reproducibility of complex simulation setups.

Bryne wraps the flexibility of UFL and DUNE-FEM in a solver-building architecture that allows users to move from first prototypes to reusable models.

This is achieved by a modular design, separating the simulation setup from the model definition and solver logic (*see right column of Fig.* 1). In bryne, the definition of a finite element model comprises its weak form, a definition of model-specific parameters, and the solver logic to compute a numerical solution to the model governing PDE. Optionally, the model can also provide a coupling interface to other models, easing data exchange between solvers on the same discrete domain. Bryne

has an object-oriented structure that allows for extension of existing models through inheritance. The package handles simulation inputs through YAML files and user-modifiable Python scripts imported from a directory at runtime. While containing the complete simulation setup, this input directory aims to be easy to navigate and doubles as a human-readable documentation of the simulation setup. To this end, the simulation results are always automatically enriched with the setup metadata, which enables easy parameter traceability and reproduction. To our knowledge, this is the first such open-source package for building reusable FEM models with DUNE-FEM.

It should be noted that bryne does not implement any low-level finite element assembly or linear solver and is, therefore, not a standalone solver library. In the present version of bryne, we rely on the DUNE-FEM package for the finite element discretization and linear solvers. DUNE-FEM in turn supports PETSc as a linear solver backend and has an interface to petsc4py [18, 19]. Weak form definitions in DUNE-FEM are based on the same UFL syntax as other open-source libraries, and many of the concepts presented in the following will translate to other FEM backends with Python bindings. A minimal FEniCS or Firedrake solver will look very similar to Fig. 1, thus it becomes relatively straightforward to extend the bryne architecture to include models written in these libraries. Hence, this work's contribution also lies in the proposition of general simulator components to elevate the general usability of Python API discretization backends.

All major software components are described in Sect. 2, starting with an example application scenario to motivate our design requirements. We spend some time explaining the building blocks of bryne FEM models. A common interface enables model extension through inheritance and data exchange of models on the same grid; this value-add is demonstrated using an example model hierarchy.

In Sect. 3, we demonstrate bryne usage in a research context. We combine a discontinuous Galerkin flow solver with a thermal phase-change solver to build a convection-coupled phase change simulation. Verification results and the solver codes are first published in this article.

2 Bryne software design

In this section, we introduce major components of the bryne software package and how they interact to build a simulation framework on top of the DUNE-FEM Python API.

2.1 Motivational example

To understand the scope of the present work, we start with a simulation example inspired by our research. It serves as a "modeler's perspective" motivation for the development of bryne and doubles as an introduction to the software design.

Imagine we would like to simulate a convection-coupled phase change process, where a solid exposed to a heat source is progressively melting, while the melt is moving due to natural convection. The setup is sketched in Fig. 2, together with a common numerical approach to solve the multiphysics problem in an operator-split fashion with flow and temperature fields solved sequentially [20, 21]. To build the final complex process model, we would have to

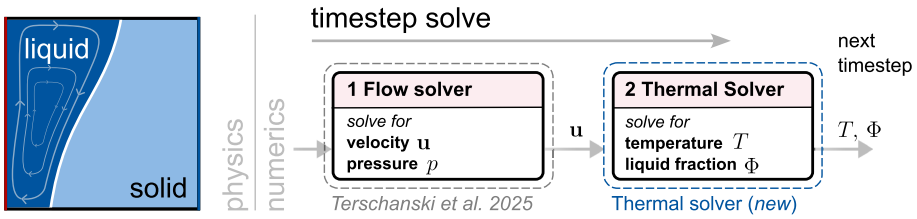

Fig. 2 Phase change problem of a solid melting into a fluid moving due to natural convection (*left*). The sketch on the right shows an example numerical model sequence of a flow solver (*1*) and a two-phase thermal solver (*2*) executed sequentially in an operator-split fashion

1. develop the individual model components, e.g. the flow solver (*(1) in Fig. 2*) to compute velocity and pressure, and the two-phase thermal solver *(2)* to compute the temperature and liquid volume fraction.
2. perform incremental quality assurance for both models, such as verification against analytical and manufactured solutions as well as empirical convergence studies.
3. develop a multi-physics coupling strategy that combines individual model components into a multi-physics solver.
4. embed the resulting multi-physics model into a simulation driver that will allow us to either study the physical process or to validate against data to test admissibility of the model's assumptions.

At all points, we want to be able to experiment with different finite element methods, linear solvers, and mesh types. To perform validation and to study real physical processes, we need to keep track of increasingly complex simulation setups, including material parameters for both phases and numerical parameters for two individual non-trivial finite element solvers. Finally, we would like to publish our results. To build trust in our method and enable independent reproduction, we must share simulation setup metadata with reviewers and readers.

Building upon the flexibility of the DUNE-FEM Python API, bryne provides a single framework to achieve these tasks.

2.2 Overview of software components

Bryne simulations rest on four pillars, illustrated in Fig. 3:

1. The bryne **setup infrastructure**, input and output management introduces a semantically structured input directory. Simulation setups are stored with the results, allowing for re-execution of the exact setup with bryne and documentation of metadata for external reproduction.
2. Simulations are composed of **FEM models** implemented as Python classes sharing a common interface. Each model implements the solver to a particular PDE problem. The object-oriented approach allows extending models through inheritance.
3. The bryne **driver** runs the main simulation time loop. At each time step, it executes a sequence of models in a user-defined coupling sequence.
4. The **coupling logic** allows models defined on the same discrete grid to exchange data. This can be used to combine models in an operator-split fashion.

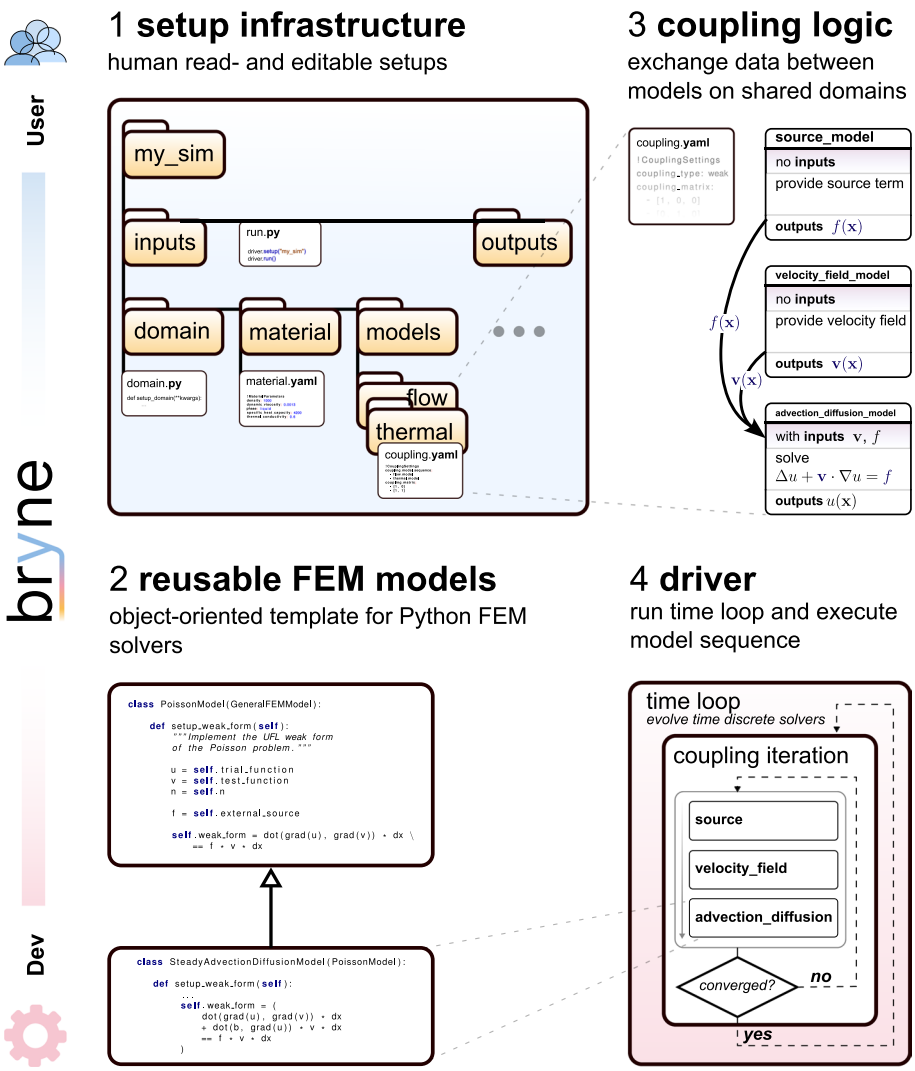

Fig. 3 An overview of core components of the bryne software package. Our package provides four main contributions on top of the DUNE-FEM Python API. A logical split between the core implementation and simulation allows for simulation setups in a semantically structured directory, which can be stored with results to ensure metadata traceability (*1*). Simulations are built from model solvers, which are implemented as classes with a common interface (*2*). Subsequently, developers can declare model inputs and outputs, which allows models on the same grid to exchange data (*3*). The bryne main driver provides a common timestepping routine and executes the model solver sequence in a simulation loop (*4*)

At the heart of bryne lies the implementation of FEM models as Python classes, and we dedicate some space to convey the building blocks of a bryne model.

2.3 Building models with bryne

All following minimal working examples are shipped with the bryne repository [22]. *Executable simulation setups can be found in the* `examples` *directory. Code snippets are excerpts of the respective example files; some are shortened for conciseness. Further details on the simulation input structure can be found in Sec.* 2.10 *to facilitate reading along with the code examples.*

Bryne simulations are built from reusable components, implementing mathematical model solvers as Python classes. In the current version of bryne, we focus on numerical PDE solvers based on finite element discretizations. In this section, we focus on two aspects of reusability,

1. **inheritance**, which allows for efficient extension of existing models and
2. model **coupling** to build complex simulations from individually developed and tested solvers.

