ACADÉMIE
DES SCIENCES
INSTITUT DE FRANCE

Comptes Rendus
Mécanique

Stéphane de Chaisemartin, Sylvain Desroziers, Guillaume Enchéry, Raphaël Gayno, Jean-Marc Gratien, Gilles Grospellier, Thomas Guignon, Pascal Havé, Benoît Lelandais, Alexandre l'Héritier, Anthony Michel, Aboul Karim Mohamed El Maarouf, Valentin Postat, Xavier Tunc and Soleiman Yousef

ArcNum: an Arcane-based numerical framework used in porous media flow simulation applications

Volume 353 (2025), p. 1289-1314

Online since: 1 December 2025

https://doi.org/10.5802/crmeca.336

The Comptes Rendus. Mécanique *are a member of the Mersenne Center for open scientific publishing*
www.centre-mersenne.org — e-ISSN : 1873-7234

ACADÉMIE DES SCIENCES
INSTITUT DE FRANCE

Comptes Rendus. Mécanique
2025, Vol. 353, p. 1289-1314
https://doi.org/10.5802/crmeca.336

Research article

ArcNum: an Arcane-based numerical framework used in porous media flow simulation applications

Stéphane de Chaisemartin [*,a], **Sylvain Desroziers** [a,b], **Guillaume Enchéry** [a], **Raphaël Gayno** [a], **Jean-Marc Gratien** [a], **Gilles Grospellier** [c], **Thomas Guignon** [a], **Pascal Havé** [a,d], **Benoît Lelandais** [c], **Alexandre l'Héritier** [c], **Anthony Michel** [a], **Aboul Karim Mohamed El Maarouf** [a], **Valentin Postat** [c], **Xavier Tunc** [a] and **Soleiman Yousef** [a]

[a] IFPEN, 1 et 4 avenue de Bois-Préau, 92852 Rueil-Malmaison Cedex, France

[b] Michelin, 23 Place des Carmes Déchaux, 63040 Clermont Ferrand Cedex 9, France

[c] CEA, DAM, DIF, 91297 Arpajon, France

[d] NUANT, Baarerstrasse 20, 6300 Zug, Suisse

E-mail: stephane.de-chaisemartin@ifpen.fr

Abstract. This article presents ArcNum, a framework built on the Arcane platform, designed to easily and efficiently develop and maintain the numerical core in finite-volume and finite-element applications. This framework first enables the automatic generation of code needed to instantiate and handle complex physical models from a textual description, through its component called GUMP. ArcNum also offers software components to create, register and evaluate a set of physical laws, requiring only the specification of their inputs, outputs, and corresponding mathematical formulations. Finally, the framework includes a component named Contribution, which combines law evaluation with automatic differentiation to assemble linear systems efficiently. The framework ArcNum has been used to develop an open source Arcane-based porous media flow simulation proxy-app, named ShArc. Single and two-phase porous media flow simulations performed with ShArc are presented to complete the framework description. In order to illustrate the ability to use ArcNum for High Performance Computing, massively parallel simulations conducted with ShArc are finally presented.

Keywords. Mathematical software framework, code generation, automatic differentiation, Arcane, proxy-app, multiphase porous media flow simulation, massively parallel computing.

Note. Article submitted by invitation.

Manuscript received 9 January 2025, revised 13 October 2025, accepted 20 October 2025.

1. Introduction

The open source Arcane platform for scientific computing [1] is an advanced framework designed to facilitate the development and execution of complex scientific simulations applications. It provides researchers, scientists, and engineers with a comprehensive set of tools and libraries specifically tailored for high-performance computing (HPC) applications.

* Corresponding author

Arcane started being developed at the CEA-DAM, the Military Applications Division of the French Alternative Energies and Atomic Energy Commission, in the early 2000s, before being co-developed with IFPEN, French Institute for Research and Innovation in Energy, Mobility and Environment, from 2007.

During these twenty years of existence, Arcane has been the foundation of several industrial applications, from laser physics to CO_2 storage or geothermal energy simulations. Today eight industrial simulators are based on this platform, for nearly three million code lines from CEA-DAM and IFPEN.

Starting in the 2020s it has been decided to gradually move toward an open source strategy to foster the collaborative development between CEA-DAM and IFPEN, to make the platform available for the community and to foster potential new collaborations. Since the end of 2021, the platform is fully available on GitHub [2].

Arcane offers services dedicated to the application developers, handling for them all the low level computer science services needed by the application and giving them tools to dynamically build the application with an expandable and flexible option system.

