Fireframe: a Firedrake based Programmable Framework for solving coupled Partial Differential Equations*

Keer Mei[a,*], Matthew Piggott[a]

[a] *Imperial College London*

Abstract

Fireframe is a programmable framework built to streamline the process of coupling Partial Differential Equations (PDEs) across different subject areas and provide users with an intuitive interface to analyze science or engineering problems. Fireframe leverages the underlying Firedrake API's functionality to perform finite element discretisation. A similar framework has previously been developed using the FEniCS [1] library for analyzing turbulent flow models, but the FEniCS framework lacks code verification and generalization to other types of coupled PDE problems. Fireframe is able to automate the process of setting up the finite element method in Firedrake for users, and provides an interface with `sympy`'s symbolic mathematics for code verification using the method of manufactured solutions (MMS). Examples of weak formulations of the Navier-Stokes equation, a groundwater chemical transport equation, and radionuclide transport equations are presented and solved using Fireframe. Code verification by MMS is performed on the radionuclide transport problem and demonstrates adherence to theoretical orders of convergence in space and time.

Keywords: Firedrake, coupled partial differential equations, finite element method, FEniCS, UFL

Code metadata:

Release	v0.0.0
Firedrake version	latest
Link for latest Firedrake version	`https://www.firedrakeproject.org/download.html`
Legal Code License	MIT License
Code versioning system used	git, Github
Code repository	`https://github.com/msc-acse/acse-9-independent-research-project-kmei92`
Software code languages	Python (3.6.8)
Operating System	Linux, MacOS (unverified)
External dependencies	`numpy`(1.16.4), `sympy`(1.4), `matplotlib`(3.1.0), `jupyter`(1.0.0), `Firedrake`(latest)
Continuous Integration	Travis CI

1. Introduction

Fireframe is a programmable Python framework which utilizes the Firedrake finite element API [2] for solving problems involving coupled partial differential equations (PDEs). Currently, Firedrake users must have some degree of familiarity with its functionality as well as technical knowledge of the finite element

*Applied Computational Science and Engineering. Module 9: Independent Research Project.
*Corresponding author
URL: `keer.mei18@imperial.ac.uk.  Github:  kmei92` (Keer Mei)

method in order to set up new problems. Fireframe aims to provide clarity and ease of use to users who want to adopt Firedrake's finite element functionality by streamlining the setup process, including defining function spaces for trial and test functions. Additionally, Fireframe provides users with the ease of combining multiple coupled systems of PDEs and an interface to build manufactured solutions for the purpose of software verification. Fireframe only requires users to issue intuitive, high level instructions for setting up and solving their own coupled problems.

Fireframe automates the setup process by utilizing dictionary and string parsing. First, users define the problem variables being solved for by Fireframe (such as velocity, pressure, concentration, etc ...) according to the desired system of PDEs being discretised. Next, users define Firedrake parameters within a dictionary container for each of the problem variables. Fireframe assumes that any variational, or weak, formulations of the PDEs provided by users are written in valid Firedrake syntax, that is using the UFL (unified form language) domain-specific language [3, 4].

The structure of Fireframe involves two interdependent Python objects. `PDESubsystem` provides the individual variational forms of a PDE (or a system of PDEs) and `PDESystem` is responsible for solving an entire coupled system.

In the remainder of this paper, three example coupled PDE problems with increasing levels of complexity are solved using Fireframe. The examples serve to highlight the extent to which Fireframe can reduce programming complexity and provide a user-friendly interface for users to specify their own coupled problems. Software design is discussed, including overcoming challenges associated with Firedrake and object interactions between Fireframe's two essential classes. Code verification is demonstrated for a multi-species radionuclide transport problem using the method of manufactured solutions (MMS). Finally, conclusions, lessons learned, and suggestions for future improvements are provided for continuity of Fireframe development.

1.1. From the FEniCS framework to a more generalized framework

Fireframe is inspired by a FEniCS framework developed for numerical experiments using different turbulence models [1, 5]. When solving for turbulence, the trade-off between computationally efficient models such as the family of Reynolds-averaged Navier-Stokes (or RANS) models versus the precise but computationally expensive Direct Numerical Simulation is not always obvious [1, 6]. Depending on the problem, one model for turbulence may be more appropriate than another. The justification for a programmable framework lies in providing researchers with an efficient tool to create and analyze a variety of turbulence models simultaneously.

In a broader sense, turbulence models are simply an extension of the Navier-Stokes equations where additional variables are coupled within the solve. One example is the k-ϵ model, where two new variables termed the turbulence kinetic energy, k, and the energy dissipation rate, ϵ, [7] are coupled to the Navier-Stokes equations. Coupled systems of PDEs are common throughout classical physics problems and not restricted to only momentum transfer; they range from elasticity, heat and mass transfer to chemical reaction kinetics. A finite element framework that can be readily applied to any type of coupled problem would be a powerful tool. Fireframe provides an object oriented foundation for solving systems of PDEs by automating the process of setting up the finite element method, assembling variational forms, and solving for the numerical solution. The foundation of Fireframe is based on the existing Firedrake finite element library and relies on Firedrake's core functionality.

2. Background

Firedrake employs the Unified Form Language (UFL) [3, 4] which is a domain-specific language that contains built-in support for automatic differentiation and describing the variational forms of PDEs. This allows the user to develop models using high-level code that resembles closely the underlying mathematical equations.

Although the existing Firedrake API provides an intuitive interface for users to deploy the individual steps of the finite element method, it is not a trivial process. The knowledge to build a mesh, specify initial/boundary conditions, create multiple and/or mixed function spaces, specify solver types, as well as a multitude of other considerations makes using Firedrake as a research tool for coupled PDEs relatively complicated, especially for new users. A framework will simplify this process by giving users the convenience of only specifying high-level Firedrake commands without the need to manage how each individual component interacts with the others. An example of a similarly motivated framework has been implemented in FEniCS to solve turbulent flow models involving coupled PDEs [1, 5].

Similar to the FEniCS framework developed in [1, 5], users should ideally only be required to specify high-level programming instructions in few short lines of code. The goal of finite element analysis is ultimately to obtain practical solutions or conclusions to real life science and engineering problems. Removing programming barriers is desirable from both a user interface and an efficiency perspective. Fireframe seeks to streamline the process of coupling PDE problems by:

1. Automating the process of setting up function spaces, functions, boundary conditions, initial conditions, parameter values, etc, by only requiring users to specify the variables and the system of PDEs that are being solved.
2. Provide an existing repository of verified variational forms of prominent physics problems such as the Navier-Stokes equation. Users of Fireframe can contribute their own numerical schemes as well to further develop the number of available PDEs that can be deployed.
3. Create an interface to streamline the adoption of common Firedrake functionality so that users can freely test and employ different combinations.
4. Facilitate code verification through the MMS approach by using Sympy's symbolic mathematics.
5. Reduce the total amount of programming required in order to use Firedrake.

3. Software Design and Implementation

3.1. Objects architecture

Fireframe is divided into two main classes: `PDESubsystem` and `PDESystem`. The architecture of Fireframe and the names of the classes are motivated from the related FEniCS framework [5]. `PDESubsystem` is responsible for handling one specific set of PDEs. For example, the Navier-Stokes equations would be described by a `PDESubsystem`. The main purpose of `PDESubsystem` is to provide the variational forms of a standalone set of PDEs to the overall `PDESystem`.

The `PDESystem` class describes the overall coupled problem. Multiple `PDESubsystem`s can be coupled to a single `PDESystem`. A `PDESystem`'s main responsibility is to create the function spaces, trial functions, test functions, functions, and to solve the system of PDEs. Auxiliary functions of the `PDESystem` include setting initial conditions, adding new `PDESubsystem`s to itself, and defining the `PDESubsystem`s that are a part of the coupled problem. The `PDESystem` class also automates error calculations using the MMS approach. The overall procedure when initializing a `PDESystem` is to:

1. Create a `PDESystem` child object. In the examples given throughout this paper, the object is referred to as `pde_solver`. Due to the complexity of setting Dirichlet boundary conditions in Firedrake (see section 3.7), it is recommended users specify the boundary conditions for the problem through a member function written for `pde_solver`. The complexity of setting Dirichlet boundary conditions is also why `PDESystem` does not provide a function to create boundary conditions. It is also recommended that users overload `PDESystem`'s `setup_constants()` function in `pde_solver`.
2. The user must issue a set of Firedrake and simulation related parameters (e.g. finite element family, simulation time) in the form of a dictionary, referred throughout this paper as `solver_parameters`, for each corresponding problem variable to be included within a `PDESystem`.

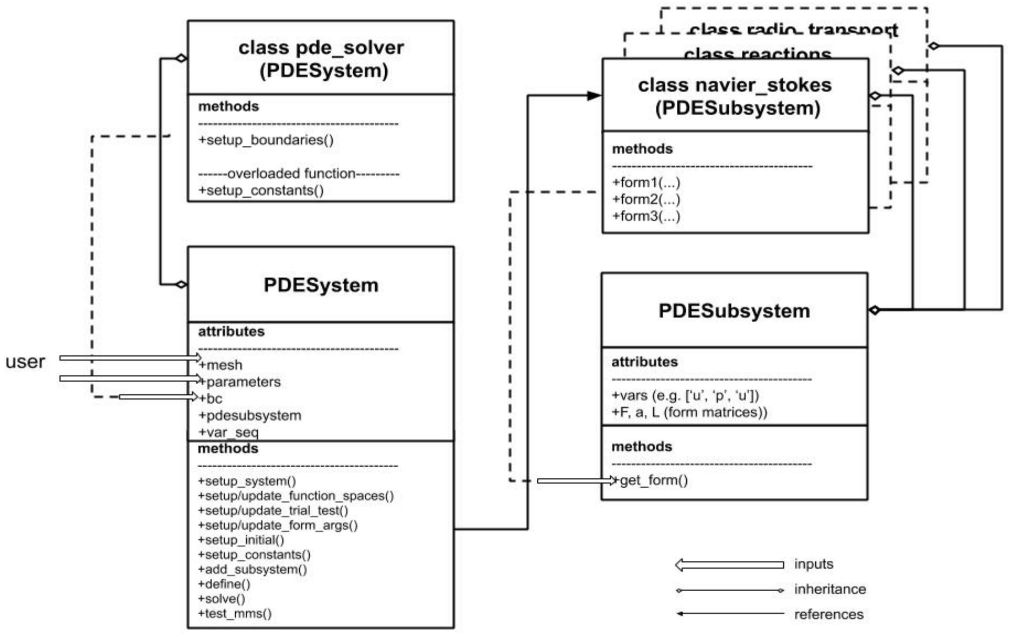


Figure 1: Fireframe architecture. Two main classes `PDESystem` and `PDESubsystem` with children classes and overloaded functions.

3. Once the Firedrake and simulation parameters have been set, the mesh should be loaded.
4. Using the parameters and mesh, users should initialize a `pde_solver` object, add any required subsystems (other coupled PDEs), and call `PDESystem`'s `define()` function from `pde_solver` to specify the order each subsystem is being solved.
5. Once all of the above has been accomplished, users may call `PDESystem`'s `solve()` function from `pde_solver` to compute the numerical solutions.