Figure 4 illustrates the model hierarchy discussed in this section. Starting from the abstract `Model` and `GeneralFEMModel` classes, a sequence of increasingly complex models (*top to bottom*) can be built by deriving from existing models. Each FEM

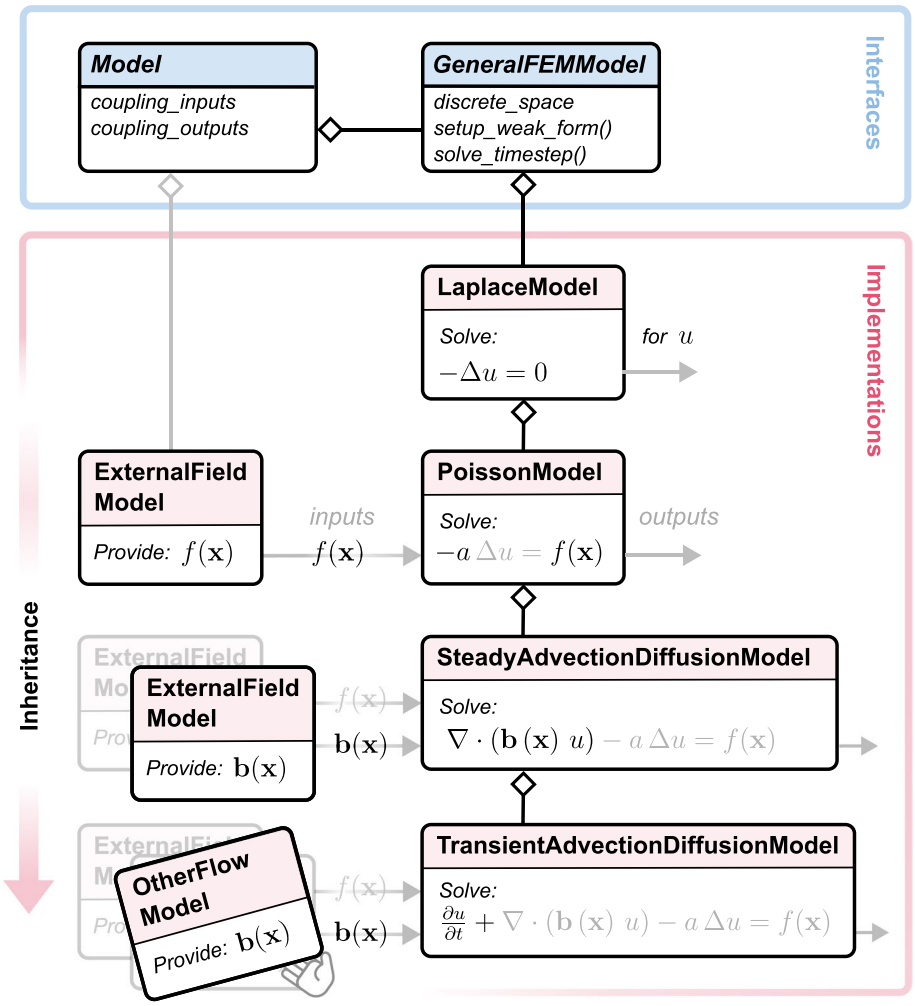

Fig. 4 Sketch of a model cascade through inheritance. Bryne FEM solvers are implemented as classes with a shared interface. Inheritance allows building upon existing models by extending the weak form implementation, coupling interface, and solver logic. The figure sketches the structure of model building examples presented in this article from top to bottom. Starting with the `LaplaceModel`, each row shows one example, where the flow of data through the bryne coupling interface is indicated from left to right

model implements its own weak form definition but common terms, parameters and solver logic can all be inherited. Coupling between models on the same spatial grid can be realized by declaring solver inputs and outputs through a common coupling interface. This way, data can be exchanged in a coupling sequence (*left to right*).

2.3.1 Basic concepts

We start with a simple example to illustrate the idea. Let's say we would like to solve the Laplace equation for scalar u,

$$-\Delta u = 0 \quad \mathbf{x} \in \Omega\,, \tag{1a}$$

$$u = g \quad \mathbf{x} \in \partial\Omega\,, \tag{1b}$$

where $\Omega \in \mathbb{R}^d$ is the d—dimensional domain and g is a constant Dirichlet boundary condition on the entire boundary $\partial\Omega$. In bryne, a finite element model for this PDE seeks to find the numerical approximation u_h of the solution u in a finite element space V_h. A core modeling step in building a bryne FEM model, therefore, is the definition of a particular weak form of the PDE, which is then used to build and solve the linear system using the DUNE-FEM backend [6]. In our first example, the classical variational problem reads

$$\textit{continuous:} \quad \text{Find } u \in V \text{ s.t. } \int_\Omega \nabla u \cdot \nabla v \, d\Omega = 0\,, \qquad \forall\, v \in V_0\,, \tag{2a}$$

$$\textit{discrete:} \quad \text{Find } u_h \in V_h \text{ s.t. } \int_{\Omega_h} \nabla u_h \cdot \nabla v_h \, d\Omega_h = 0\,, \qquad \forall\, v_h \in V_{h,0}\,, \tag{2b}$$

where u and v are continuous trial and test functions living in the spaces V and V_0, respectively, where subscript $\bullet_0$ denotes that test functions vanish on the boundary $\partial\Omega$. We use subscript $\bullet_h$ to denote the discrete approximation to the continuous counterparts and hence u_h, v_h are discrete trial and test functions. Since we want to enable bryne users to change the finite element space as part of the simulation setup, the discrete approximation space V_h is deliberately left unspecified. To build our Laplace model solver in bryne, we can now write a Python class `LaplaceModel` that defines

1. the discrete weak form (2b) and
2. any operations needed in order to solve the linear system resulting from the discrete variational problem.

An excerpt of the `LaplaceModel` class implementation reads like this:

```python
from ufl import TestFunction, TrialFunction, dx, grad, dot
from bryne.models.model import GeneralFEMModel

class LaplaceModel(GeneralFEMModel):
    ...
    def setup_weak_form(self):
        """Implement the UFL weak form of the Laplace problem."""
        u = self.trial_function
        v = self.test_function

        self.weak_form = dot(grad(u), grad(v)) * dx == 0
    ...
    def solve_timestep(self):
        """Implements the actual linear system solve"""
        self.scheme.solve(target=self.u_h)
```

The actual assembly of the linear system and the linear solver logic itself is handled by the numerical backend, which in the current version of bryne uses the DUNE-FEM [6] Python API. DUNE-FEM, in turn, uses the Unified Form Language (UFL) [7] to parse the weak form input and generate problem-specific C++ code. The default quadrature order used during FEM matrix assembly is $2p$ for element integrals and $2p+1$ for surface integrals, where p is the highest polynomial order of the approximation space. Alternatively, DUNE-FEM provides options to customize the quadrature order either globally or for individual UFL expressions.

The UFL syntax in `setup_weak_form()` is shared with FEniCS and Firedrake, which facilitates reading and writing bryne models for users familiar with these libraries. We build on this example class to elaborate on the benefits of this approach over a monolithic Python script in the following sections.

As shown in Fig. 4, our `LaplaceModel` is a subclass of `GeneralFEMModel`, which in turn derives from the abstract `Model` class. Deriving from `GeneralFEMModel`, we inherit attributes and methods shared by finite element solvers implemented with DUNE-FEM. Examples include the discrete test and trial function definitions, the discrete approximation space, and a common handling of user-input boundary conditions. All models implemented in bryne, including `GeneralFEMModel`s, derive from an abstract `Model` class, which defines a common interface for the variables or fields a model computes and the inputs it requires to perform the computation. This design choice allows us to

1. reuse a model in different simulation setups,
2. combine it with other models and
3. to extend the model through inheritance,

once first implemented.

2.3.2 Model cascades through inheritance

The object-oriented structure of bryne models allows us to extend PDE solvers in analogy to the mathematical model hierarchy. To solve a more complex PDE containing the terms we have already written a model class for, we can simply derive a new model class

from the existing one and modify the weak form. A generalization of the initial Laplace model is, for instance, given by the Poisson equation with a spatially varying source term $f(\mathbf{x})$ and Dirichlet boundary conditions,

$$-\Delta u = f(\mathbf{x}) \quad \mathbf{x} \in \Omega\,, \tag{3a}$$

$$u = g \quad \mathbf{x} \in \partial\Omega\,. \tag{3b}$$

Equation (3a) contains the Laplace Eq. (1a) as the special case $f(\mathbf{x}) \equiv 0$. To represent this in code, we build a new model class `PoissonModel` that inherits from `LaplaceModel` and modifies the weak form to include the source term $f(\mathbf{x})$:

```
from bryne.models.laplace_model.laplace_model import LaplaceModel
...
class PoissonModel(LaplaceModel):
    ...
    def setup_weak_form(self):
        ...
        self.external_source = ... # some value or function
        self.weak_form = dot(grad(u), grad(v)) * dx \
            == self.external_source * v * dx
```

where we have rewritten the entire weak form instead of just modifying `self.weak_form` for clarity.

Even in this most simple example, the question now becomes how and where to set the actual value of the source term `external_source`. Naively, we could set it in the `setup_weak_form()` method, but this would introduce a hard dependency on the `PoissonModel` class code, making it hard to reuse the model in different simulation setups. We could try to parse a user input for `external_source`, which also sacrifices flexibility. What if the source term is actually physically given as the solution to another equation, possibly a PDE?

2.4 Combining models: coupling interfaces in bryne

Here we describe how we can add a coupling input to our `PoissonModel` class to allow other models to provide the source term $f(\mathbf{x})$ at simulation runtime. Classes derived from the bryne `Model` class can implement their own coupling interface through three methods,

1. `define_coupling_input` to specify the names and dimensions of the inputs that a model expects.
2. `define_coupling_output` to provide names and dimensions of the outputs that the model provides. Other models, in turn, can use these outputs if they have a matching coupling input.
3. `link_coupling_input` to assign coupling inputs to model attributes.

Using these methods, we can add the source term $f(\mathbf{x})$ as a coupling input to our `PoissonModel` class as follows:

```python
from bryne.models.interface import CouplingInterfaceData

def define_coupling_input(self):
    self.coupling_input_data["external_source_poisson"] = \
        CouplingInterfaceData(
            coupling_data=None, ...
            data_array_dimensions=[1],
            coupling_is_optional=True)
```

Here we specify that the `PoissonModel` expects a coupling input named `external_source_poisson`, which is a scalar field on the discrete domain. By declaring the coupling optional, we allow the model to be run even if no input with the matching name `external_source_poisson` is provided. For the Poisson model, this is reasonable as we can use the Laplace limit $f(\mathbf{x}) \equiv 0$ as a sensible default. Similarly, we can make the solution u_h to the Poisson equation available to other models by declaring it as a coupling output:

```python
def define_coupling_output(self):
    self.coupling_output_data["poisson_scalar_field"] = \
        CouplingInterfaceData(
            coupling_data=self.u_h, ...
            data_array_dimensions=[1])
```

With this setup, we can now combine our `PoissonModel` with any other model that

- provides a coupling output with the key `external_source_poisson`. Such models can be used to provide the source term $f(\mathbf{x})$.
- uses a coupling input with the key `poisson_scalar_field`. Such models can use the solution u_h of the Poisson equation as an input.