Above the Arcane platform, dedicated to handle data and parallelism, numerical frameworks were developed to share the numerical tools (numerical schemes, (non)linear solvers...) between the applications. The works on linear solvers gave birth to Alien [3], a solver interface library designed to provide an application with a unique entry point to the world of linear solver libraries. On top of these (non)linear systems to solve, it was necessary to supply the applications with a way to simplify the complex writing of the Jacobian assembly arising for example in porous media flow simulations. ArcNum was developed with this objective.

In order to share the use of such Arcane-based numerical frameworks with the community, a numerical demonstrator, ShArc, has been released in open source on GitHub [4]. This Arcane-based proxy-application uses the frameworks Alien and ArcNum to solve porous-media flow problems. ShArc is now a proxy-app of Exa-DI NumPEx project; in this context an installation guide is provided on GitLab [5].

This paper is especially dedicated to the ArcNum framework and is organized in the following way. As an introduction, we first recall the main tools provided by the Arcane framework and highlight recent evolutions of the platform. We then present the ArcNum framework itself, detailing its three main components:

(i) a physical data model generator named GUMP;
(ii) a physical law framework;
(iii) a tool to assemble the Jacobian contributions, associated with automatic differentiation.

Finally, numerical results obtained with the ShArc demonstrator, combining all these tools, are presented for two porous media flow simulation test cases. In addition, a performance study of ShArc on massively parallel configurations is reported.

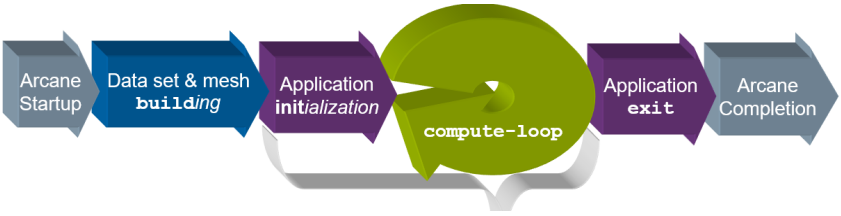


Figure 1. The different stages of the Arcane loop. The init, compute-loop and exit stages are the main stages used by the application developer to plug its *Modules* entry-points.

2. The Arcane Framework

Before diving into the ArcNum tools, we present in this part the Arcane Framework, an open source modular platform for HPC, on which it is based.

2.1. *An open source modular structure*

The Arcane Framework is made of several components, available on a single GitHub repository [6]:

Arccon: made of CMake functions and package finders;
Arccore: containing the base types and parallel tools of Arcane;
Axlstar: gathering the code-generation tools;
Alien: the linear algebra API (cf. Section 2.3);
Arcane: the Arcane platform itself (cf. Section 2.2).

With this modular structure, both Arcane and Alien use the common base provided by Arccore, the set of generators provided by Axlstar and the common CMake tools included in Arccon.

The Arcane framework committee decided in 2021 to release the platform with an open source Apache-2.0 license. All Arcane Framework components are now available on GitHub, see [2], through the *ArcaneFramework* organization, gathering the Arcane developers at CEA-DAM and IFPEN.

2.2. *Arcane: a complete platform for scientific computing*

The C++ platform Arcane provides all the building blocks needed to develop a parallel application. In particular, it provides a mesh structure with easy-to-use parallelization tools, distributed variables and parallel IOs, to name just a few of its features. It is based on an efficient component architecture allowing to define simulation problems as plugins or *Modules*, in the platform language, and to share functionalities through *Services*, defined by an interface, i.e. a class defining a programming contract. We refer to [1] for a more detailed description of the Arcane platform. We here only briefly describe the main functionalities and then present the platform recent developments.

The Arcane framework was originally designed to conduct simulations through the call of various entry-points, implemented by the developer in the *Modules*. Within the *Modules* entry-points, the application developer has access, amongst others, to the simulation mesh, natively distributed over subdomains, the simulation variables, or the simulated case options. The main program of the application only consists in calling the Arcane launcher. The application developer can then focus on developing *Modules*, and possibly *Services* to share treatment between *Modules*. The *Modules* entry-points are registered in a *time-loop*, mainly composed of three stages: the init, compute-loop and exit stages, see Figure 1. Both *Modules* and *Services* are made up of the following elements:

(i) a descriptor file, in xml format, used to defined the entry-points for a *Module* or the interface of a *Service*, and a list of options;
(ii) a C++ class, used to implement the *Module* entry-points and the *Service* interface.