In the following sections, this procedure will be detailed step by step using the radionuclides transport problem described in section 4.3. Figure 2 illustrates the process of setting up and solving a `PDESystem`.

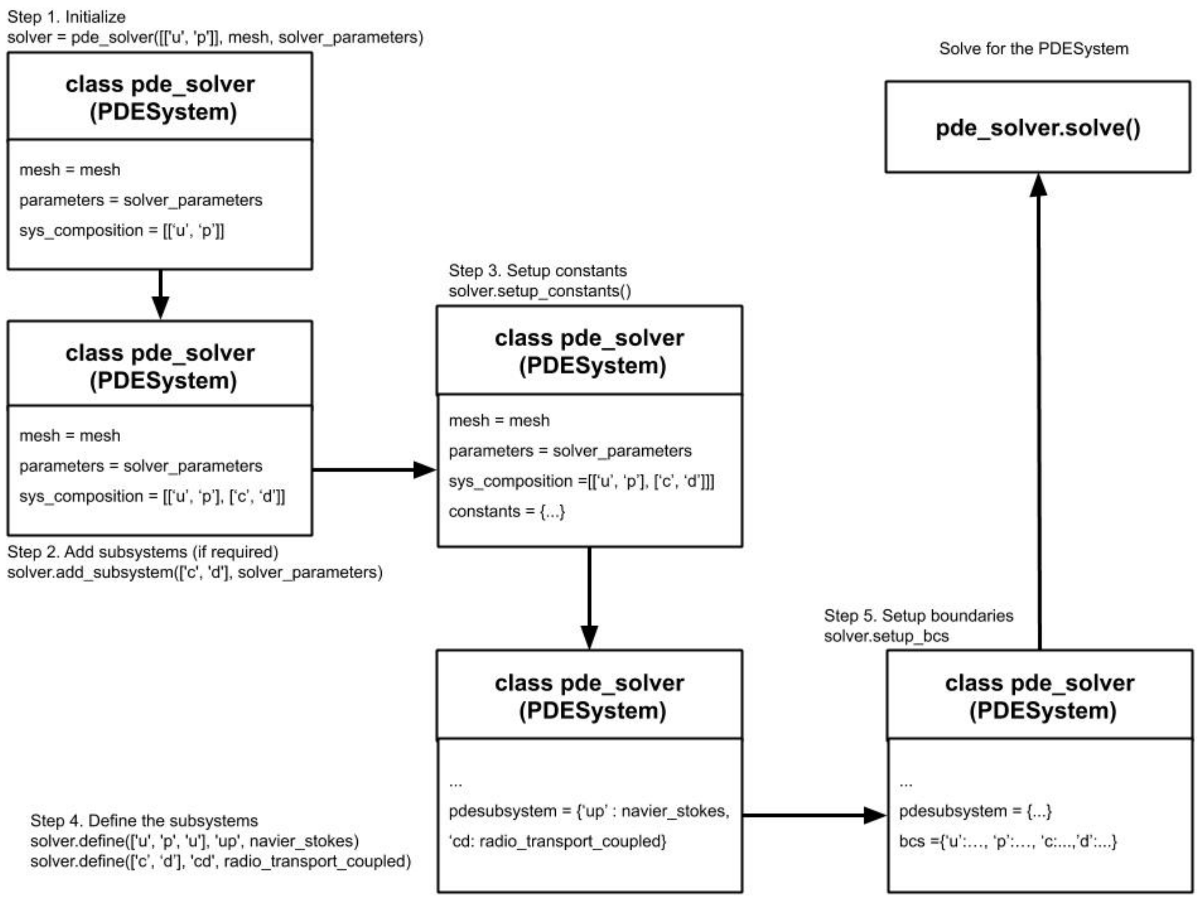

Step 1. Initialize
solver = pde_solver([['u', 'p']], mesh, solver_parameters)
class pde_solver (PDESystem)
mesh = mesh
parameters = solver_parameters
sys_composition = [['u', 'p']]
class pde_solver (PDESystem)
mesh = mesh
parameters = solver_parameters
sys_composition = [['u', 'p'], ['c', 'd']]
Step 2. Add subsystems (if required)
solver.add_subsystem(['c', 'd'], solver_parameters)
Step 3. Setup constants
solver.setup_constants()
class pde_solver (PDESystem)
mesh = mesh
parameters = solver_parameters
sys_composition =[['u', 'p'], ['c', 'd']]]
constants = {...}
Step 4. Define the subsystems
solver.define(['u', 'p', 'u'], 'up', navier_stokes)
solver.define(['c', 'd'], 'cd', radio_transport_coupled)
class pde_solver (PDESystem)
...
pdesubsystem = {'up' : navier_stokes, 'cd: radio_transport_coupled}
Step 5. Setup boundaries
solver.setup_bcs
class pde_solver (PDESystem)
...
pdesubsystem = {...}
bcs ={'u':..., 'p':..., 'c:...,'d':...}
Solve for the PDESystem
pde_solver.solve()

Figure 2: Fireframe flow chart. Step 1. Define mesh, parameters, and variables. Step 2. Add additional `PDESubsystem`s. Step 3. Setup constants to be used in variational forms. Step 4. Define the sequence of variables to be solved. Step 5. Set up boundary conditions. Step 6. Solve the `PDESystem`

3.2. Firedrake parameters

Despite the wide variety of coupled PDE type problems in general, solving them using Firedrake requires using many of the same Firedrake parameters. For example, in Firedrake users must always specify the polynomial degree of the finite element function spaces used to perform the discretisation and the finite element family (e.g. Continuous Galerkin 'CG' or Discontinuous Galerkin 'DG'). Users may also wish to specify other simulation parameters such as the time step and final time when solving time dependent problems, or select from Firedrake's built-in set of iterative methods [8] and preconditioners to be used to solve the linear systems that result from the discretisation. These parameters are always a part of the process of specifying a problem in Firedrake and providing an interface to access them is a necessary functionality of Fireframe. The parameters provide the `PDESystem` with knowledge on how to build the function spaces, the functions, as well as how to solve the PDEs.

Fireframe allows users to specify these Firedrake and simulation parameters by providing a default dictionary with `defaultdict` values (shown in table 1). The `defaultdict` container automatically generates a default value for any new key added to the dictionary, which is useful for reducing redundant programming when adding a new subsystem to a `PDESystem` with new problem variables being solved for.

Table 1: Default solver parameters provided by Fireframe. For a list of ksp_types (defining iterative solver parameters), see [8].

default_solver_parameters		
key	value	description
'degree':	defaultdict(lambda: 1)	the order of finite elements
'family':	defaultdict(lambda: 'CG')	the family of finite elements
'ksp_type':	defaultdict(lambda: ' ')	iterative method selection [8]
'precond':	defaultdict(lambda: ' ')	preconditioner selection
'order' :	defaultdict(lambda: 1)	the number of coupled variables within one PDE
'space' :	defaultdict(lambda: fd.FunctionSpace)	type of function space to be used
'T':	10.0	end time of simulation
'dt':	0.001	time step of simulation

Users may update the values of the default dictionary for their own problems by using a provided function named `recursive_update`. The reason why a dictionary container is used for Fireframe's parameters is because this structure provides a simple look up scheme for the Fireframe parameters associated with each problem variable being solved for in the problem. Accessing the Firedrake parameters requires Fireframe to only remember the names of the problem variables in its `PDESystem`.

For illustration, consider the radionuclide transport problem described in section 4.3. The following is an example of how to setup and update the default parameters:

```
solver_parameters = recursive_update(solver_parameters,
{'space': {'u': fd.VectorFunctionSpace, 'c': fd.MixedFunctionSpace, 'd' :
    fd.MixedFunctionSpace},
'degree': {'u': 2},
'order' : {'c': 3, 'd':3},
'ksp_type': {'u': 'gmres', 'p': 'gmres', 'c': 'gmres', 'd':'gmres'},
'precond': {'u': 'sor', 'p' : 'sor', 'c': 'sor', 'd':'gmres'},
'dt' : 0.05,
'T' : 50})
```

An important note: the `'family'` parameter is not specified in the example above nor have any of the parameters been set for pressure, p. When a parameter is not specified, the default dictionary value is used. For the problem illustrated in the example above, all finite elements used will be first order Continuous Galerkin (`'CG'`) finite elements. For example, the finite elements built for pressure will be on a `fd.FunctionSpace(mesh, 'CG', 1)`. The convenience of a default dictionary is that users will not need to specify any parameters that they do not wish to differ from default.

3.3. Obtaining the variational forms: `PDESubsystem`

Fireframe obtains the variational forms of each `PDESubsystem` through user created `PDESubsystem` child objects with defined variational form functions (e.x. `navier_stokes`). The child objects must define the variational form/forms through a class method called form. For example: `def form1(*args, **kwargs)`. The child objects must contain the same number of forms as the number of underlying equations it is responsible for. For example, in the Chorin projection scheme for the Navier-Stokes equations, there are three equations [9] to be solved. The corresponding `PDESubsystem` describing the Chorin scheme must also contain three forms. The form functions are to be numbered ascending from 1 to n, where n is the total number of forms. The snippet below illustrates the `navier_stokes(PDESubsystem)` object containing Chorin's projection method:

```
class navier_stokes(PDESubsystem):
        def form1(self, u_trl, u_tst, u_n, p_n, n, deltat, f, nu, **kwargs
            ):
                """
        Solving for intermediate velocity
        """
        def sigma(u, p):
            return 2*nu*fd.sym(fd.nabla_grad(u)) - p*fd.Identity(len(u))
        u_mid = 0.5 * (u_n + u_trl)
        Form = fd.inner((u_trl - u_n)/deltat, u_tst) * fd.dx \
        + fd.inner(fd.dot(u_n, fd.nabla_grad(u_mid)), u_tst) * fd.dx \
        + fd.inner(sigma(u_mid, p_n), fd.sym(fd.nabla_grad(u_tst))) * fd.dx \
        + fd.inner(p_n * n, u_tst) * fd.ds \
        - fd.inner(nu * fd.dot(fd.nabla_grad(u_mid), n), u_tst) * fd.ds \
        - fd.inner(f, u_tst) * fd.dx
        return Form

    def form2(self, p_trl, p_tst, p_n, u_, deltat, **kwargs):
        """
        Solving for pressure
        """
        Form = fd.inner(fd.nabla_grad(p_trl), fd.nabla_grad(p_tst)) * fd.dx \
        - fd.inner(fd.nabla_grad(p_n), fd.nabla_grad(p_tst)) * fd.dx \
        + (1/deltat) * fd.inner(fd.div(u_), p_tst) * fd.dx
        return Form

    def form3(self, u_trl, u_tst, u_, p_n, p_, deltat, **kwargs):
        """
        Updating velocity for divergency free condition
        """
        Form = fd.inner(u_trl, u_tst) * fd.dx \
        - fd.inner(u_, u_tst) * fd.dx \
        + deltat * fd.inner(fd.nabla_grad(p_ - p_n), u_tst) * fd.dx
        return Form
```

The method which `PDESubsystem` uses to obtain the variational forms is through Python's `eval()` function. Using the double asterisk ** magic variables in Python, the `PDESubsystem` can retrieve the required forms from the `PDESystem` by passing a dictionary of all the problem variables' names with their corresponding values via `eval(**self.form%d(**form_args, **constants)%i)`. Therefore, the `PDESystem` creates the dictionary of functions for each problem variable, and each `PDESubsystem` takes only the arguments they are responsible for to assemble the forms.