To show how to use generic functions defined on the grid as coupling inputs, we now assume that our source term is given by a prescribed scalar function

$$f_{\text{ex},1} = f(\mathbf{x}) = \begin{cases} f_{\text{source}} & \text{if } \|\mathbf{x} - (0.25, 0.25)^T\|_2 < r\,, \\ f_{\text{source}} & \text{if } \|\mathbf{x} - (0.75, 0.75)^T\|_2 < r\,, \\ 0 & \text{else}\,, \end{cases} \tag{4}$$

defined on a unit square domain $\Omega = [0,1]^2$ with $r = 0.25$. This spatially varying source applies a constant value of f_{source} on two circular regions centered around $(0.25, 0.25)^T$ and $(0.75, 0.75)^T$. For functions to be analytically evaluated on the grid, bryne provides the `ExternalField` class, which can provide arbitrary scalar, vector, or tensor fields on the grid with a coupling name defined by the user. For this to work, users have to provide a DUNE-FEM `gridFunction` that takes standard UFL expressions, which allows for complex space and time-dependent function definitions. The implementation of the piecewise constant source term from Eq. (4) can be found in the `examples/3_poisson_source_term` folder of the data repository, where the name of the field computed by `ExternalField` is defined in the `source_model/settings.yaml` file

```
---   !ExternalFieldSettings
field_output_name: external_source_poisson
range: 1
```

to match the expected input name that we defined for the `PoissonModel` class.

With this setup, users can combine the two models in a model sequence. Model coupling is defined in a `coupling.yaml` configuration file as sketched in Fig. 5 (*left*). The coupling model sequence defines the order in which models are solved. Here, the first model in the sequence provides the `ExternalField` source term to the second `PoissonModel`. In the default setting of a *weak coupling*, the sequence (*second column of Fig.*5) runs once for each timestep of the simulation, exchanging data according to the connections defined using a coupling matrix.

In the coupling matrix, each row i represents the inputs to the i-th model in the coupling sequence, and an input is taken from the j-th model in column j if the entry $a_{ij} = \{i, j\}$ is one. The diagonal entries are set to one by convention. In our minimal example, the coupling matrix looks like this:

$$\begin{matrix} & \textit{from}\ \text{ExternalField} & \textit{from}\ \text{PoissonModel} \\ \begin{matrix}\textit{into}\\ \textit{into}\end{matrix} & \begin{matrix}\text{ExternalField}\\ \text{PoissonModel}\end{matrix} & \begin{matrix}\text{ExternalField}\\ \text{PoissonModel}\end{matrix} \end{matrix} = \begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix}. \tag{5}$$

According to the coupling sequence shown in Fig. 5 (*left*), the first row lists inputs into the `ExternalField` model while the second row lists inputs into the `PoissonModel`. Other than the diagonal entries, the entry at the second row and first column is one. With this, the bryne driver can connect the output of the `ExternalField` model, given by the source term $f(\mathbf{x})$, to the corresponding input of the `PoissonModel`. Ultimately, the `PoissonModel` can assign the input value to its own member variable `external_source` in the `link_coupling_input()` method:

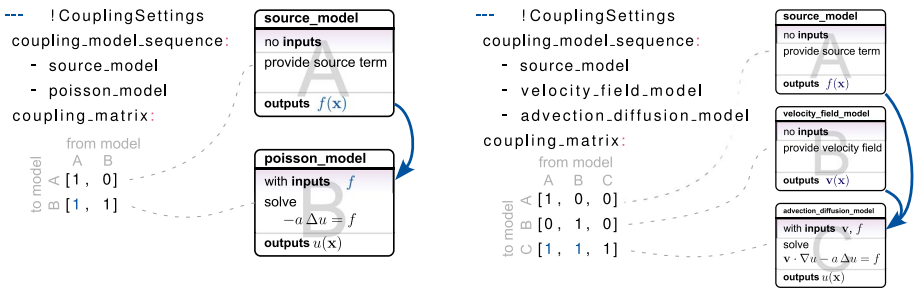

Fig. 5 Model coupling setup with bryne. In each timestep, models are solved in the order defined by the coupling sequence (*model boxes top to bottom*). If two models have a matching coupling interface, outputs of one model can be used as inputs for another model in the coupling sequence (*arrows*). These connections are set up via a coupling matrix, where each off-diagonal entry of one in row i and column j indicates that the i-th model in the sequence takes an input from the j-th model. In the left example, the Poisson model **B**, second in the coupling sequence, takes the output of the external field providing the source term **A**, which comes first in the sequence. Subsequently, the entry {2, 1} of the coupling matrix is one

```
def link_coupling_input(self):
    ... # check if coupling input exists, then
    self.external_source = self.coupling_input_data[
        "external_source_poisson"
    ].coupling_data
```

An example solution to the Poisson model for the first example setup with constant Dirichlet boundary conditions $g = 1$ and the source term from Eq. (4) is shown in Fig. 6 (*left*).

The coupling matrix input format is flexible in that it allows users to create arbitrary couplings. The data flow between models can be one-way or two-way, the latter being indicated by symmetric entries $a_{ij} = a_{ji}$. For one-way coupling, we have $a_{ij} = 1$ and $a_{ji} = 0$ or vice versa. Optional coupling inputs can be omitted by setting the corresponding entry to zero.

To summarize, the bryne coupling interface handles data exchange between classes derived from the bryne `Model` class. Since the term "coupling" is quite overloaded, it should be clearly stated that we do not refer to an exchange of data across interfaces or between different grids or even software packages here. Instead, we use "coupling" in an operator-split sense, referring to the passing of data between models defined on the same discrete grid at each timestep.

Weak and strong coupling

By default, bryne uses weak (sometimes also called "loose") coupling, meaning that the models in the coupling sequence are solved once for each simulation timestep. Users can omit the coupling setup entirely to run a collection of models without any data exchange. In that case, models will just be run in alphabetical order of appearance, as determined by the model input folder names.

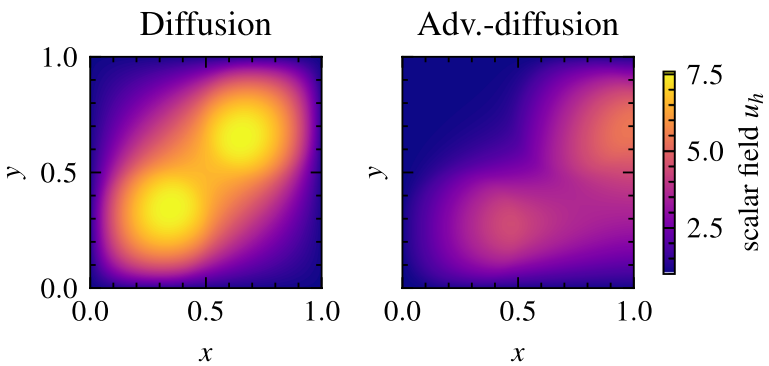

Fig. 6 Results for the first model building examples. The left figure shows the numerical steady state solution for the Poisson problem with constant Dirichlet condition $g = 1$and a spatially varying source term $f(\mathbf{x})$ given by two circles of radius $r = 0.25$ centered at $(x, y) = (0.25, 0.25)$ and $(0.75, 0.75)$ with $f_{\text{source}} = 100$ (*see Eqn.* (4)). On the right, a linear advection term with a constant velocity field $\mathbf{v} = (1, 0)$ is added, and a homogeneous Neumann condition is applied on the right boundary ($x = 1$). As a result, the solution is skewed in the direction of the external flow field

Bryne also supports strong coupling, where the coupling sequence is iterated. This requires each model to define a convergence criterion based on the change in solution fields between two consecutive coupling iterations.

2.5 Adding model-specific parameters

After we have combined our first two models, we show how to add parameters as user inputs through metadata files during simulation setup. After successfully adding a space-dependent source term as a generic coupling input, we would now like to add the diffusion constant a to Eq. (3a), such that our `PoissonModel` solves an approximation to

$$-a\,\Delta u = f(\mathbf{x}) \quad \text{in } \Omega\,. \tag{6}$$

In a first draft of a solver, users would likely just hard-code the diffusion coefficient a somewhere in the `PoissonModel` (*recall introductory figure* 1). The point is now to make the input parameter a traceable and modifiable without touching the model core implementation.

Adding model-specific parameters to be loaded via the bryne simulation driver takes only a few lines of code. The key to the parameter handling is a model-specific settings class that derives from the bryne `GenericSettingsClass`. Model parameters are then simply added as class attributes in the constructor. For example, to add a constant diffusion constant, we can define

```
from bryne. ... .generic_settings import GenericSettingsClass
class PoissonSettings(GenericSettingsClass):
    yaml_tag = "!PoissonSettings"

    def __init__(self, **kwargs):
        super().__init__(**kwargs)
        self.diffusion_coefficient = 1.0
```

Here, a default value of 1.0 can be specified. Any parameter docstring will be available in the API documentation. Then, in the `PoissonModel` class, we only add

```
from bryne.models.poisson_model.poisson_settings import (
    PoissonSettings
)
class PoissonModel(Model):
    def __init__(self, **kwargs):
        ...
        self.model_settings = PoissonSettings()
```

to make the diffusion coefficient available everywhere in the model code. To set the diffusion coefficient in our input simulation setup, we can add a `settings.yaml` file to the Poisson model setup folder, where users can set and check the value of a:

```
--- !PoissonSettings
diffusion_coefficient: 0.5
```

If the file is not provided, a will default to the value defined in the `PoissonSettings` class. At the start of each simulation, the `settings.yaml` file will be copied to the run output directory along with the rest of the simulation inputs, allowing users to trace the exact value of a used in each individual simulation run.

2.6 Adding model complexity

Full example code can be found in `advection_diffusion_example_model.py`, located in `bryne/models/advection_diffusion_model`.

With these basic building blocks in place, we can continue down the model cascade. We are reusing our existing Poisson model to build a solver for the steady linear advection–diffusion problem

$$\nabla \cdot (\mathbf{b}(\mathbf{x})\, u) - a\, \Delta u = f(\mathbf{x}) \quad \mathbf{x} \in \Omega\,, \tag{7a}$$

$$u = g \quad \mathbf{x} \in \Gamma_E\,, \tag{7b}$$

$$\nabla u \cdot \mathbf{n} = 0 \quad \mathbf{x} \in \Gamma_N\,, \tag{7c}$$

where the boundary $\partial\Omega = \Gamma_E \cup \Gamma_N$ consists of disjoint Dirichlet boundary Γ_E and natural Neumann boundary Γ_N parts. Since the Poisson model is fully contained in the advection–diffusion equation, we can build a new `SteadyLinearAdvectionDiffusionModel` by deriving from the `PoissonModel` (*recall the overview in Fig. 4*).

To use our model in combination with other models that provide a velocity field $\mathbf{b}$ as coupling output, we can add a vector of dimension d with key `transport_velocity` to the coupling interface in `define_coupling_input`. This way, we have again avoided defining the velocity field in the model code. Instead, we can now flexibly exchange the actual source of the velocity field. In a first trial, we might want to use a known velocity field and, similar to the previous example, we can configure an `ExternalField` to provide an analytical expression on the grid. The corresponding coupling setup is sketched on the right of Fig. 5, where the coupling sequence of the three models to provide the source term $f(\mathbf{x})$, the velocity field $\mathbf{b}(\mathbf{x})$ and the numerical solution of Eq. (7) is shown. An example where the previous Poisson setup with a spatially varying source term is complemented by a constant advective transport with velocity $\mathbf{b} = (1, 0)^T$ is shown in Fig. 6 (*right*). After testing the model with a known velocity field, we could then easily switch to a physics-based velocity field, as computed, for example, by an incompressible Navier–Stokes model.