3.3.1. Naming convention, sequencing, and parameter consistency

One shortcoming of Fireframe is its dependency on the user being consistent in naming the problem variables. For example, if a user wants to define a new `PDESystem` to solve for the Navier-Stokes equation, they may choose to specify the velocity variable as u and the pressure variable as p when initializing the `PDESystem`. Following this convention, the form described in the `PDESubsystem` object must also be expressed using u and p; it would be incorrect to write the form function using v and q to describe velocity and pressure.

The `solver_parameters` must also define the Firedrake parameters for u and p. This is because the magic variable $**$ relies on having correct key values specified in the dictionary argument to correspond with the actual argument names written in the `form` functions.

Furthermore, the naming convention in the `PDESystem` class relies on problem variables' names to construct function spaces, trial, and test functions. All trial functions are named after problem variables with a `'_trl'` suffix and all test functions are named after problem variables with a `'_tst'` suffix. Current iterations of problem variables are named with a `'_n'` suffix and the next iteration of problem variables are named with a `'_'` suffix. For example, in the `navier_stokes` object highlighted above, `u_` refers to the next time step velocity, `u_n` refers to the current time step velocity, `u_trl` refers to the trial function on the velocity function space, and `u_tst` refers to the test function on the velocity function space. All variational forms described in the corresponding `PDESubsystem` must be written using this specific naming scheme.

One special exception is when users specify a `MixedFunctionSpace`. `MixedFunctionSpace`s are a way to express multiple combined finite element function spaces that contain multiple problem variables combined into one. Mixed spaces are useful for nonlinear equations that are written using the singular coupled problem variable. Treatment of `MixedFunctionSpace` is as follows:

- the `solver_parameter` key `['order']` specifies how many individual functions spaces are coupled together in a `MixedFunctionSpace`. Alternatively, users can create their own `MixedFunctionSpace` by passing in a list of function spaces. For example, in the shallow water equation described in section 4.3.1 velocity and wave height are coupled together inside a mixed problem variable called z. The `'space'` parameter for z is `'space' : {'z' : [fd.VectorFunctionSpace, fd.FunctionSpace]}`.
- the trial and test functions for a sub component inside a `MixedFunctionSpace` variable will have the suffix `'_trl%i'` and `'_tst%i'` where `i` ranges from 1 to `['order']`, respectively. For example, in the chemical transport problem described in section 4.2, the trial and test functions for the first chemical species will be `'c_trl1'` and `'c_tst1'`, respectively.
- the functions of the current iteration and next iteration of a sub component in a `MixedFunctionSpace` variable will have the suffix `'_n%i'` and `'_%i'` where `i` ranges from 1 to `['order']`, respectively. For example, in the chemical transport problem described in section 4.2, the current and future iteration of concentration for the first chemical species will be `'c_n1'` and `'c_1'`, respectively.

If a `PDESystem` is a coupled system, such as the radionuclides transport problem where velocity from a hydrodynamics model is coupled to the species transport of the radionuclides, then the same naming convention must be maintained across both `PDESubsystems`. For example, the velocity term expressed in the Navier-Stokes `PDESubsystem` and the radionuclide transport `PDESubsystem` must both be u. Again, the reason for this is because the `PDESystem` provides the same functions to each `PDESubsystem`. If the naming convention is inconsistent, the magic variables $**$ will not recognize the arguments from the `PDESystem` and the forms will be lost in the `PDESubsystem`.

Another requirement is that the forms within a `PDESubsystem` are numbered in the sequence they should be solved for. For example, in the Chorin projection scheme used to solve the Navier-Stokes equations, `form1` must be the variational form associated with solving for an intermediate velocity [9].

3.4. Defining the function spaces, functions, and subsystems: `PDESystem`

The `PDESystem` class is initialized with three arguments: a variables composition list, a Firedrake mesh, and a parameters dictionary (`solver_parameters`). The mesh sets the domain of the problem. Creating the mesh is the responsibility of the user, and Firedrake supports a variety of different formats [10]. The user must load the mesh through Firedrake and pass it as an argument when initializing a `PDESystem`.

The parameters dictionary, previously mentioned in section 3.2, identifies the type of function spaces (e.g. FunctionSpace, VectorFunctionSpace, MixedFunctionSpace), order, degree of finite elements, etc.., that the `PDESystem` will create for each problem variable in the variables composition list.

The variables composition list contains the names of the problem variables that the `PDESystem` is intended to solve for and it is a Python `list` of `list` container. In a coupled problem that includes velocity, pressure, and concentration for example, the composition will include `[['u', 'p', 'c']]` representing velocity, pressure, and concentration respectively. Each internal `list` represents a subsystem.

The advantage of the Fireframe framework becomes obvious when the user decides to specify numerous different variables with varying function spaces. `PDESystem` will automatically generate all of the required functions to be used in the variational forms without any further user input. The `PDESystem` will also breakdown mixed function spaces into individual sub spaces. All of the generated functions are stored in a dictionary called `form_args`, which is an attribute of `PDESystem`, and then passed into each `PDESubsystem`. The framework is thus able to completely eliminate the process of setting up function spaces and functions by the user.

For example, the variable composition of the radionuclides problem is given as `[['u', 'p'], ['c', 'd']]`. Here, there are two internal lists which correspond to two separate subsystems. The code to initialize the `PDESystem` for this problem is:

```
# either choose to initialize the Navier-Stokes problem first
# and then add the transport problem
solver = pde_solver([['u', 'p']], mesh, solver_parameters)
solver.add_subsystem(['c', 'd'], solver_parameters)

# or, initialize the entire system altogether
solver = pde_solver([['u', 'p'], ['c', 'd']], mesh, solver_parameters)
```

`solver` is not initialized with the entire `[['u', 'p', 'c', 'd']]` because only separate `PDESubsystem` objects for the Navier-Stokes and radionuclide transport models are provided. This is the basis of coupling PDEs as either the transport model or the Navier-stokes model can be switched for another at the user's discretion. An example is demonstrated in section 4.3.1 where the Navier-stokes equation is switched with the shallow water equations [11]. Users may decide to create a `PDESubsystem` that includes the variational forms for the entire problem, but this approach would eliminate the flexibility of coupling.

3.5. Defining Constants

Fireframe allows users to create a dictionary of constants to be used in the variational forms described in a `PDESubsystem`. A dictionary is required because the `PDESubsystem` will parse this information when retrieving the forms by calling `eval(**self.form%d(**form_args, **constants)%i)`. This setup provides convenience as many variables may be shared among different `PDESubsystems` within one `PDESystem` and it would be tedious to declare the value of each variable for every form inside a `PDESubsystem`.

3.6. Defining the PDEs: interaction between `PDESystem` and `PDESubsystem`

Before attempting to solve a `PDESystem`, it is insufficient to only initialize the problem with a variables composition list, mesh, and parameters dictionary. Users of Fireframe must actively define the subsystems that are to be included in the `PDESystem`. This is required because the `PDESystem` at this stage does not have information of which `PDESubsystem` objects it should retrieve the variational forms from.

The interface between the entire coupled problem described by the `PDESystem` and the standalone PDEs described by `PDESubsystem` is through `PDESystem`'s `define(self)` method. Through `define(self)`, a `PDESystem` selects `PDESubsystem` objects to provide variational forms responsible for solving the problem variables.

All `PDESystems` have an attribute called `self.pdesubsystem` which is a dictionary container that stores `PDESubsystem` objects. Through this interface, the `PDESystem` can retrieve and collect all of the forms that have been created by each individual subsystem to be used when solving for the entire coupled problem.

3.7. *Challenges associated with time-dependent boundary conditions*

Fireframe is designed with support for different types of boundary conditions. Certain types of boundary conditions such as Neumann boundary conditions can be expressed directly within the variational forms [12] and are maintained inherently by the expressions written inside the form functions of `PDESubsystem` objects. To impose Dirichlet boundary conditions require using Firedrake's `fd.DirichletBC(...)` method.

Difficulties arise when attempting to impose time-dependent Dirichlet boundary conditions. This is because the object returned from calling `fd.DirichletBC(...)` is a fixed instance and will not update according to the varying time value of the problem. Therefore, Dirichlet boundary conditions must be instantiated at every time step if they are time-dependent.

Another difficulty associated with imposing Dirichlet boundary conditions is related to the numerical method for solving PDEs. For example in the Chorin projection method, a no-slip ($u = 0$) Dirichlet boundary condition may be imposed on equation 3 to model flow within a channel, as highlighted in figure 5. Next, a Dirichlet boundary condition is required for the inlet and/or outlet pressures in equation 4 for the same channel flow problem. However, a Dirichlet boundary condition is not required to be maintained for solving equation 5 in the same channel flow problem. As boundary conditions are set based on the names of the variables, the `PDESystem` must differentiate when Dirichlet boundary conditions can be applied. For this reason, `PDESystem` does not have its own `setup_bcs` function and it is up to the user to define the appropriate Dirichlet boundaries when creating their own `pde_solver` objects.

To address the complexity of setting up boundary conditions, a `PDESystem`'s `self.bc` attribute is only initialized when a subsystem is first defined through the `define` method. This is because the variable sequence will inform if there are any repeat variables. Below is an example to illustrate how to setup Dirichlet boundary conditions:

```
1  class pde_solver(PDESystem):
2      def __init__(self, comp, mesh, parameters):
3          PDESystem.__init__(self, comp, mesh, parameters)
4
5      def setup_bcs(self):
6          x, y = fd.SpatialCoordinate(self.mesh)
7
8          bcu = [fd.DirichletBC(self.V['u'], fd.Constant((0,0)), (10, 12)),
                # top-bottom
9            fd.DirichletBC(self.V['u'], ((1.0*(y - 1)*(2 - y))/(0.5**2) ,0),
                 9)] # inflow
10         bcp = [fd.DirichletBC(self.V['p'], fd.Constant(0), 11)]  # outflow
11
12         self.bc['u'][0] = [bcu, None, None, None,'fixed']
13         self.bc['p'] = [[bcp, None, None, None, 'fixed']]
```

The above example highlights some important conventions when defining boundary conditions that users must be aware of:

1. `self.V['u']` refers to the function space that velocity lies on. `self.V` itself is a Python dictionary that contains the function spaces created for each problem variable. `self.V` is an attribute of every `PDESystem`.
2. The conditions inside a Dirichlet boundary may depend on the mesh and coordinates used. Firedrake's `SpatialCoordinate` function retrieves the coordinate arrays when implementing spatially dependent expressions.
3. It is the user's responsibility to set the boundary on the correct location. For example, (10, 12) on line 8 refers to the upper and lower boundaries of the mesh in this example. The identifiers of the boundary will depend on how the users have created the mesh and this is knowledge that is not

provided inherently. Therefore, users must first be aware of what mesh is to be used in the problem before defining boundary conditions.

4. In the above example, a boundary is imposed on `self.bc['u'][0]`. The total length of the boundary condition associated with each problem variable is equal to the number of times it appears in the variable sequence (i.e. how many times it is being solved for) multiplied by how many subspaces are represented by that problem variable. Referring to section 3.2, `'u'` is specified with an `order` of '1' to signify it contains only one subspace, and it is being solved for twice in the Chorin's projection scheme. Therefore, the total length of `self.bc['u']` is 2. The index `[0]` on line 12 indicates that only the first equation solving for `'u'` is imposed with a boundary condition.

Each boundary condition is initialized as a fixed `list` with a length of 5. The convention is as follows:

1. index[0] refers to the `fd.DirichletBC(...)` object being applied.
2. index[1] is a Firedrake conditional, mathematical expression, or constant value that describes the value of the boundary condition.
3. index[2] refers to the location of the boundary to be applied. For example, `'on_boundary'` refers to all boundary of the domain. On lines 12 and 13, index[2] appears as `None`. This is because the boundary conditions are not going to be updated so it is unnecessary to declare the locations.
4. index[3] refers to the specific subspace of a `fd.MixedFunctionSpace`, if applicable.
5. index[4] must be either of two options: `'fixed'` or `'update'`. `'fixed'` informs the `PDESystem` that this boundary is not time dependent and will not be updated at each time step; `'update'` requires the specific expression declared in index[1] to be reapplied at every time step, on the location specified in index[2] (and if required, in the subspace specified by index[3]).

Strict adherence to this convention of setting up boundary conditions is required for Fireframe to obtain the correct numerical solutions. Notably, this system is complicated and can be difficult to adhere to even for experienced users of Firedrake. However, the current implementation is a method that captures the variance of Dirichlet boundaries that can be applied.

3.8. Design implementation, changes, and justification

Originally, Fireframe was developed so that the individual `PDESubsystem` objects would be responsible for creating problems and solvers objects by leveraging Firedrake's built-in `variational_solver` module [13]. This approach increases initialization time as the `PDESystem` will wait and collect all of the solver objects created by its `PDESubsystems`. The advantage of the old approach is that it reduces the time required for performing time integration once all of the solvers have been specified. However, this approach is not suitable for the scenario of a changing Dirichlet boundary condition that updates at each time step. For example, when attempting code verification by MMS, a time-varying Dirichlet boundary condition may be imposed on the manufactured variable.

The reason why this original approach was unsuitable is that the solver objects created by Firedrake's `LinearVariationSolver/NonlinearVariationalSolver` are objects that have been instantiated with a fixed set of boundary conditions. To maintain a time dependent Dirichlet boundary condition, the framework would have to recreate the problem and solver objects once again and update the boundary conditions. Intuitively, this removes the entire speed advantage of predefined solver objects.

Fireframe was updated to its current design where `PDESubsystems` are only responsible for providing the variational forms to the overall `PDESystem`. The `PDESystem` takes on the responsibility of setting up the solving each form and if necessary, updates the boundary conditions when doing so. While this new approach provides more flexibility to the framework, it imposes a penalty during run time as the `PDESystem` will have to try a number of different combinations of parameters and linear/non-linear solvers depending on how the user has chosen to setup the problem. Another disadvantage of the new design is the convoluted workaround for setting up Dirichlet boundary conditions.

4. Example problems and test cases

To highlight the flexibility of using Fireframe, three example problems are presented in this section. First, an introduction to the Chorin's scheme for solving the Navier-Stokes equations is presented. Next is a problem that couples the velocity calculated from Chorin's scheme with a chemical reactions transport model. The final examples couple velocity calculated from the Chorin's scheme and the shallow water equations to a radionuclide transport model that includes phase transfers and radioactive decay.

4.1. Example: Navier-Stokes equation and Chorin Projection

In the field of fluid mechanics, one of the most predominant problems is solving the incompressible Navier-Stokes momentum equation along with the continuity equation. The incompressible Navier-Stokes equation itself is a PDE containing both velocity u and pressure p terms:

$$\frac{\partial u}{\partial t} + u \cdot \nabla u = \frac{1}{\rho}\nabla p + \mu \nabla^2 u, \tag{1}$$

$$\nabla \cdot u = 0, \tag{2}$$

where t is the time variable, u is velocity, p is pressure, ρ is density, and μ is viscosity. One scheme for numerically solving the Navier-Stokes equations is Chorin's projection method [14, 9], which breaks down the Navier-Stokes and continuity equations into the following three equations. First, an intermediate velocity is calculated using the Navier-Stokes momentum equation while ignoring the pressure term. The variational form of this PDE is given by:

$$\int_\Omega \frac{u^* - u^{n-1}}{\triangle t} v dV + \int_\Omega u^{n-1} \cdot \nabla u^{n-1} v dV = \int_\Omega -\mu \nabla u^{n-1} \cdot \nabla v dV, \tag{3}$$

Next, the pressure term is calculated using the intermediate velocity:

$$\int_\Omega \nabla p^n \cdot \nabla q dV = -\int_\Omega \frac{(\nabla \cdot u^*) q}{\triangle t} dV, \tag{4}$$

Finally, the new velocity at time n is updated by using the values of the intermediate velocity and pressure calculated previously:

$$\int_\Omega u^n v dV = \int_\Omega u^* v dV - \int_\Omega \triangle t \nabla p^n v dV, \tag{5}$$

where Ω is the domain of the problem, V is the volume of the integral, Δt is the time step, v is the test function on the velocity function space, q is the test function on the pressure function space, u^* is an intermediate velocity, u^{n-1} is the previous iteration velocity, u^n is the current iteration velocity, and p^n is the current iteration pressure. The above equations written in UFL syntax are illustrated in section 3.3. The code below is an example of how to create a `PDESystem` using the `navier_stokes` object:

```
# first, we update our solver parameters and instruct it
# that we are building a system that includes velocity and pressure
    variables
solver_parameters = recursive_update(solver_parameters,
{'space': {'u': fd.VectorFunctionSpace},
'degree': {'u': 2},
'ksp_type': {'u': 'gmres', 'p': 'gmres'},
7  'precond': {'u': 'sor', 'p':'sor'},
8  ...})
9  # we then define a mesh (domain that our problem exists in)
10 mesh = fd.Mesh("../../meshes/flow_past_cylinder.msh")
11 # we create our solver, provide the variable composition, mesh, and
       parameters and tell our solver to solve the system
12 solver = pde_solver([['u', 'p']], mesh, solver_parameters)
13 solver.setup_constants()
14 solver.define(['u', 'p', 'u'], 'up', navier_stokes)
15 solver.setup_bcs()
16 solver.solve(time_update=False)
```

There are several important naming schemes presented in the example above that users must be aware of. The subsystem name `'up'` on line 14 refers to the entire Navier-Stokes system, as the Navier-Stokes equation is a PDE involving both velocity u and pressure p. This corresponds to our variable composition when initializing `pde_solver` by passing in the variables list `['u', 'p']`.

Line 14 (`solver.define(['u', 'p', 'u'], 'up', navier_stokes)`) includes more syntactic details. The first argument `['u', 'p', 'u']` instructs the `solver` that in this `'up'` system, the `solver` must first calculate u, then p, and finally u again in this order. This sequencing is important because it corresponds to the steps of the Chorin projection method described in equations 3 to 5. The argument `navier_stokes` instructs the solver that it should retrieve the variational forms from the `PDESubsystem` object called `navier_stokes`. Therefore, the `navier_stokes` object must have the three forms associated with each step of Chorin's scheme.

4.2. Groundwater transport and reaction kinetics

Chemical mass balance in groundwater transport is an important area of study for hydrologists wishing to maximize the use of groundwater reservoirs or study contamination events of a chemical spill [15]. The modelling of the chemicals' mass balance provides insight on how each chemical species will behave over time inside a domain. The complexity of this problem is that the mass balance of each chemical species are also functions of advection (the flow of water) and may be functions of the others' concentration. Therefore, the chemicals mass balance equations must be coupled to another equation that provides a velocity term (for example, the Navier-Stokes equation). As the Navier-Stokes equation has already been discussed in section 4.1, this section will focus solely on the coupled mass balance equations.

Consider the scenario where two chemical reactants coexist in a flow regime and together they facilitate an irreversible first-order reaction to form a third product [16, 17]. The system of PDEs to describe the mass balance for all three chemical components are as follows:

$$\frac{\partial c_1}{\partial t} + u \cdot \nabla c_1 - \nabla \cdot (D \nabla c_1) = f_1 - K c_1 c_2, \tag{6}$$

$$\frac{\partial c_2}{\partial t} + u \cdot \nabla c_2 - \nabla \cdot (D \nabla c_2) = f_2 - K c_1 c_2, \tag{7}$$

$$\frac{\partial c_3}{\partial t} + u \cdot \nabla c_3 - \nabla \cdot (D \nabla c_3) = f_3 + K c_1 c_2 - K c_3, \tag{8}$$

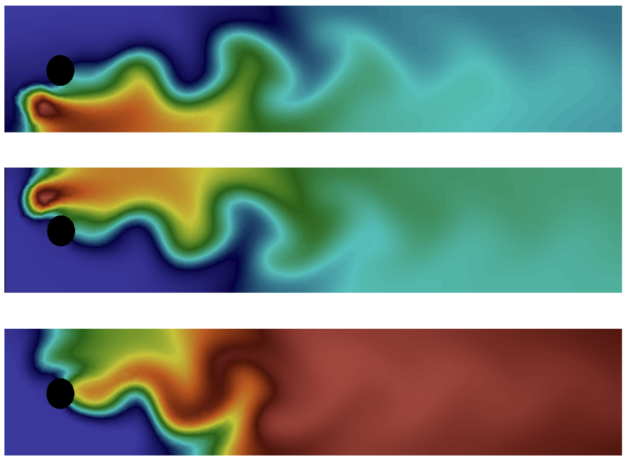
where t is the time variable, c is concentration, u is velocity, D is a diffusion coefficient, f is a source/sink term, K is a first order reaction coefficient, and subscripts 1 and 2 refer to chemical reactants and subscript 3 refers to a chemical product. To solve this problem, it is necessary to couple the chemical mass balance equations to a flow model that provides the velocity term u, such as the Navier-Stokes equation. The `PDESystem` class provides an `add_subsystem` function to couple the reactions equations to the `navier_stokes` object created previously,

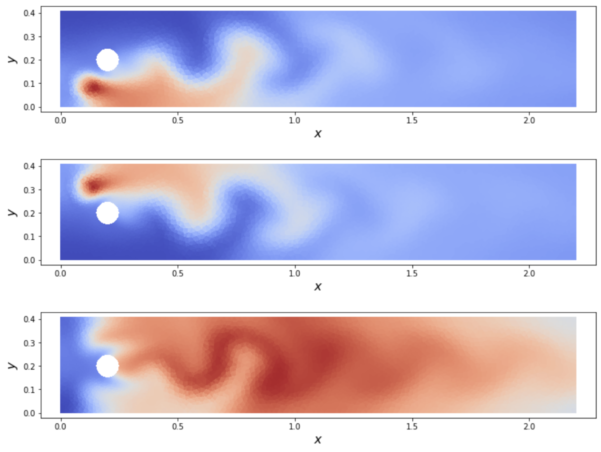
```
1  # update the parameters
2  solver_parameters = recursive_update(solver_parameters,
3  {'space': {'u': fd.VectorFunctionSpace, 'c' : fd.MixedFunctionSpace},
4  'degree': {'u': 2 },
5  'order' : {'c' : 3},
6  'ksp_type': {'u': 'gmres', 'p': 'gmres', 'c':'gmres'},
7  'precond': {'u': 'sor', 'p' : 'sor', 'c':'sor'},
8  'dt' : 0.0005,
9   'T' : 1.0 })
10
11 # load the mesh
12 mesh = fd.Mesh("../../meshes/cylinder1.msh")
13 # declare a new pde_solver object with a velocity and pressure variable
14 solver = pde_solver([['u', 'p']], mesh, solver_parameters)
15 # add a concentration system to the pde_solver object
16 solver.add_subsystem('c', solver_parameters)
17 # set up constants
18 solver.setup_constants()
19 # define the pdesystems and variable sequence
20 solver.define(['u', 'p', 'u'], 'up', navier_stokes)
21 solver.define(['c'], 'c', reactions)
22 # setup boundary conditions
23 solver.setup_bcs()
24 # solve
25 solver.solve()
```

One discrepancy between the mass balance equations and the Navier-Stokes equation is that the concentration of each chemical species is coupled with another (i.e. the concentration of the second chemical appears in the mass balance of the first chemical, etc). Therefore, this is a nonlinear PDE and one method to solve for this type of problem is to leverage Firedrake's `MixedFunctionSpace` functionality. This is reflected when updating the `solver_parameters` on line 3 to specify that the concentration variable `'c'` lies in a `MixedFunctionSpace` of `'order':{'c' : 3}`.