To achieve this, the actual implementation of the model then only has to add the weak form advection term and the definition of a coupling input for the velocity field $\mathbf{b}$. By inheriting from an existing model, we don't have to rewrite any of the solver logic and model setup, reducing the entire implementation of the advection–diffusion model to only a few lines of effective code.

2.7 Transient models

Full example code can be found in `transient_advection_diffusion_model.py`, located in `bryne/models/advection_diffusion_model`.

With the steady advection–diffusion model in place, we can now add a time dependency to Eq. (7a) to obtain the transient advection–diffusion equation

$$\frac{\partial u}{\partial t} + \nabla \cdot (\mathbf{b}(\mathbf{x})\, u) - a\, \Delta u = f(\mathbf{x}) \quad \mathbf{x} \in \Omega\,. \tag{8}$$

The time discretization is a property of the model implementation and can be modified directly in `setup_weak_form()` by modifying the weak form according to a chosen time integration scheme. To compute the unknown numerical solution u_{n+1} at discrete time t_{n+1}, a simple implicit Euler time discretization,

$$\frac{\partial u}{\partial t} \approx \frac{u_{n+1} - u_n}{\Delta t} \quad \text{with } u_n = u(t_n) \text{ at time } t_n\,, \tag{9}$$

for a given solution u_n at the previous time step t_n can be implemented in UFL syntax

```
u = self.trial_function
v = self.test_function
time_discretization = dot((u-self.u_n)/ self.dt, v)*dx

self.weak_form = (
    time_discretization + a * dot(grad(u), grad(v)) * dx + ...
)
```

Parameters like the time discretization step size Δt can be controlled per model or globally through a `time.yaml` settings file in the simulation setup. Other time discretizations are possible by implementing the corresponding weak form terms. Modifying the stencil from Eq. (9), we could, for example, easily switch the Euler scheme for a third-order BDF2 method.

Again, the actual reordering of the weak form and the linear system assembly is handled by the DUNE-FEM backend and is not part of bryne. To this end, users are left with the task (and freedom) to implement the actual time update by overwriting the `solve_timestep()` method in the model class. In our example, this is a simple update along the lines of

```
def solve_timestep(self):
    self.time.value += self.dt.value
    self.u_n.assign(self.u_h)
    self.scheme.solve(target=self.u_h) # actual solve for u_{n+1}
    self.timestep += 1
```

but more complex update schemes can be implemented as required by the model. In particular, users can write their own nonlinear solvers or non-trivial predictor-corrector schemes. For interested readers, an example for this is implemented in the `ThermochemistryDGModel` class, which is described later and distributed as part of the supplementary code.

The modular design of our FEM solvers now again reduced the needed implementation effort to the actual differences between the steady and the transient versions of the advection–diffusion model. Since the coupling interface has been inherited, the transient solver can be directly combined with other models in the same way as its steady counterpart, for example, a coupling sequence similar to the one shown in Fig. 5 (*right*). With that, we can freely modify the source term and velocity field, both of which are part of the user simulation setup and can be arbitrary UFL functions of both space and time. Figure 7 shows a demo, where the source term is modified to be a spatio-temporally varying function,

$$f(\mathbf{x}) = \begin{cases} f_{\text{source}} & \text{if } \|\mathbf{x} - (0.25, 0.25)^T\|_2 < r\,, \\ 0 & \text{else}\,, \end{cases} \quad \text{for } t < 1.0 \quad \textit{and} \quad 0 \quad \text{for } t \geq 1.0\,, \tag{10}$$

built from the lower left circular source of the steady examples with $f_{\text{source}} = 10$ and $r = 0.25$, which is switched off for $t > 1.0$. The velocity field is $\mathbf{b} = (1, 1)^T$, $a = 0.01$, and all boundaries are homogeneous Neumann boundaries, leading to the comet-trail-like transport in the downwind direction of the circular source.

2.8 Boundary and initial conditions

In the previous example, we have sketched the architecture of model building in bryne. The question remains where and how to set the boundary conditions to close PDE problems. Because the assembly of the linear system is handled by DUNE-FEM, the enforcement of essential boundary conditions is closely linked to the backend library syntax.

For maximum flexibility, boundary conditions can be defined in a `boundary.py` file, which is part of the simulation inputs for each FEM model. In this file, users are provided with a template function `setup_boundary()` that can be modified to set up the boundary conditions for the model. For example, a basic Dirichlet boundary condition for our transported scalar u with a value of 1 on the left boundary of a 2D rectangular grid could be set like this:

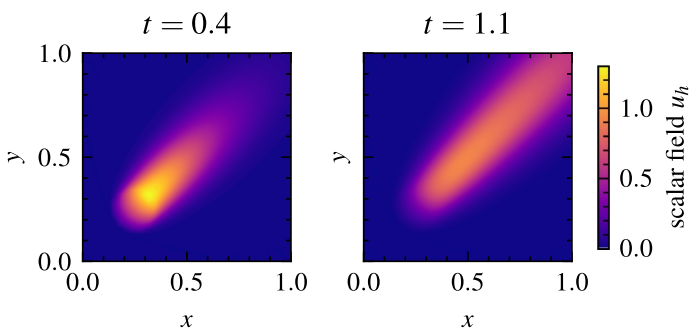

Fig. 7 The second basic example adds a time discretization to the steady advection–diffusion model and solves the model for $a = 0.01$ with a spatio-temporally varying source term $f(\mathbf{x}, t)$. The source consists of the bottom-left circular source term of the steady example problem with $f_{\text{source}} = 10$, which is switched off for $t > 1.0$

```python
from ufl import eq
from dune.ufl import DirichletBC, BoundaryId

def setup_boundary(model: GeneralFEMModel,
                   boundary: ModelBoundary):
    ...
    space = model.discrete_space
    is_left = eq(BoundaryId(space), 1)
    boundary.dirichlet_boundary_eqns = \
        [DirichletBC(space, 1.0, is_left)]
    ...
```

At runtime, the bryne driver will call this function, passing in each FEM model instance and its respective boundary attribute. Generally, boundary conditions can be passed in directly as DUNE-FEM `DirichletBC` objects or by directly passing UFL values to the degrees of freedom in the solution space of the model. The actual processing of the boundary values, such as interpolation of the prescribed initial condition, happens in a method `set_boundaries()` that all `GeneralFEMModel`s have to implement. Different types of boundary conditions are managed through the `ModelBoundary` class. For interested readers, we refer to the class API documentation for a more in-depth discussion. A variety of `boundary.py` examples are distributed with the attached repo, both in the `examples` directory and `tests` subfolders.

Note that for some boundary conditions, model implementations need to handle their own processing of the information stored in the `ModelBoundary` class. This includes non-homogeneous flux boundary conditions or weakly enforced Dirichlet values, the latter being common in discontinuous Galerkin (DG) methods. We forego a discussion here to keep this description concise, but examples can be found in the `StokesBrinkmanDGModel` and `ThermochemistryDGModel` class implementations.

2.9 Extending the examples

The previous examples, summarized in Fig. 4, have shown the essential ingredients of bryne FEM models. The example models are kept minimal and therefore lack any stabilization terms. The toy examples nevertheless open up a range of extension possibilities, and we invite readers to try the supplementary Docker container and play with the examples. Some ideas that could be explored:

1. build your own space and time-dependent source term,
2. make the velocity field space and time dependent (*an example with a vortex-like flow field can be found in the* `examples` *directory*),
3. extend the transient advection–diffusion model, for example, by adding a reactive term
4. play with the time discretization to try out different time integration schemes, or
5. modify the weak form to apply stabilization or a DG scheme to stabilize advection (*a mixed DG solver for incompressible flows is implemented in the* `StokesBrinkmanDGModel` *class*).

Below, we summarize some benefits of writing a model in bryne over singular notebooks or scripts:

- There is no need to rewrite or copy-paste any parts of the solver logic other than the call to the DUNE-FEM (non-)linear solver. The bryne driver provides the outer time loop, and just by adding your model to the simulation setup, it will be solved at each timestep.
- Any working base version of a model can be extended by inheritance.
- For coupled problems that can be solved in an operator-split fashion, models can be built and tested individually and then combined.
- By using the bryne `ExternalField` model to provide known fields like source terms, experimenting becomes easier since the exact definition of the field will be stored with each simulation run.
- Similarly, model parameters can be exposed as user inputs to make setups easily modifiable and reproducible.

2.10 Simulation setup structure

As shown in Fig. 8, bryne simulation setups are folders with a tree structure that mirrors the building blocks of a simulation. At the core of a bryne simulation lies an `inputs` directory, which contains the input parameters grouped by simulation components.

Basic inputs are provided in two ways. Static inputs such as settings or parameters with simple numerical or string values can be provided in human-readable YAML files:

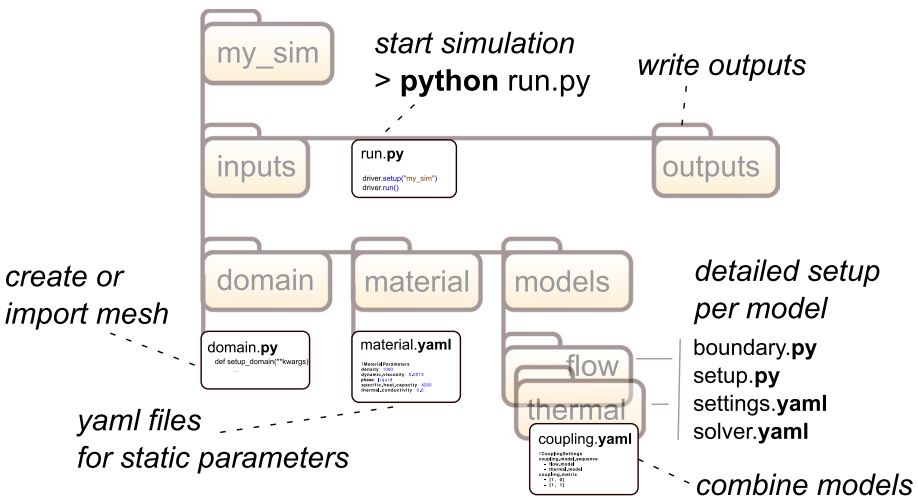

Fig. 8 Structure of bryne simulation inputs. A minimal simulation folder consists of a directory `inputs` providing the simulation setup and a Python script `run.py` that calls the bryne simulation driver. Simulation setups are structured by simulation components, the actual input definition is realized through a combination of editable YAML files and Python scripts imported at simulation runtime

```
--- !MaterialParameters
phase: liquid
density: 1000
dynamic_viscosity: 0.0013
specific_heat_capacity: 4200
thermal_conductivity: 0.6
```

Dataclasses, such as `MaterialParameters`, derive from `YAMLObject` and therefore can be instantiated by loading a YAML file with the corresponding class tag, in this case `!MaterialParameters`. Documentation and default values for parameters can be provided in the class definition. For example, we can check the `MaterialParameters` class to find

```
class MaterialParameters(GenericSettingsClass):
    yaml_tag = "!MaterialParameters"

    def __init__(self, **kwargs):
        super().__init__(**kwargs)
        self.thermal_conductivity = 0.6
        """Thermal conductivity, default is [W / (m*K)]"""
        self.latent_heat = 333700
        """Latent heat (of solidification), default is [J/kg]"""
```

The setup is robust in the sense that default values will be available at runtime even if the user does not provide a value in the YAML file. An advantage is that the parameter explanation in docstrings is available to the user as part of the automated API documentation. Examples of bryne pre-implemented parameter classes can be found in `bryne.io_manager.settings`. In a later example, we will show that it is easy to add new custom parameter classes when developing new models.