This coupled transport model highlights one advantage of adopting the Fireframe framework: once a set of PDEs has been implemented in Fireframe, coupling additional sets of PDEs onto the existing system becomes a trivial procedure. The only requirement is that users add the additional `PDESubsystem` in sequence. For example, in this reactions transport model, the reactions system `'c'` is added onto the Navier-Stokes system `'up'` because the velocity is required to be solved before the concentrations of chemicals.

Figure 3b shows Fireframe's solution to the chemical transport model, which matches closely to the solution given in the official FEniCS documentation [17]. Small discrepancies in the concentration profile can be attributed to the different meshes used between the two implementations, as the numerical solution to this problem is sensitive to the local mesh refinement near the cylinder.


(a) Image obtained from FEniCS tutorial [17].



(b) Calculated results from Fireframe.

Figure 3: Reactants concentration profiles. Top is reactant 1, middle is reactant 2, and bottom is product 3 at time = 5 seconds.

4.3. Kinetics model: Radionuclides transport in 3 phases

A more complicated coupled problem is the transport of radionuclides in water systems. Many types of transport models such as box models and kinetic models have been proposed [18, 19]. Despite the availability of the different types of transport models, the problem reduces to the coupling of a hydrodynamics model (flow of water) with a transport model (mass balance of the radionuclides). The following are 2D depth-averaged hydrodynamic equations described in [19]:

$$\frac{\partial z}{\partial t} + \frac{\partial}{\partial x}(Hu_x) + \frac{\partial}{\partial y}(Hu_y) = 0, \tag{9}$$

$$\frac{\partial u_x}{\partial t} + u_x\frac{\partial u_x}{\partial x} + u_y\frac{\partial u_x}{\partial y} + g\frac{\partial z}{\partial x} - \omega u_y + \frac{\tau_{ux}}{\rho H} = A\left(\frac{\partial^2 u_x}{\partial x^2} + \frac{\partial^2 u_x}{\partial y^2}\right), \tag{10}$$

$$\frac{\partial u_y}{\partial t} + u_x\frac{\partial u_y}{\partial x} + u_y\frac{\partial u_y}{\partial y} + g\frac{\partial z}{\partial x} + \omega u_x + \frac{\tau_{uy}}{\rho H} = A\left(\frac{\partial^2 u_y}{\partial x^2} + \frac{\partial^2 u_y}{\partial y^2}\right), \tag{11}$$

where subscripts x and y are spatial coordinates, t is the time variable, z is the displacement of water surface above the mean sea level, D is the depth of water below the mean sea level, $H = D + z$ is the total water depth, u is the velocity, g is gravity, ω is the Coriolis parameter, ρ is the density of water, τ_{ux} and τ_{uy} are sea bed friction stresses, and A is the horizontal eddy viscosity [19]. The velocity obtained from solving these hydrodynamic equations are used in the kinetic models for radionuclide transport.

The kinetic models used to describe the transport of radionuclides in [19] all involve transfers between the dissolved (water) and sediment (soil) phase. The complexity of the kinetic models and what separates this problem from the chemical reactions transport problem in section 4.2 is that in each phase there are multiple species of radionuclides. The kinetic model is a system of nonlinear PDEs as opposed to the single nonlinear PDE described in by the chemical reactions problem in section 4.2. Consider the 1-step reversible kinetic model between the dissolved phase and sediment phase:

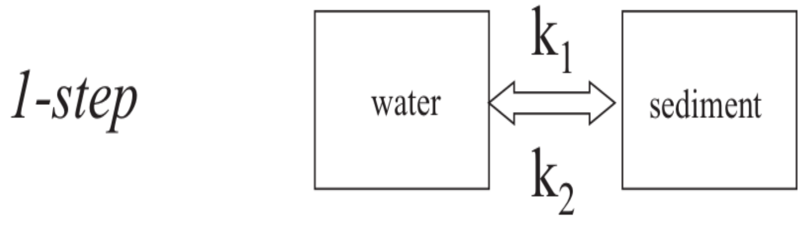


Figure 4: 1-step kinetic model between dissolved phase and sediment phase [19]. Not shown is an intermediate transfer from the dissolved into a suspended phase.

The system of PDEs to quantify the concentration of each of the radionuclide species in each phase is written as follows [19]:

$$\begin{aligned}\frac{\partial(HC_{d,i})}{\partial t}+\frac{\partial(u_xHC_{d,i})}{\partial x}+\frac{\partial(u_yHC_{d,i})}{\partial y}&=\frac{\partial}{\partial x}\left(HK_d\frac{\partial C_{d,i}}{\partial x}\right)+\frac{\partial}{\partial y}\left(HK_d\frac{\partial C_{d,i}}{\partial y}\right)\\&\quad-k_1C_{d,i}H+k_2C_{s,i}H+k_2A_{s,i}L\rho_sF\phi-\lambda_iC_{d,i}H,\end{aligned} \tag{12}$$

$$\begin{aligned}\frac{\partial(HC_{s,i})}{\partial t}+\frac{\partial(u_xHC_{s,i})}{\partial x}+\frac{\partial(u_yHC_{s,i})}{\partial y}&=\frac{\partial}{\partial x}\left(HK_d\frac{\partial C_{s,i}}{\partial x}\right)+\frac{\partial}{\partial y}\left(HK_d\frac{\partial C_{s,i}}{\partial y}\right)\\&\quad+k_1C_{d,i}H-k_2C_{s,i}H+(eros-dep)-\lambda_iC_{s,i}H,\end{aligned} \tag{13}$$

$$\frac{\partial A_{s,i}}{\partial t}=k_1\frac{C_{d,i}H}{L\rho_sF}-k_2A_{s,i}\phi+(dep-eros)-\lambda_iA_{s,i}, \tag{14}$$

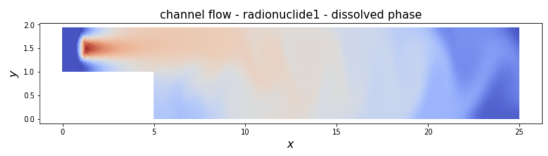
where C_d is the specific activity in the dissolved phase, C_s is the specific activity in the suspended phase, A_s is the specific activity in the sediment phase, K_d is a diffusion coefficient, k_1 and k_2 are kinetic transfer coefficients, L is a mixing depth in the sediment, ρ_s is the sediment bulk density, ϕ is a correction factor, F is a fraction of small particles, λ is a radioactive decay coefficient, *eros* is an erosion term, *dep* is a deposition term, and $i = 1, \ldots, n$ refers to each species of radionuclides. The solution to the hydrodynamics model provides the water currents (or velocity) at each point of the model domain, which are then used to solve for the transport equation [19].

4.3.1. Coupling flow and transport models

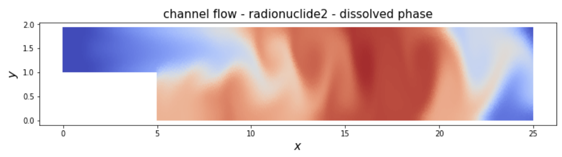
For the analysis presented, the kinetic model is simplified by removing the deposition (*dep*) and erosion (*eros*) terms and is assumed to be depth independent (no longer dependent on H). A simplified hydrodynamics model is incorporated by ignoring the effects of the Coriolis force. This essentially reduces the problem into the shallow water equation (equation 1 described in [11]). To highlight the flexibility of Fireframe, the kinetic model is coupled to both the Navier-Stokes equation as well as to the shallow water equation.

Implementing the radionuclides transport model is more complicated than the chemical reaction-transport model as it involves specifying multiple `MixedFunctionSpaces`. The code snippet below combines the Navier-Stokes equations with the kinetics model:

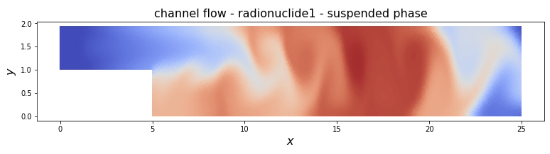
```
# update the parameters
solver_parameters = recursive_update(solver_parameters,
{'space': {'u': fd.VectorFunctionSpace, 'c': fd.MixedFunctionSpace, 'd' :
    fd.MixedFunctionSpace},
...
'order' : {'c': 3, 'd':3},
...)
#load mesh
mesh = fd.Mesh("../../meshes/step_long.msh")
# add subsystems for navier stokes and radio_transport
solver = pde_solver([['u', 'p']], mesh, solver_parameters)
solver.add_subsystem(['c', 'd'], solver_parameters)
#setup constants
solver.setup_constants()
# define subsystems and variable sequence
solver.define(['u', 'p', 'u'], 'up', navier_stokes)
solver.define(['c', 'd'], 'cd', radio_transport_coupled)
# setup boundary conditions
solver.setup_bcs()
#solve
solver.solve()
```



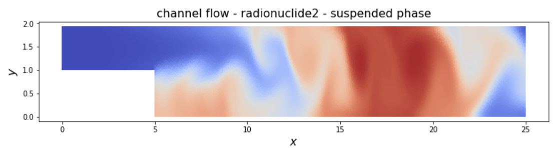
(a) Dissolved phase



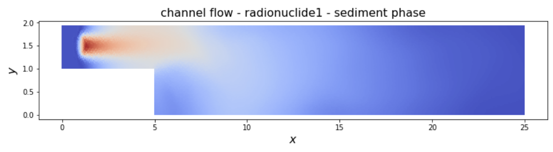
(b) Dissolved phase



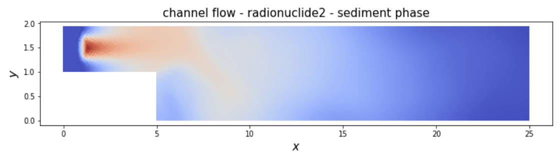
(c) Suspended phase



(d) Suspended phase



(e) Sediment phase



(f) Sediment phase

Figure 5: Visualization of radionuclide 1 (left) and 2 (right) concentration profiles over a simulation period of 50(s) in the coupling of the Navier-stokes equation to the radionuclide transport kinetics model.

In the example above, it is assumed that there is no reverse transfer from the sediment and suspended phase back into the dissolved phase ($k_2 = 0$). The radionuclide concentration profile in the sediment phase appears stationary, as it is not affected by advection. Another observation is that the second radionuclide concentration begins to increase as the first radionuclide decays, which indicates the presence of radioactive decay.

Coupling the kinetic model to the shallow water equation [11] is similar. One adjustment to be made is to describe a special function space that is a combination of a `VectorFunctionSpace` and a `FunctionSpace`. A way to create this special function space is to pass in a `list` into the `solver_parameters`. A revised `PDESubsystem` called `radio_transport_hydro` is used as its form is written with the velocity term calculated from the shallow water equation:

```
# update the parameters
solver_parameters = recursive_update(solver_parameters,
{
'space': {'z': [fd.VectorFunctionSpace, fd.FunctionSpace], # special
    function space
          'c': fd.MixedFunctionSpace, 'd' : fd.MixedFunctionSpace},...}
)
# load mesh
mesh = fd.Mesh("../../meshes/gibraltar.msh")
# add subsystems for navier stokes and radio_transport
solver = pde_solver([['z']], mesh, solver_parameters)
solver.add_subsystem(['c', 'd'], solver_parameters)
# setup constants
solver.setup_constants()
# define subsystems and variable sequence
solver.define(['z'], 'z', depth_avg_hydro)
# update a new transport model to use velocity from z
solver.define(['c', 'd'], 'cd', radio_transport_hydro)
# setup boundary conditions
solver.setup_bcs()
# solve
solver.solve(time_update=True)
```

For the shallow water equations demonstration, a highly idealized version of the Strait of Gibraltar is set-up motivated by a case studied in [19]. A time varying Dirichlet boundary condition is imposed on the height of the currents on the inlet of the Strait (assumed to be the left sided boundary). To keep track of time varying Dirichlet boundaries, the argument `time_update=True` is applied. The implementation of the shallow water equation below is written with a no normal flow boundary condition on every other boundary except the inlet (left sided boundary):

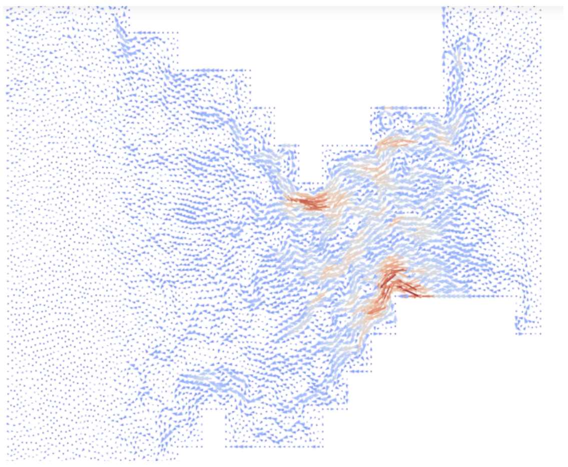
$$h = \alpha cos(\omega t), \quad \text{on} \quad \Gamma \tag{15}$$

$$u \cdot n = 0, \quad \text{on} \quad \Gamma_{other} \tag{16}$$

where h is the wave height component of the mixed function space, α and ω are constants, t is time and Γ and Γ_{other} are the inlet and every boundary except the inlet, respectively. After integration by parts, the continuity equation in the variational form becomes:

$$\int_\Omega \nabla \cdot (Hu) v dV = -\int_\Omega H(u \cdot \nabla v) dV + \int_\Gamma H(u \cdot n) v dS, \tag{17}$$

where v is the test function on the wave height function space, V is the volume of the integral, S is the surface of the integral, and n is the normal direction to the boundaries of the domain. The second term on the right evaluates to zero and thus drops out for every boundary except the inlet, due to the no normal flow condition imposed by equation 16. The consequence of maintaining this boundary condition everywhere except on the inlet is that the velocities and current heights become reflective within the domain [19, 11]. This can be seen in the velocity profile in figure 6a after the initial wave has travelled across the domain.


(a) Velocity of the waves travel reflect backwards from right to left.

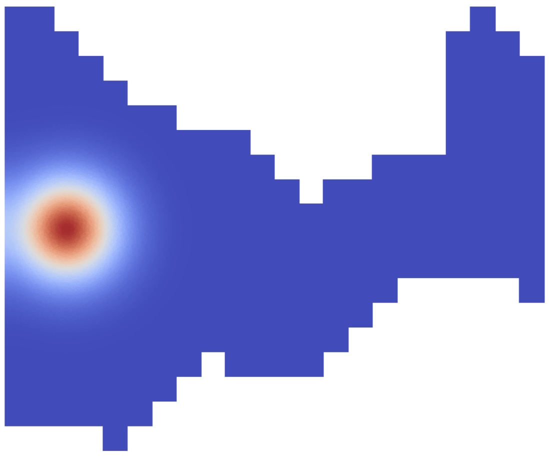

(b) Dissolved phase concentration of radionuclide one.

Figure 6: Shallow water transport model simulation on the Strait of Gibraltar at 2 seconds.

Alternatively, a Flather boundary condition for the outlet can be specified [11, 19]. A Flather boundary condition specifies an exit velocity and wave height outside the boundary domain and it allows the numerical results at the boundary to dissipate outwards:

$$u - u_* = \sqrt{\frac{g}{H}}(h - h_*) \quad \text{on} \quad \Gamma_{out} \tag{18}$$

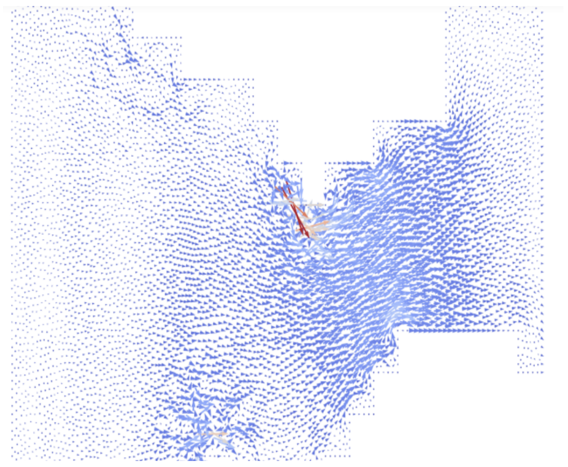
where u_* is the exit velocity outside the domain and h_* is the exit wave height outside the domain. The Flather boundary condition prevents the previously observed reflection of the waves. It is not specified using a `fd.DirichletBC(...)` object, but rather it is implemented directly into the variational form of the PDE.

From Fireframe, to switch to the Flather's implementation of the shallow wave equation requires selecting a different `PDESubsystem` called `depth_avg_hydro_flather` when defining the problem. Note: the user is still responsible for specifying a correct Flather boundary condition (exit velocity and wave height):

```
# update parameters and specify systems...
# use flather PDESubsystem
solver.define(['z'], 'z', depth_avg_hydro_flather)
# transport model remains unchanged
solver.define(['c', 'd'], 'cd', radio_transport_hydro)
# setup boundary conditions and solve...
```

Currently, the simulation runs only to 0.8 seconds because the exit velocity is specified as a fixed value. This causes numerical instability inside the domain as the exit velocity and wave height are not being updated with time.

Preliminary results indicate that the radionuclide transport kinetic model has been successfully coupled to the shallow water equation. However, as the simulation times for both implementations of the shallow water equations are comparatively short, the radionuclide concentration profile changes very little from its initial positions and retain the Gaussian shape of the source (seen in figures 6b and 7b).


(a) The velocity profile is continually directed towards the outlet on the right.

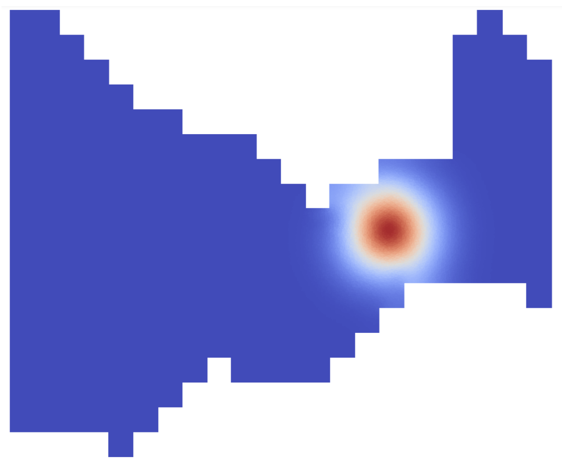

(b) Dissolved phase concentration of radionuclide one. New initial position.

Figure 7: Shallow water transport model simulation on the Strait of Gibraltar at 0.8 seconds using a Flather boundary condition.

5. Verification and Analysis

5.1. Performance analysis framework

When solving for complex systems of PDEs, analyzing the trade off between numerical accuracy and computational efficiency is important. The finite elements used and the variational forms posed will determine the best approach to solving a problem.

One example to consider is the groundwater transport of chemical species described in section 4.2. Two viable approaches to solving equations 6 to 8 are presented: either as a system of linear PDEs or as a singular nonlinear PDE with each species coupled together into a single `MixedFunctionSpace` (presented in section 4.2).

The two different approaches require significantly different setups through traditional Firedrake commands, but are trivial to setup through Fireframe. To create a linear system of PDEs for all three chemicals, all that is required is to specify a new `reactions_uncoupled(PDESubsystem)` with form functions describing the linear variational forms of each chemical species' mass balance, and to create another solver using this new `PDESubsystem`.

```
# to solve for the decoupled system, we can specify a new set
# of parameters as
{...
'space': {'u': fd.VectorFunctionSpace, 'c1': fd.FunctionSpace, 'c2' :fd.
    FunctionSpace, 'c3':fd.FunctionSpace},
...})
...
# and define the new solver object with the uncoupled variables
solver = pde_solver([['u', 'p']], mesh, solver_parameters)
solver.add_subsystem(['c1', 'c2', 'c3'], solver_parameters)
...
# define the new pdesystem
solver.define(['u', 'p', 'u'], 'up', navier_stokes)
solver.define(['c1', 'c2', 'c3'], 'c1c2c3', reactions_uncoupled)
```

```
...
solver.solve()
```

Using Python's `time` module, the computational speed between the solvers - one that uses `reactions` and the other uses `reactions_uncoupled` - can be directly compared.