Some parts of the simulation setup can be more complex or require Python code to be executed at runtime. One example is the setup of the computational domain, where a YAML-based approach quickly becomes limiting as it is not flexible enough to cover options for all mesh types or external mesh file imports. For such cases, inputs are provided as small pre-defined Python functions that are imported at runtime. For example, the `domain` folder contains a file `domain.py` with a user-modifiable function code `setup_domain()` to set up the grid:

```python
from dune.grid import cartesianDomain
from dune.alugrid import aluCubeGrid as leafGridView
from bryne.config.config import simulation_config as config

def setup_domain(**kwargs):
    # define x_min, x_max, y_min, y_max, n_x, n_y.
    # ... then
    domain = cartesianDomain([x_min, y_min],
                             [x_max, y_max],
                             [n_x, n_y])
    config.gridView = adaptiveGridView(leafGridView(domain))
```

We decided on this approach as it allows for maximum flexibility. For example, due to the modular design of the DUNE-FEM backend [6], changing the grid type from a quad to a triangular grid is as simple as exchanging `aluCubeGrid` with `aluConformGrid` in the import statement. We provide several setup examples that demonstrate usage without requiring a deep understanding of the DUNE-FEM Python API.

DUNE-FEM loads structured and unstructured meshes from the dune-native dune grid format, `.dgf`. Alternatively, the domain can be constructed from a Python dictionary of vertices and cells. This allows users to parse geometry information from other mesh formats such as Gmsh [23] `.msh` files, which can be loaded through third-party libraries such as meshio [24].

Even a minimial FEM simulation setup will always require a discrete domain and at least one model to solve. Other than the `domain`, the second essential simulation building block is the `models` directory. It contains a subfolder with the detailed parameter setup for each model to be solved in the simulation. The execution order and data exchange between models is configured in a `coupling.yaml` file, discussed later.

The bryne simulation driver is created and executed in a Python script `run.py` located in the simulation root directory,

```python
from bryne.driver.driver import SimulationDriver

driver = SimulationDriver()
driver.setup()
driver.run()
```

and the simulation can then be started by executing `python run.py`. The DUNE-FEM backend also supports MPI [25] distributed memory parallelism, which can be invoked via `mpirun -np N python run.py`, where N is the number of processes. For this to work, the particular simulation needs to consist of an MPI-compatible combination of linear solver, preconditioner and numerical grid, see supplementary Appendix A.

For every simulation run, bryne will store a copy of the simulation `inputs` directory in the output directory of the simulation run. That way, structured metadata on the exact simulation setup is always kept with the simulation results. Hard-to-document simulation details such as the exact polynomial type, mesh, convergence tolerances, relaxation

factors, and material parameters are hence always sustainably archived. Because the `inputs` folder archived for a given simulation contains the full simulation setup, it can be rerun as a bryne simulation. This constitutes a strength of our approach, as it allows for easy reproduction given a fixed bryne and DUNE-FEM version. To pin the exact model version used to achieve a result, git metadata such as branch names and commit hashes are stored with the outputs.

In conjunction, our approach results in a collection of re-executable metadata for simulation setups.

2.11 Adaptive mesh refinement

We conclude the chapter on bryne features with a brief discussion of mesh adaptivity. Figure 9 shows a sketch of a bryne timestep solve for a weakly coupled model sequence. Some DUNE-FEM grids, such as `ALUGrid` [26], support adaptive mesh refinement (AMR). Bryne builds on this to allow for h—refinement and coarsening. Currently, two types of refinement are supported:

- **Initial refinement** criteria are evaluated once at the start of the simulation. This can be used to refine around regions of interest, based on the initial condition or geometric features.
- **Timestep refinement** criteria are evaluated at the end of each timestep. As shown in Fig. 9, users can provide custom indicator functions of the current solution fields to determine where the mesh should be refined or coarsened.

In the current implementation, all refinement criteria require a scalar indicator function $I(\mathbf{x})$ on the grid, which is evaluated at the corresponding adaptivity call. Bryne then passes this function and user-defined coarsening and refinement thresholds to the DUNE-FEM backend, which will then adapt the mesh according to

$$\text{do} \quad \begin{cases} \text{refine} & \text{if } I(\mathbf{x}) > \text{refinement threshold}\,, \\ \text{coarsen} & \text{if } I(\mathbf{x}) < \text{coarsening threshold}\,, \\ \text{nothing} & \text{otherwise}\,. \end{cases} \tag{11}$$

The function $I(\mathbf{x})$ can be any space- and time-dependent UFL expression. In particular, it can depend on any of the current solution fields computed in the model sequence. The indicator function as well as coarsening and refinement thresholds, number of

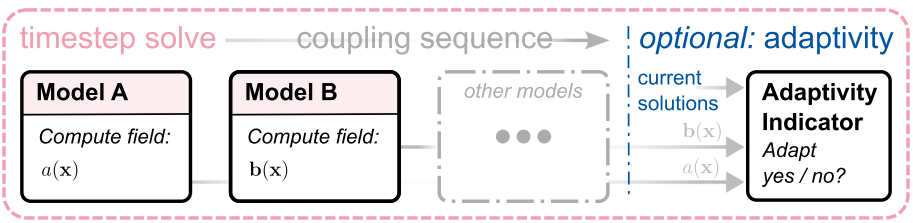

Fig. 9 Sketch of a timestep solve in bryne. At each timestep, a sequence of models is solved (*left to right*). Each model computes its solution field(s), optionally passing data to other models down the coupling sequence. Optionally, bryne supports h—refinement for meshes that support adaptivity, such as grids based on `ALUGrid` [26]. Users can provide custom refinement criteria, based on an indicator function that can use any current solution fields

refinement calls (*how often to refine per adaptivity call*), and maximum refinement levels can optionally be set in a subfolder `adaptivity` of the simulation input directory. In the following discussion of the application examples, we show an illustrative example of a mesh refinement setup for the 1D Stefan problem.

Note that the DUNE-FEM backend supports p—refinement of the finite element degree. We have not yet experimented with this feature in bryne and forego a discussion here.

3 Application examples

Starting from a toy PDE example, we have covered basic ingredients of the bryne workflow. In the following, we show some more complex examples to demonstrate how we use bryne in our research. We start with a brief overview of the pre-implemented models that ship with the open-source release of bryne and then discuss how they can be combined to build complex simulations incrementally.

3.1 Pre-implemented models

Python source code for the pre-implemented models can be found in `bryne/models/specific_model_name` *directory of the bryne repository.*

The initial open-source release of bryne includes a set of pre-implemented solvers that demonstrate the effectiveness and value-add of the bryne framework. Table 1 shows a short overview of models, including the example model hierarchy. Other than that, the first release comes with two solvers:

- The `StokesBrinkmanDGModel` class implements a mixed discontinuous Galerkin FEM solver for the Darcy-Brinkman-Stokes (DBS) model described in detail in our previous work [17]. The DBS model describes incompressible flow on heterogeneous domains, where the porosity or liquid volume fraction Φ is allowed to vary between 0 (*no flow*) over $\Phi \ll 1$ (*Darcy flow*) to 1 (*Navier–Stokes flow*). The `StokesBrinkmanDGModel` solver seeks an approximation to the velocity field $\mathbf{u}$ and pressure p described by the equation [27]:

Table 1 Overview of FEM models that are part of the bryne release described in this article

	Name	Equation	Interface		Ref.s
			In	*Out*	
Demo	Laplace	(1a)		u	
	Poisson	(3a)	f	u	
	AdvectionDiffusionExample	(7)	$f, \mathbf{b}$	u	
	TransientAdvectionDiffusion	(8)	$f, \mathbf{b}$	$u(t)$	
Research	ExternalField	custom algebraic equations	user def	user def	
	Permeability	porosity-permeability relation	Φ	$\mathbf{K}(\Phi)$	[27]
	StokesBrinkmanDG	Darcy-Brinkman-Stokes model	$\underline{\Phi}$, $\underline{\mathbf{K}}$, T	$\mathbf{u}$, p	[17]
	ThermochemistryDG	solid-liquid energy conservation	$\mathbf{u}$	Φ, T	[21]

Models from the example section are shown first, the last four models are part of our application examples. The two *interface* columns indicate coupling inputs and outputs of the model, as defined in the `define_coupling_input()` method of the model class. Underlined variables are mandatory inputs needed to run the models. Other inputs are optional and the respective model will provide sensible defaults (usually assuming that the corresponding contribution is zero)

$$\rho_l \left(\frac{\partial \mathbf{u}}{\partial t} + \mathbf{u} \cdot \nabla \left(\frac{\mathbf{u}}{\Phi} \right) \right) - \eta_l \nabla^2 \mathbf{u} + \nabla p + \Phi \, \eta_l \, \mathbf{K}(\Phi)^{-1} \, \mathbf{u} = \mathbf{f} \text{ on } \Omega^{\text{DBS}} \times (0, T] \,, \tag{12a}$$

$$\nabla \cdot \mathbf{u} = g \text{ on } \Omega^{\text{DBS}} \times (0, T] \,, \tag{12b}$$

$$\mathbf{u} = \mathbf{u}_0 \,, \quad p = p_0 \text{ at } t = 0 \,. \tag{12c}$$

Here, Φ is the porosity or liquid volume fraction field, $\mathbf{K}(\Phi)$ is the permeability tensor, and ρ_l, η_l are the density and dynamic viscosity of the fluid, respectively. A core property of this model is that the Navier-Stokes equations and Darcy's law are recovered from Eqns. (12) in the limit $\Phi \to 1, \left\| \mathbf{K}^{-1} \right\| \to 0$ and $\Phi \to 0, \left\| \mathbf{K}^{-1} \right\| \to \infty$, respectively. In bryne, the body force $\mathbf{f}$ term can be used to model thermal or chemical buoyancy through a Boussinesq approximation. The respective temperature field T is declared a coupling input in the `StokesBrinkmanDGModel` class, allowing the model to take the temperature field as input from arbitrary other models. We will later show an example of how this can be used to couple the DBS model to a thermal energy conservation model in a phase change simulation. For a detailed discussion, description of the discretization method, and a series of validation examples, we refer to [17].

Note that the code itself was not public before, but is now available as part of the bryne release [22].