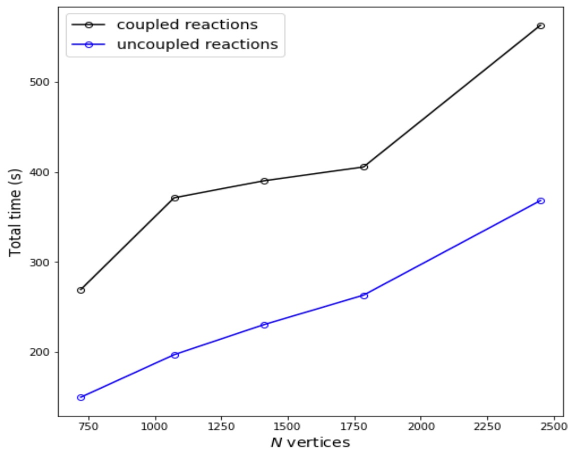
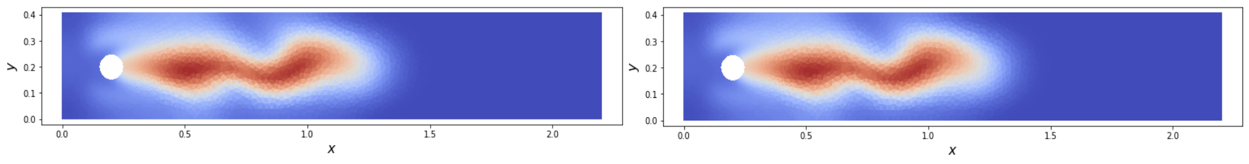


Figure 8: Calculation time(s) of coupled and uncoupled reaction equations.

For this reactions problem, solving the uncoupled systems of PDEs provides a noticeable speed improvement over solving the singular nonlinear equation and this relationship appears to be linearly related to mesh refinement. This can be attributed to the preconditioning and iterative methods used by Firedrake for solving linear systems. In contrast, Fireframe does not currently provide preconditioning techniques for nonlinear forms. Figure 9 indicates little to no numerical discrepancy between the solutions obtained via the two approaches.



(a) Final results obtained at 1(s) from coupled equation (b) Final results obtained at 1(s) from uncoupled equation

Figure 9: Numerical solution to chemical 3 concentration profile from equation 8

This example highlights the benefit of adopting Fireframe, which is its ability to efficiently couple different systems of PDEs together to streamline analysis and benchmarking of underlying numerical methods and physics.

5.2. Method of Manufactured Solution

Spatial and temporal convergence are important in demonstrating the validity of any finite element software, and verification of convergence rates can be efficiently performed using Fireframe. Software verification ensures that the time stepping schemes used in the variational forms and/or the finite elements used in the formulation are correctly implemented. When an analytical solution to the problem is unknown, the most appropriate verification technique is to use the method of manufactured solutions (MMS).

In the MMS procedure one chooses an "artificial" solution and substitutes this solution into a PDE/ system of PDEs. This leads to a residual which can be computed analytically and subsequently used as a source term in the MMS procedure. The chosen “artificial” solution is then an exact solution to the problem that includes this source term [20].

The “exact” solution is also used to impose a Dirichlet boundary condition at each time step. The numerical solution obtained by solving the PDE with the extra source term can be evaluated against the “exact” solution at the final simulation time to obtain a measure of the error. In Firedrake’s, there is a built-in function that calculates the l_2 norm between the numerical and exact solutions.

Setting up a MMS solution in Fireframe requires the user to create a `PDESubsystem` that includes the manufactured source term. The next step is to call the `PDESystem`’s `test_mms()` function.

5.3. Formulating the variational forms

5.3.1. Radionuclide transport MMS

Referring back to section 4.3, a MMS solution is created for the specific activity in the dissolved phase for the first radionuclide ($i = 1$) given in equation 12. This MMS solution assumes all species of radionuclides in the suspended and sediment phase (equations 13, 14) are initially zero and only dependent on the dissolved phase. On a two dimensional domain, Ω, with x and y coordinates and time variable t, the proposed MMS solution for the specific activity of the first species in the dissolved phase in equation 12 is:

$$C_{d,1exact} = \exp(xyt). \tag{19}$$

The source term calculated analytically from this solution and equation 12 can be written as:

$$\begin{aligned} f_{source} &= xyC_{d,1exact} + u \cdot (ytC_{d,1exact}, \quad xtC_{d,1exact}) \\ &\quad - K_d((yt)^2 C_{d,1exact} + (xt)^2 C_{d,1exact}) + k_1 C_{d,1exact} \\ &\quad + \lambda_1 C_{d,1exact}. \end{aligned} \tag{20}$$

The source term from equation 20 can then be inserted back into equation 12 to obtain the MMS variational form:

$$\begin{aligned} \int_\Omega \frac{(C^n_{d,1} - C^{n-1}_{d,1})}{\triangle t} v dV + \int_\Omega u \cdot \nabla C^n_{d,1} v dV = &- \int_\Omega K_d(\nabla C^n_{d,1} \cdot \nabla v) dV - \int_\Omega k_1 C^n_{d,1} v dV \\ &+ \int_\Omega k_2 C^n_{s,1} v dV + \int_\Omega k_2 A^n_{s,1} L \rho_s f \phi v dV - \int_\Omega \lambda_1 C^n_{d,1} v dV \\ &+ \int_\Omega f_{source} v dV, \end{aligned} \tag{21}$$

where $C^n_{d,1}$ is the current iteration of specific activity for the first radionuclide in the dissolved phase, $C^{n-1}_{d,1}$ is the previous iteration of specific activity for the first radionuclide in the dissolved phase, V is the volume of the integral and v is the test function on the function space for the specific activity in the dissolved phase. Solving equation 21 provides a numerical solution to the specific activity $C_{d,1}$ that can be tested against the exact MMS solution described in equation 19 for convergence. A first order, Continuous Galerkin `('CG')` finite element is used for this analysis. The numerical time stepping scheme used is an implicit, backwards Euler method [21].

5.4. Setting up MMS in Fireframe

Performing a convergence analysis by MMS in Fireframe requires the user to setup and initialize a new `PDESystem` object, identical to the process of solving a coupled problem. For the radionuclides transport problem, the following code specifies a spatial convergence test.

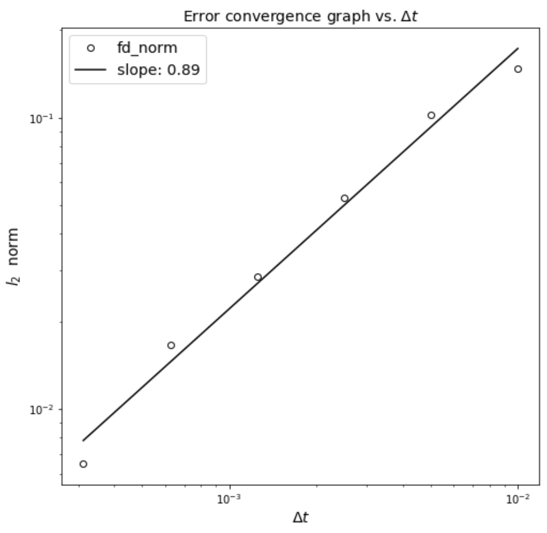
```
# create default parameters parameters
solver_parameters = recursive_update(solver_parameters,
{...})
# load mesh
mesh = fd.Mesh("..")
# add subsystems for navier stokes and radio_transport equations
solver = pde_solver([['u', 'p']], mesh, solver_parameters)
solver.add_subsystem(['c', 'd'], solver_parameters)
#setup constants
solver.setup_constants()
# define subsystems and variable sequence
solver.define(['u', 'p', 'u'], 'up', navier_stokes)
solver.define(['c', 'd'], 'cd', radio_transport_coupled_mms)
# setup boundary conditions
solver.setup_bcs()
# create symbolic exact solution
x, y, t = sy.symbols(('x', 'y', 't'))
expr = sy.exp(x*y*t)
meshes = [fd.Mesh("../../meshes/step%d.msh" % i) for i in range(1, 9)]
solver.test_mms('c', expr, spatial=True, f_dict={"exp":fd.exp}, meshes=
    meshes, plot=True, index=0)
```

Fireframe requires users to input a symbolic representation of the exact manufactured solution. Furthermore, the symbolic expression should also be written using symbols named x, y, and t when applicable. An argument `f_dict` allows Fireframe to translate symbolic mathematical expressions into Firedrake expressions. Currently, Fireframe supports both spatial convergence testing (`spatial=True`) and temporal convergence testing (`temporal=True`). Users are required to provide either a list of meshes or a list of Δt values when calling the `test_mms` function. An `index` argument is provided should the manufactured solution be a sub component of a `MixedFunctionSpace`, such as the case with the radionuclides transport problem.

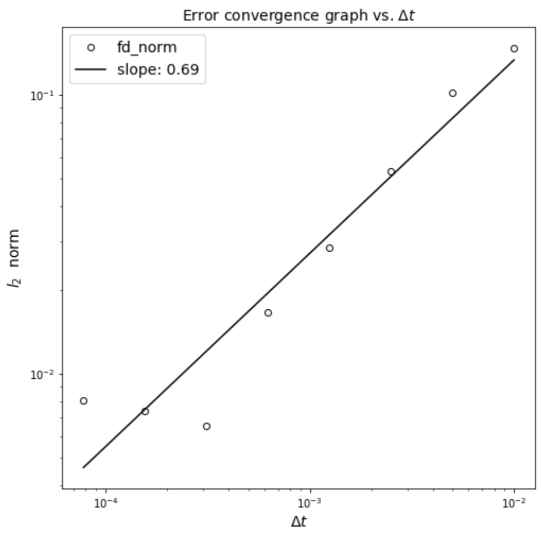
5.4.1. Temporal convergence

The time stepping scheme used in equation 21 is a backwards Euler method, which is expected to be first-order convergent in time [21]. The MMS solution to the radionuclide transport problem begins at a starting time of zero and completes at an end time of 0.1 seconds. The error obtained is the l_2 norm between the exact solution $C_{d,1exact}$ and the calculated solution from the solver.

The results presented in figure 10a indicate that the numerical scheme posed in the variational form for this radionuclides transport problem is indeed approximately first-order convergent in time. However, for very low Δt values, further decreases in the error ceases, as shown in figure 10b.



(a) Δt values from 0.01s to 3.125E-4s.

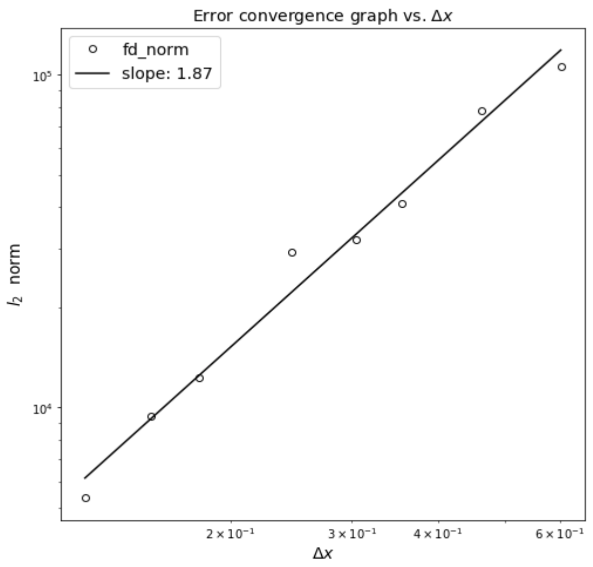


(b) Δt values from 0.01s to 7.81E-5s.

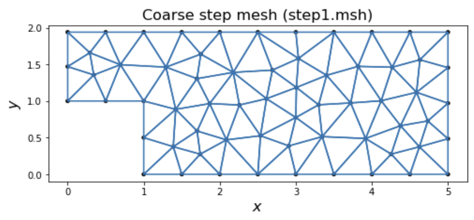
Figure 10: Temporal discretization errors.

5.4.2. Spatial discretization

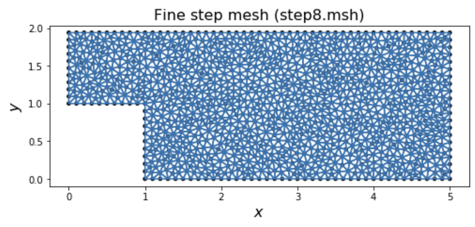
Spatial convergence of the finite element method depends on the family of finite elements used in setting up the function spaces. For this MMS analysis, first order Continuous Galerkin `('CG')` finite elements are used to approximate the concentration of the first species in the dissolved phase. The MMS solution to the radionuclide transport problem is fixed at a small Δt value of 0.001 seconds. The problem is solved until an end time of 2 seconds. A range of meshes from coarse to fine are used to obtain different Δx values.



(a) Spatial discretization errors.



(b) Initial mesh used (most coarse). Average Δx value of 0.426



(c) Final mesh used (most fine). Average Δx value of 0.087

Figure 11: Spatial discretization errors.

For a small Δt value, the spatial convergence of the radionuclide problem using first order CG finite elements approaches second order (as shown in figure 11). No reduction in convergence rate is exhibited, even for very fine mesh sizes.

5.4.3. Discussion on observed order

Truncation error of numerical methods that involve both temporal and spatial discretization can be described by the following notation:

$$error = \mathcal{O}(\Delta t^m + \Delta x^n). \tag{22}$$

The errors presented in the convergence graphs (figures 10a, 10b, and 11a) are calculated between the numerical solution and the manufactured solution and contains contributions from both the time stepping scheme as well as the finite elements used.

In the convergence analysis for the radionuclides transport problem, each parameter (Δt and Δx) was analyzed in isolation with the other being fixed. Error reduction from the refinement of one variable will always include a fixed contribution from the other. When the truncation error is dominated by one variable, the observed convergence of discretizing the other variable will not align closely with the theoretical order. Therefore, it is not expected that theoretical orders of accuracy will always be obtained exactly.

For temporal convergence, this phenomenon becomes evident when the time step Δt reduces to very small values as shown in figure 10a. There are two possible explanations for the observed asymptotic behavior of temporal convergence:

1. The time step is significantly small that floating-point arithmetic dominates. The solver may not be finishing at exactly the designated end time of 0.1 seconds for every Δt value which implies that the error is calculated inconsistently across data points.
2. The truncation error associated with time discretization becomes so insignificant that spatial discretization errors begin to dominate.

A combination of the two factors above explain why the backward euler scheme used in the radionuclide transport problem does not exhibit exactly first order convergence.

For spatial convergence, it is noteworthy that the mesh elements used (see figures 11b or 11c) are triangular. To calculate Δx values, a simplifying assumption was made by assuming all triangles are equilateral triangles with an averaged area taken among all elements. The Δx values are then obtained by calculating the average length of one side of this idealized equilateral triangle. This simplifying assumption for calculating Δx values will introduce inaccuracy in the spatial error convergence plot, which may explain why convergence is not exactly second order. Furthermore, errors calculated from spatial refinement will also have a fixed contribution from the errors associated with the time stepping scheme, so obtaining exactly second order convergence is not expected.

Nonetheless, the convergence analysis from the MMS solution to the radionuclides problem presented in this section demonstrates close adherence to theoretical orders of convergence for both time and space. This verifies that the time stepping scheme written in equation 21 as well as the finite elements used for the spatial discretisation in this problem have been implemented appropriately. The small deviations from theoretical orders of convergence can be explained by the way truncation errors are calculated as well as the simplifying approximations made to obtain Δx values.

6. Discussion and Conclusion

6.1. Challenges and Difficulties

Several key challenges presented themselves throughout the development of Fireframe. Creating a generalized framework that leverages the entirety of Firedrake's library of functions and adapting them to a variety of different types of problems was an ambitious undertaking. For the development of Fireframe, time constraints required that focus be applied to a single aspect, which was to create an interface to couple multiple PDEs into one problem.

Next, maintaining time changing Dirichlet boundary conditions and implementing them intuitively into a framework was challenging. The challenge here lies in the fact that not every form being solved will require a boundary condition, but the framework itself must be aware of which boundary conditions are changing. This became a difficult feature to program into `PDESystem` and the setting up of boundary conditions has not been made more streamlined in Fireframe.

Implementing new `PDESubsystems` also required a trial and error process as some formulations were inappropriate and unable to solve the problems. For example, attempting to split the coupled function space for the shallow water equation [11] led to numerical instability. Fireframe still does not address the need for the end users to be aware of how to correctly pose the problem.

6.2. Current limitations

The current Fireframe functionality has only been tested and verified on two dimensional problems. Advanced Firedrake functionality such as ExtrudedMesh and higher dimensional (i.e. 3D) problems have not been resolved using Fireframe.

The existing MMS framework for spatial convergence is also built on the foundation that problems are two dimensional. The calculation for Δx values inherently assume that users are choosing triangular finite elements. Furthermore, the Sympy interface for specifying a manufactured solution only allows for scalar variables. For example, the existing `text_mms()` function is unsuitable for specifying a manufactured solution for velocity, which is a vector variable.

6.3. Moving forward

One useful functionality that does not currently exist in Fireframe is the support for an offline coupled solver. An offline solver would allow one or more coupled variable to be read into the time loop from an existing output file. This is a useful feature because depending on the time stepping schemes used, stability constraints for one set of PDEs may not be an issue for another. An offline mode can also be computationally less expensive and more efficient when replicating simulations, for example where the same hydrodynamic fields can be used with multiple chemical reaction-transport problems. The current strategy of solving all `PDESubsystems` together in the same loop constrains all PDEs to the same time step which is an inefficient design.

Although a framework is built for manufactured testing, Fireframe only allows users to declare a `sympy` expression for the manufactured solution. Users are still required to implement `PDESubsystem` objects that includes the manufactured source term in the variational forms of PDEs. A more powerful feature would be to extend Fireframe to automatically product the entire source term from a `sympy` expression.

6.4. Closing remarks

A functional programmable Firedrake framework has been developed in Fireframe. Support for a variety of function spaces, coupled PDE problems, and time-dependent problems is included. Fireframe's automation of setting up function spaces and functions yields considerable reduction of programming effort required from users. Examples of different coupled PDE problems are presented and solved using Fireframe. Code verification by the method of manufactured solutions verifies that a system of nonlinear PDEs in the radionuclide transport problem coupled with the Navier-Stokes equation is converging in accordance with theoretical orders in time and space.

7. Acknowledgements

The author would like to thank the following people for their contributions to this project:

- Professor Matthew Piggot for guidance and instruction
- The Firedrake team, specifically Dr. David Ham and Dr. Koki Sagiyama for their support on using Firedrake. Mr. Florian Wechung for his support on setting up continuous integration
- Professor Mikael Mortensen for inspiration of a programmable FEniCS framework `cbd.pdesys`

References

[1] Garth N. Wells Mikael Mortensen, Hans Petter Langtangen. A fenics-based programming framework for modelling turbulent flow by the reynolds-averaged navier-stokes equations. *Advances in Water Resources*, 34(9).

[2] Lawrence Mitchell Michael Lange Fabio Luporini Andrew T. T. Mcrae Gheorghe-Teodor Bercea Graham R. Markall Florian Rathgeber, David A. Ham and Paul H. J. Kelly. Firedrake: automating the finite element method by composing abstractions. *ACM Trans. Math. Softw.*, 43:1–24:27, 2016. doi: 10.1145/2998441. URL `http://arxiv.org/abs/1501.01809`.

[3] Ufl: A finite element form language. *Automated Solution of Differential Equations by the Finite Element Method, Volume 84 of Lecture Notes in Computational Science and Engineering*, 84(17), 2012.

[4] K. B. Ølgaard M. E. Rognes M. S. Alnaes, A. Logg and G. N. Wells. Unified form language: A domain-specific language for weak formulations of partial differential equations. *ACM Transactions on Mathematical Software*, 40, 2014.

[5] Mikael Mortensen. Cbc.pdesys. URL `https://launchpad.net/cbcpdesys`. Accessed 06-2019.

[6] Walter Frei. Which turbulence model should i choose for my cfd application. URL `https://uk.comsol.com/blogs/which-turbulence-model-should-choose-cfd-application/`. Accessed 06-2019.

[7] B. E. Launder W.P. Jones. The prediction of laminarization with a two-equation model of turbulence. *Int. J Heat Mass Transfer*, 15:301–314, 1972.

[8] PETSc. Ksptype. URL `https://www.mcs.anl.gov/petsc/petsc-current/docs/manualpages/KSP/KSPType.html#KSPType`. Accessed 08-2019.

[9] FeniCS project. 12. incompressible navier-stokes equations. URL `https://fenicsproject.org/olddocs/dolfin/1.3.0/python/demo/documented/navier-stokes/python/documentation.html`. Accessed 08-2019.

[10] Firedrake. Defining variational problems, . URL `https://www.firedrakeproject.org/variational-problems.html#time-dependent-boundary-conditions`. Accessed 08-2019.

[11] C.T. Jacobs and Matthew Piggott. Firedrake-fluids v0.1: numerical modelling of shallow water flows using an automated solution framework. *Geoscientific Model Development*, 8:533–547, 03 2015. doi: 10.5194/gmd-8-533-2015.

[12] Daisy T. Cheng Alexander H.-D. Cheng. Heritage and early history of the boundary element method. *Engineering Analysis with Boundary Elements*, 29:268–302, 2005.

[13] Firedrake. Source code for firedrake.variational_solver, . URL `https://www.firedrakeproject.org/_modules/firedrake/variational_solver.html/`. Accessed 08-2019.

[14] Alexandre Chorin. A numerical method for solving incompressible viscous flow problems. *Journal of Computational Physics*, 135(135(CP975716):118 – 125, 1997.

[15] John D. Bredehoeft. Mass transport in flowing groundwater. *Water Resources Research*, 9(1):194–210, 1973.

[16] Jorge Ancheyta. *Chemical Reaction Kinetics*. Wiley, 2017.

[17] Hans Petter Langtangen and Anders Logg. Solving pdes in python. URL `https://fenicsproject.org/tutorial/`. Accessed 08-2019.

[18] Rikke Margrethe Closter Johannes Sandberg Anders Christian Erichsen, Flemming Møhlenberg. Models for transport and fate of carbon, nutrients and radionuclides in the aquatic ecosystem at Öregrundsgrepen. Technical Report R-10-10, 2010.

[19] Raul Perianez. *Modelling the Dispersion of Radionuclides in the Marine Environment.* Springer, 2005. ISBN 3-540-24875-7.

[20] Patrick Knupp Kambiz Salari. Code verification by the method of manufactured solutions, 2010.

[21] Lubin Vulkov(ed.) Ivan Dimov, Istvan Farago. Numerical analysis and its applications. *5th International Conference, NA 2010 Lozenetz, Bulgaria*, 2012.