- The `ThermochemistryDGModel` class implements a DG-FEM solver for the two-phase energy conservation equation:

$$\frac{\partial H}{\partial t} + \nabla \cdot (\mathbf{u} \, H_l - k(\Phi) \, \nabla T) = 0 \quad \text{in } \Omega \times (0, T] \,, \tag{13a}$$

where $\mathbf{u}$ is the liquid velocity, k is the thermal conductivity. The bulk enthalpy H of the liquid-solid mixture is composed of solid and liquid phase enthalpies H_l and H_s through

$$H = \Phi \, H_l + (1 - \Phi) \, H_s \,, \quad k(\Phi) = \Phi \, k_l + (1 - \Phi) \, k_s \,, \tag{13b}$$

$$H_l = \rho_l \, c_{p,l} \, T + \rho_l \, L \,, \quad H_s = \rho_s \, c_{p,s} \, T \,, \tag{13c}$$

and $0 \leq \Phi \leq 1$ indicates the liquid volume fraction. The latent heat contribution L enters implicitly as part of the liquid enthalpy definition, which, together with the nonlinearity of the diffusive term, makes Eq. (13) challenging to solve by standard means. The specific discretization method and iterative solution scheme to find bulk enthalpy H (or temperature $T = T(H)$) and Φ to satisfy Eq. (13) goes back to observations from [28] and is described in detail in [21]. This particular update scheme, implemented in the `ThermochemistryDGModel`, has been shown to be superior to other approaches in terms of stability and required nonlinear iterations [29]. In the context of classical finite elements, similar predictor-corrector schemes have been discussed in [30, 31]. To our knowledge, this is the first implementation of such a DG-based enthalpy method to become open-source.

In the following, we focus on the usage of bryne as a simulation builder. We thus emphasize the reusability of tested models and put less stress on numerical details. In-depth numerical discussion is provided in the respective references and the full source code is available as part of the supplementary code repository. It also contains the bryne test suite with several regression tests that check the correctness of various subcomponents of the presented models.

3.2 Simulation building: convection-coupled phase change

Simulation metadata, results and raw figures are available as part of the Zenodo data supplement [32].

The final example combines the `StokesBrinkmanDGModel` and the `ThermochemistryDGModel` to build a convection-coupled phase change simulation. The resulting simulation replicates an experimental benchmark for melting Octadecane with data from [29]. With this, we illustrate how bryne benefits a typical numerical modeling project, where several simulation subcomponents are combined:

- the `StokesBrinkmanDGModel` solves the incompressible flow problem for liquid velocity $\mathbf{u}$ and pressure p.
- a `Permeability` model is used to model a spatio-temporally varying permeability field $\mathbf{K}(\Phi)$ to effectively force zero flow in the solid phase ($\Phi = 0$) and Stokes flow in the liquid phase ($\Phi = 1$).
- the `ThermochemistryDGModel` self-consistently solves the energy conservation equation for the bulk enthalpy H, temperature T and liquid volume fraction Φ.

The two DG-based solvers are examples of complex solvers for PDE models that include several subproblems, all of which must be tested individually. Development will therefore be done in several steps, with test complexity gradually increasing. Figure 10 shows the sequence of models as they appear in the coupling sequence of the following examples. With bryne, we can first develop and test the

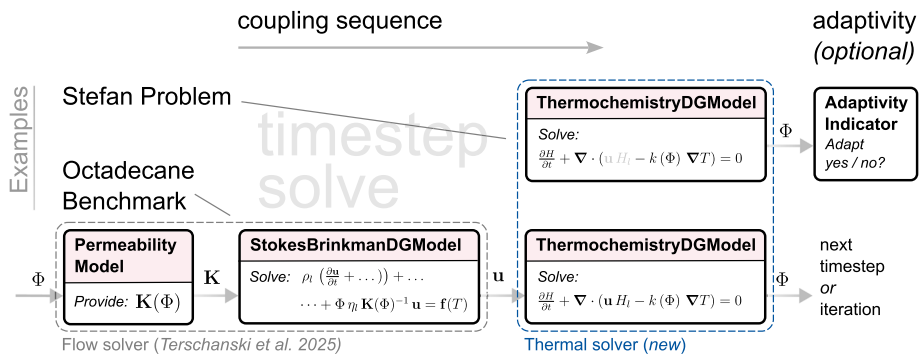

Fig. 10 Overview of bryne application examples presented in Sect. 3.2. Examples are sorted by appearance from top to bottom. The coupling sequence (left to right) shows the active models and the flow of information between them. In a weakly coupled simulation, the sequence is executed once per timestep solve; in an optional strongly coupled simulation, it can be iterated. Fields can be passed to an adaptivity indicator function for adaptive mesh refinement between timesteps

`ThermochemistryDGModel` without advection (*Stefan problem, first row*). In a second step (*Octadecane benchmark, second row*), we can combine the model with a flow solver, validated separately in [17].

We have already discussed how Python-based discretization backends such as DUNE-FEM offer a convenient way of adding model complexity. Bryne adds to that a structured way of incremental model development:

1. In this concrete example, we started with the flow solver implemented in the `StokesBrinkmanDGModel`. We could build trust in our initial prototype method after extended testing on various 2D benchmarks [17]. With the initial incompressible solver in place, it is easy to add a linear Boussinesq buoyancy term,

$$\mathbf{f} = \rho_l \, \Phi \, \beta \, (T - T_0) \, \mathbf{g} \,, \tag{14}$$

   as a body force term in Eq. (12a), where β is the liquid thermal expansion coefficient, T_0 is a reference temperature, and $\mathbf{g}$ is the gravitational acceleration vector. The temperature field T can then be declared as a bryne coupling input.
2. In our case, the temperature field is to be computed from a two-phase energy conservation solver, which can be developed and tested independently. Following [21], the 1D Stefan problem serves as validation of the iterative solver for Eq. (13) in the absence of advective transport ($\mathbf{u} = 0$). Since this benchmark does not require a flow field, we can first write and test our prototype `ThermochemistryDGModel`, independently of a flow solver.

Fig. 12 shows the results from a validation study computed with bryne, where the numerical solution is compared to the analytical solution of the 1D Stefan problem. In this setup, sketched on the left of Fig. 11, a 1D liquid rod of length $L = 0.05\,m$ at initial temperature $T_0 = T(x,\, t = 0) = 278\,K$ is suddenly cooled from the left boundary at $x = 0$ with a constant temperature $T_c = 268\,K$ below the melting temperature $T_m = 273\,K$. Benchmark details and reference solutions taken from [21] are repeated in the supplementary Appendix B. On the left of Fig. 12, we compare the normalized numerical temperature solution $(T_{\text{num}} - T_c)/(T_0 - T_c)$ (*dashed blue line*) with

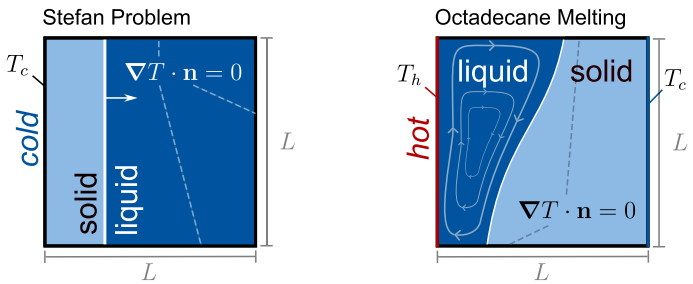

Fig. 11 Setup sketches for the application examples. In the 1D Stefan problem (*left*), a liquid, initially at constant temperature, is suddenly cooled below the freezing temperature. The scenario serves as a benchmark for solidification with purely diffusive heat transport. We build a solver for convection-coupled phase change, tested on the Octadecane benchmark (*right*), by combining the thermal solver with a flow solver. In this setup, a solid is heated above its melting temperature at the left boundary, and natural convection causes a rotational flow in the melt

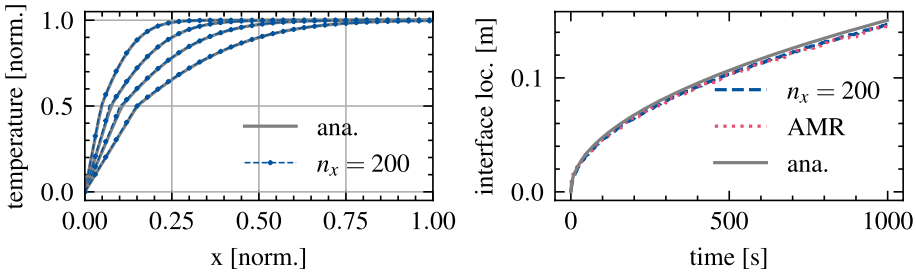

Fig. 12 Verification of the bryne `ThermochemistryDGModel` on the 1D Stefan problem. The figure on the left compares the numerical temperature profile (*blue dashed line*) with the analytical solution (*solid gray line*) at consecutive times $t = \{50, 100, 250, 500\}\,s$. Kinks in the temperature profile indicate the phase change interface (PCI) location, which on the right is shown together with the analytical PCI as a function of time. One numerical solution (*dashed blue line*) is computed on a quadrilateral grid with $n_x \times n_y = 200 \times 1$ first-order Lagrangian finite elements and a time step size of $\Delta t = 0.5\,s$. On the right, we also compare with a solution computed with adaptive mesh refinement (*AMR, dotted pink line*) on a coarse base mesh of $n_x \times n_y = 25 \times 1$quadrilateral elements, locally refined around the PCI in two bisection refinement steps. *Simulation data is available in the* `full_stefan_1d` *and* `full_stefan_1d_with_amr` *folders in the Zenodo data supplement* [32]

the corresponding analytical solution (*solid gray line*). On the right, we show the location of the phase change interface (PCI) as a function of time, where the numerical PCI (*dashed blue line*) is approximated from the $\Phi = 0.5$ contour. At final time ($t = 1000\,s$), the deviation of the numerical PCI relative to the analytical interface location is less than 1 %. An almost identical solution can be obtained with adaptive mesh refinement (AMR) (*dotted pink line*), as indicated by the dotted pink line on the right of Fig. 12 (*for the temperature plot lines are indistinguishable visually*). The AMR solution is computed from a coarse base mesh with $n_x \times n_y = 25 \times 1$ quadrilateral elements, implemented using dune ALUGrid [26]. The grid is locally refined and coarsened around the PCI at each timestep in two bisection steps. Our example refinement indicator

$$I(\mathbf{x}) = \|\nabla\Phi(\mathbf{x})\|_2 \tag{15}$$

is based on the gradient of the liquid volume fraction Φ, which is nonzero only close to the interface between the solid and liquid phases. The resulting AMR solution, shown on the right of Fig. 12, achieves a similar accuracy to the finely resolved mesh while using significantly fewer elements. We forego a detailed discussion of AMR efficiency here but include the example and simulation data to show how bryne integrates mesh adaptivity into the simulation workflow.

3. To add advective transport to the energy conservation model, we can now add corresponding terms to the weak form definition in the `ThermochemistryDGModel` class. Again, the advection velocity is declared as an optional coupling input.
4. We can now test our DG discretization for the advective term by coupling our draft `ThermochemistryDGModel` to a known velocity field, similar to the introductory Example (see Fig. 4). At this point, the modular structure of bryne allows us to test subcomponents of our more complex discretization without having to hard-code any velocity field in the model implementation.

With the flow and energy conservation models tested individually, we can now reuse our models to combine them into a single simulation setup:

5. To model flow on a domain with an implicit boundary between solid and liquid phase as indicated by the liquid volume fraction Φ, we can use a permeability model with infinite permeability in the liquid phase ($\Phi = 1$) and near-zero permeability in the solid phase ($\Phi = 0$).
6. Ultimately, all models can be combined in a single coupling sequence, where the `StokesBrinkmanDGModel` provides the velocity field $\mathbf{u}$ to the `ThermochemistryDGModel` which in turn computes the temperature field T entering the buoyancy term (*Eq.* (14)) and liquid volume fraction Φ to update the permeability field $\mathbf{K}(\Phi) = k(\Phi)\mathbf{I}$ for the flow solver.

The resulting coupling, sketched in the bottom row of Fig. 10, corresponds to a classical operator-split, where the hydromechanical and thermal equations are solved sequentially. Optionally, the sequence can be iterated in a strong coupling iteration until the coupling update error $\epsilon_{cp}^{\Psi} = \|\mathbf{\Psi}_{m+1} - \mathbf{\Psi}_m\|_2$ of all numerical fields $\mathbf{\Psi}$ between the last two coupling iterations $m+1$ and m is below a convergence threshold. By verifying that solution fields do not change significantly after the first coupling iteration, we have confirmed that the assumption of weak coupling is a reasonable approximation for the flow regime of the following benchmark setup.

Figure 11 (*right*) shows the numerical benchmark configuration corresponding to the experimental setup from [29]. A $[-0.02, 0.02]^2\,m$ square domain with adiabatic top and bottom walls is filled with Octadecane, initially solid at constant temperature $T_0 = 298.15\,K$ (*top row of Fig.*13). While the right boundary ($x = 0.02\,m$) is kept at a constant temperature $T_c = 298.15\,K$, the left boundary ($x = -0.02\,m$) is suddenly heated to a constant temperature $T_h = 308.15\,K$ above the melting point $T_m = 301.15\,K$. This causes the solid to melt, and the subsequent heating of the liquid phase forces natural convection. The resulting clockwise flow field accelerates melting at the top of the cavity, and a non-trivial liquid–solid phase change interface (PCI) develops.

Figure 13 shows the temperature and velocity contours at different times t, obtained on a 100×100 quadrilateral grid with discontinuous Lagrangian elements of order $\{2, 1, 1\}$ for velocity, pressure, and temperature, respectively. Material properties are summarized in Table 2. The timestep size is chosen as $\Delta t = 0.01\,t_v$, where $t_v = \left(\rho_l L^2\right)/\mu_l \approx 330.92\,s$

Table 2 Material properties for the Octadecane benchmark

Parameter			Solid	Liquid
Density	ρ	$\mathrm{kg\,m^{-3}}$	867	775.6
Specific heat capacity	c_p	$\mathrm{J\,kg^{-1}}$	1900	2240
Thermal conductivity	k	$\mathrm{W\,m^{-1}\,K^{-1}}$	0.32	0.15
Thermal expansion coefficient	β	$\mathrm{K^{-1}}$		8.36×10^{-4}
Dynamic viscosity	μ	Pa s		3.75×10^{-3}
Latent heat	L	$\mathrm{J\,kg^{-1}}$	243680	
Melting temperature	T_m	K	301.15	

The same properties are used in [21, 29], original data is from [33, 34]

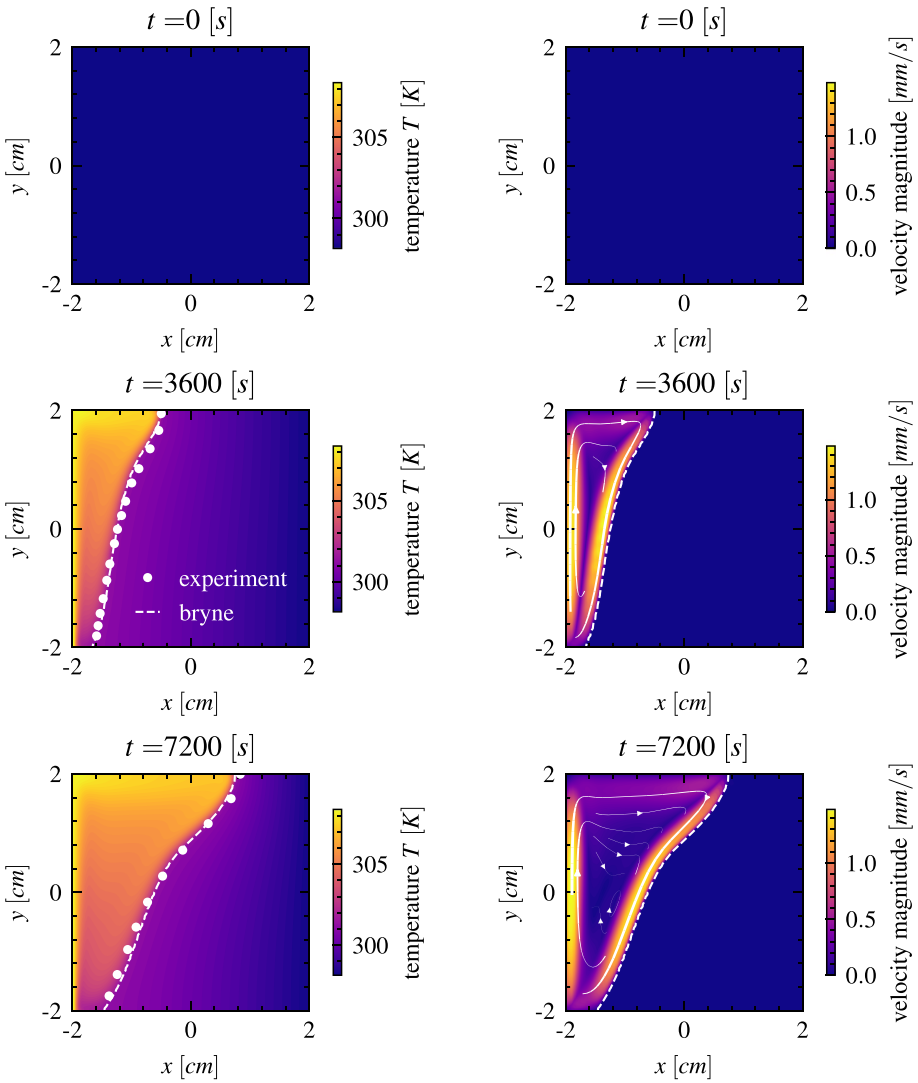

Fig. 13 Octadecane benchmark for a convection-coupled melting simulation. The left and right columns show the temperature and velocity contours at different times t. Results are obtained on a 100×100 quadrilateral grid with a nondimensional time step size corresponding to a physical time step of $\Delta t = 3.31\,s$. The numerical liquid–solid phase change interface is approximated by the $\Phi = 0.5$ contour, indicated by the dashed white line. White dots correspond to the experimentally measured interface location published in [29]. *Simulation and reference data is available in the* `octadecane` *folder of the Zenodo data supplement* [32]

is the viscous timescale for the cavity with side length $L = 0.04\,m$. The interface between solid and liquid phases is approximated by the contour of $\Phi = 0.5$, which is indicated by the dashed white line. White dots show the interface location as measured in [29]. Temperature and flow field as well as the interface location agree well with the experimental data and numerical campaigns from [21, 29]. For the latter, a small mesh convergence study is shown in Fig. 14, where the numerical PCI location obtained from a sequence of refined quadrilateral meshes is compared to the experimental data from [29] at two times. In agreement with previous observations, the enthalpy-based nonlinear update scheme converges reliably in $\mathcal{O}(10)$ nonlinear iterations per timestep. However, the rigorous weak enforcement of boundary values for the DG scheme needed to obtain

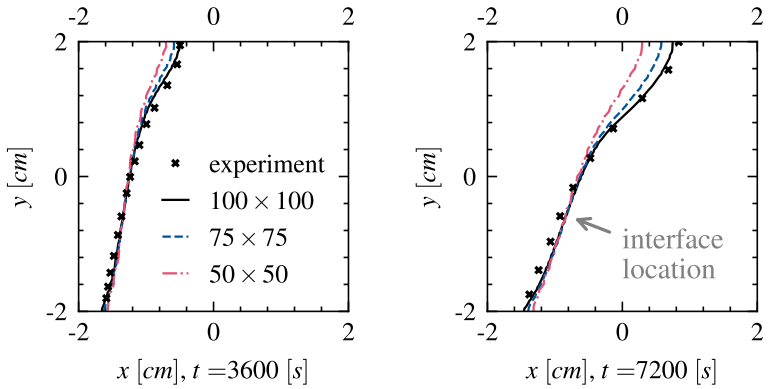

Fig. 14 Location of the liquid–solid phase change interface at two representative time snapshots. The approximate numerical interface location computed with bryne at different mesh resolutions is recovered from the $\Phi = 0.5$ contour. The reference data comes from experimental measurements published in [29]

the results from Figs. 13 and 14 adversely affects the conditioning of the linear system, resulting in costly linear solves in each nonlinear iteration. In this study, we have focused on an initial physical correctness check. Improving the linear solver efficiency and scalability will be part of a future work.

4 Conclusion

In this article, we presented the bryne Python software framework for reusable FEM models implemented in DUNE-FEM. With this, we address a gap between the rapid-prototyping capabilities of open-source FEM backends and the need for a reproducibility-enabled infrastructure to tackle complex simulation setups. Bryne enhances the reusability of DUNE-FEM model-based workflows by adding three dimensions of model reusability:

1. wrapping core components of FEM simulations in an object-oriented architecture allows writing FEM solvers that can be reused, either in conjunction with other models or by extending the model through inheritance.
2. Our first implementation of a model coupling interface allows building operator-split multiphysics simulations from models defined on the same grid.
3. Simulation inputs can be provided by combining YAML files for static parameters and user-modifiable Python scripts. This allows for sustainable archiving of simulations and setups that are easier to read, modify, and reproduce.

Starting from a minimal example, we have shown what comprises a bryne FEM model and how the object-oriented approach allows for building upon existing models incrementally. This approach has facilitated our research by providing efficient infrastructure for reproducible workflows. The software framework enables complex multiphysics models built from individually developed and tested components. This was demonstrated with the help of a convection-coupled phase change simulation that combines a flow and a thermal solver. For both models, now made open-source for the first time, bryne handles the complex simulation setup and solver logic. This allowed us as model developers to focus on the model implementation and testing. Using bryne, the release

of these models along with the entire setup of verification examples comes at no additional development cost.

In bryne's first release, we focused on finite element models built from the DUNE-FEM Python API. Many of the concepts presented here translate to other Python FEM frameworks such as FEniCS or Firedrake. Extending the bryne model interface to allow for FEniCS or Firedrake FEM models would be an interesting future step. While we have only considered data exchange across models defined on the same mesh, more complex couplings could be realized through an external coupling library such as preCICE [35].

Supplementary Information

The online version contains supplementary material available at https://doi.org/10.1007/s44207-026-00015-w.

Supplementary file 1

Acknowledgements
We thank Ingo Steldermann for his helpful comments on the manuscript draft.

Author contributions
BT: Writing—original draft, Visualization, Validation, Software, Methodology, Investigation, Formal analysis, Data curation, Conceptualization. RK: Writing—review and editing, Software. AD: Writing—review and editing, Software. JK: Writing—review and editing, Supervision, Resources, Funding acquisition, Conceptualization.

Funding
Open Access funding enabled and organized by Projekt DEAL. This work was funded by the German Research Foundation (DFG), Germany - grant number 460819306. Robert Klöfkorn was supported by the Swedish Research Council through grant AI-Twin (2024-04904).

Data availability
Data and code have not been described elsewhere before. Simulation data, including inputs and paraview files, along with the raw result figures are available on Zenodo: https://doi.org/10.5281/zenodo.15850111 Bryne is open-source software under the GNU General Public License v3.0 or later. The active source repository can be found at www.bryne-code.com This article refers to release version 1.1.0, which is pinned at the following Zenodo repository: https://doi.org/10.5281/zenodo.19603432 We provide a Bryne Docker container including a fixed compatible dune-fem version to facilitate reproduction. The container can host a JupyterLab for users who just want to quickly review the examples or the source code. Instructions on how to use the container are included in the documentation distributed on Zenodo.

Code availability
Bryne is open-source software under the GNU General Public License v3.0 or later. The active source repository can be found at www.bryne-code.com This article refers to release version `1.1.0`, which is pinned at the following Zenodo repository [22]: https://doi.org/10.5281/zenodo.19603432 We provide a bryne Docker container including a fixed compatible DUNE-FEM version to facilitate reproduction. The container can host a JupyterLab for users who just want to quickly review the examples or the source code. Instructions on how to use the container are included in the documentation distributed on Zenodo.

Declarations

Competing interests
The authors declare no competing interests.

Received: 15 July 2025 Revised: 6 July 2026 Accepted: 10 July 2026
Published online: 21 September 2026

References

1. Arndt D, Bangerth W, Davydov D, Heister T, Heltai L, Kronbichler M, et al. The deal.II finite element library: design, features, and insights. Comput Math Appl. 2021;81:407–22. https://doi.org/10.1016/j.camwa.2020.02.022.
2. Kondov I, Sutmann G. Centre Européen de Calcul Atomique et Moléculaire (eds.): Multiscale Modelling Methods for Applications in Materials Science: CECAM Tutorial, 16 - 20 September 2013, Forschungszentrum Jülich; Lecture Notes. Schriften Des Forschungszentrums Jülich: [...], IAS Series, 2013;19. Forschungszentrum, Zentralbibliothek, Jülich.
3. Baratta IA, Dean JP, Dokken JS, Habera M, Hale JS, Richardson CN, et al. DOLFINx: the next Generation FEniCS Problem Solving Environment. Zenodo. 2025. https://doi.org/10.5281/zenodo.18101307.
4. Rathgeber F, Ham DA, Mitchell L, Lange M, Luporini F, Mcrae ATT, et al. Firedrake: automating the Finite Element Method by Composing Abstractions. ACM Trans Math Software. 2017;43(3):1–27. https://doi.org/10.1145/2998441.

5. Bastian P, Blatt M, Dedner A, Dreier N-A, Engwer C, Fritze R, et al. The Dune framework: basic concepts and recent developments. Comput Math Appl. 2021;81:75–112. https://doi.org/10.1016/j.camwa.2020.06.007.
6. Dedner A, Kloefkorn R, Nolte M. Python Bindings for the DUNE-FEM Module. Zenodo. 2020. https://doi.org/10.5281/ZENODO.3706994.
7. Alnæs MS, Logg A, Ølgaard KB, Rognes ME, Wells GN. Unified form language: a domain-specific language for weak formulations of partial differential equations. ACM Trans Math Softw. 2014;40(2):1–37. https://doi.org/10.1145/2566630.
8. Ltd., P.S.: FEATool Multiphysics. 2025. Accessed: 2025-06-17. https://www.featool.com
9. Multiphenicsx. Accessed: 2025-06-17. https://multiphenics.github.io.
10. Bergersen A, Slyngstad A, Gjertsen S, Souche A, Valen-Sendstad K. turtleFSI: a Robust and Monolithic FEniCS-based Fluid-Structure Interaction Solver. J Open Source Softw. 2020;5(50):2089. https://doi.org/10.21105/joss.02089.
11. Öğüç M, Okyar AF, Khajah T. FeVAcS: a package for visualizing acoustic scattering from 1D periodic obstacles. Software Impacts. 2025;24:100756. https://doi.org/10.1016/j.simpa.2025.100756.
12. Quintino N, Tiago J. Opytimal—a Python/FEniCS Framework to Solve PDE-based Optimal Control Problems Considering Multiple Controls in 2D and 3D Domains. In Review. 2025. https://doi.org/10.21203/rs.3.rs-6546784/v1.
13. Wintzer A, Abali BE, Dadzis K. Multiphysics simulation of crystal growth with moving boundaries in FEniCS. Comput Methods Appl Mech Eng. 2025;437:117783. https://doi.org/10.1016/j.cma.2025.117783.
14. Kjeldsberg HA, Sundnes J, Valen-Sendstad K. A verified and validated moving domain computational fluid dynamics solver with applications to cardiovascular flows. Int J Numer Methods Biomed Eng. 2023;39(6):3703. https://doi.org/10.1002/cnm.3703.
15. Kyas S, Volpatto D, Saar MO, Leal AMM. Accelerated reactive transport simulations in heterogeneous porous media using Reaktoro and Firedrake. Comput Geosci. 2022;26(2):295–327. https://doi.org/10.1007/s10596-021-10126-2.
16. Dedner A, Kane B, Klöfkorn R, Nolte M. Python framework for hp-adaptive discontinuous Galerkin methods for two-phase flow in porous media. Appl Math Model. 2019;67:179–200. https://doi.org/10.1016/j.apm.2018.10.013.
17. Terschanski B, Klöfkorn R, Dedner A, Kowalski J. Stable across regimes: a mixed DG method for Darcy-Brinkman-Stokes type flows. Comput Methods Appl Mech Eng. 2025;442:117962. https://doi.org/10.1016/j.cma.2025.117962.
18. Dalcin LD, Paz RR, Kler PA, Cosimo A. Parallel distributed computing using Python. Adv Water Resour. 2011;34(9):1124–39. https://doi.org/10.1016/j.advwatres.2011.04.013.
19. Balay S, et al. PETSc/TAO Users Manual. 2025. https://doi.org/10.2172/2476320.
20. Parkinson JRG, Martin DF, Wells AJ, Katz RF. Modelling binary alloy solidification with adaptive mesh refinement. J Comput Phys X. 2020;5:100043. https://doi.org/10.1016/j.jcpx.2019.100043.
21. Kaaks BJ, Rohde M, Kloosterman J-L, Lathouwers D. An energy-conservative DG-FEM approach for solid-liquid phase change. Numer Heat Transfer Part B Fund. 2023;0(0):1–27. https://doi.org/10.1080/10407790.2023.2211231.
22. Terschanski B, Klöfkorn R, Dedner A, Kowalski J. Bryne: sustainable Prototyping of Finite Element Models - Software Release. https://doi.org/10.5281/zenodo.19603432.
23. Geuzaine C, Remacle J-F. Gmsh: a 3-D finite element mesh generator with built-in pre- and post-processing facilities. Int J Numer Meth Eng. 2009;79(11):1309–31. https://doi.org/10.1002/nme.2579.
24. Schlömer N. Meshio: tools for Mesh Files. Zenodo. 2024. https://doi.org/10.5281/zenodo.1173115.
25. Clarke L, Glendinning I, Hempel R. The MPI message passing interface standard. In: Decker KM, Rehmann RM, editors. Programming Environments for Massively Parallel Distributed Systems. Basel: Birkhäuser Basel; 1994. p. 213–8. https://doi.org/10.1007/978-3-0348-8534-8_21.
26. Alkämper M, Dedner A, Klöfkorn R, Nolte M. The DUNE-ALUGrid Module. Archive of Numer Softw. 2016;4(1):1–28. https://doi.org/10.11588/ans.2016.1.23252.
27. Bars L, Worster MG. Interfacial conditions between a pure fluid and a porous medium: implications for binary alloy solidification. J Fluid Mech. 2006;550(1):149. https://doi.org/10.1017/S0022112005007998.
28. Swaminathan CR, Voller VR. On the enthalpy method. Int J Numer Methods Heat Fluid Flow. 1993;3(3):233–44. https://doi.org/10.1108/eb017528.
29. Faden M, König-Haagen A, Brüggemann D. An Optimum Enthalpy Approach for Melting and Solidification with Volume Change. Energies. 2019;12(5):868. https://doi.org/10.3390/en12050868.
30. Krabbenhoft K, Damkilde L, Nazem M. An implicit mixed enthalpy-temperature method for phase-change problems. Heat Mass Transf. 2006;43(3):233–41. https://doi.org/10.1007/s00231-006-0090-1.
31. Nedjar B. An enthalpy-based finite element method for nonlinear heat problems involving phase change. Comput Struct. 2002;80(1):9–21. https://doi.org/10.1016/S0045-7949(01)00165-1.
32. Terschanski B, Klöfkorn R, Dedner A, Kowalski J. Bryne: sustainable Prototyping of Finite Element Models - Simulation Data. https://doi.org/10.5281/zenodo.15850111.
33. Vélez C, Khayet M, Ortiz De Zárate JM. Temperature-dependent thermal properties of solid/liquid phase change even-numbered n-alkanes: n-Hexadecane, n-octadecane and n-eicosane. Appl Energy. 2015;143:383–94. https://doi.org/10.1016/j.apenergy.2015.01.054.
34. Zhang P, Ma ZW, Wang RZ. An overview of phase change material slurries: MPCS and CHS. Renew Sustain Energy Rev. 2010;14(2):598–614. https://doi.org/10.1016/j.rser.2009.08.015.
35. Chourdakis G, Davis K, Rodenberg B, Schulte M, Simonis F, Uekermann B, Abrams G, Bungartz H, Cheung Yau L, Desai I, Eder K, Hertrich R, Lindner F, Rusch A, Sashko D, Schneider D, Totounferoush A, Volland D, Vollmer P, Koseomur O. preCICE v2: a sustainable and user-friendly coupling library [version 2; peer review: 2 approved]. Open Research Europe 2022. https://doi.org/10.12688/openreseurope.14445.2.