An Arcane mesh is made of the classical geometric items such as cells, faces, nodes and edges. Non-geometric items, dofs (degrees of freedom) or particles, can also be added and connected to the geometric items. The platform can read various formats (vtk, msh and med formats) and generate a few mesh types too (see Section 2.4). It enables the use of several mesh partitioners (Metis, Parmetis, PTScotch and Zoltan) in a transparent manner. The Arcane mesh is intrinsically

unstructured and dynamic, which means that we can add new cells during the simulation and enrich the existing mesh. We can also create dynamically group of items to identify regions of the mesh. The mesh variables are then adapted automatically to take into account these changes in an efficient way. Let us add that a simulation may use several meshes if needed. Distributed variables can be defined on this/these mesh(es). A mesh variable is linked to a specific item family (a family of cells, faces, edges, nodes or dofs) and can have scalar or vector values of various types: Real (Arcane name for double precision floating points), Int16, Int32, Int64...

Arcane is designed to make parallel operations easier. By providing the concept of *SubDomain* with its *own* and *ghost* mesh items, it allows the application developer to easily implement an application based on domain decomposition. Ghost item values can be easily synchronized and explicit communications between the subdomains are not necessary. Arcane also offers load balance functionalities. The communications are classically carried out through mpi. Threaded and hybrid mpi+thread implementations are also available, though not used currently by the main Arcane-based applications.

An Arcane code example is given in Listing 1 to illustrate how mesh, variables and parallel synchronization operations are used to compute a variable defined on the mesh nodes from a second variable defined on the mesh cells. Note that, in this example, the group `mesh->allNodes()` only contains the nodes of the current subdomain. A final synchronization step is therefore required to ensure the correctness of the computation, as some ghost node values may remain inconsistent.

```
1  Arcane::IMesh* mesh; // given by Arcane
2  Arcane::VariableCellInt32 cell_var {VariableBuildInfo{
3                                          mesh,"MyCellVariable"}};
4  Arcane::VariableNodeInt32 node_var {VariableBuildInfo{
5                                          mesh,"MyNodeVariable"}};
6  cell_var.fill(1);
7  node_var.fill(0);
8  ENUMERATE_(Node, inode, mesh->allNodes()) {
9      for (Cell cell : inode->cells()) {
10         node_var[inode] += cell_var[cell];
11     }
12 }
13 node_var.synchronize();
```

Listing 1. Example of the computation of a node variable from a cell variable in parallel.

This abstraction of mesh enumeration and parallel communications allows applications developers to concentrate on the coding of physics and numerical operators. The Arcane level of abstraction remains low with direct access to mesh variables. This allows to easily adapt existing numerical algorithms, and offers the possibility to define finite element or finite volume methods on the top of this Arcane API.

Thanks to this subdomain-based structure, simulators at CEA and IFPEN have been used in parallel simulations on meshes ranging from thousands of millions up to dozens of billion of cells on up to hundreds of thousands of cores, see Section 5.

2.3. *Alien: a generic extensible linear algebra framework*

The Arcane framework now integrates a generic and extensible linear algebra framework called Alien [3]. This linear algebra API was initiated at IFPEN in the 2010s. It was designed to answer the problems arising when using different linear algebra packages in scientific software, and to

easily switch between different sparse matrix storage formats. Alien provides the application developer with a unique API to handle matrices, vectors and to solve linear systems. An example of system assembling and resolution is given in Listing 2. These Alien matrices and vectors are then converted at runtime into the representation of the chosen linear algebra package among PETSc, HYPRE, Trilinos, MTL or SuperLU. Alien also gives access to IFPEN internal solvers such as IFPSolver and MCG Solver [7]. It has been shown in [3] that these data structure conversions don't introduce a significant overhead in CPU-time.

```
auto pm = Env::parallelMng();
auto s = Space{10,"MySpace"};
auto md = MatrixDistribution{s,s,pm};
auto vd = VectorDistribution{s,pm};
auto A = Matrix{md}; // build a 10×10 square matrix
auto x = Vector{vd}; // build a 10 elements vector
auto b = Vector{vd}; // build a 10 elements vector
{
    // fill a tri-diagonal matrix
    auto builder = DirectMatrixBuilder(A,eResetValues);
    builder.reserve(30); // non-zero values
    for (Integer index = 0; index < 10; ++ index) {
        builder(index,index) = 2;
        if (index + 1 < 10) builder(index, index + 1) = -1;
        if (index - 1 >= 0) builder(index, index - 1) = -1;
    }
}
auto* solver = createSolver(/*...*/); // choose your favorite solver
auto status = solver->solve(A,x,b)
```

Listing 2. Handling distributed Alien matrices, vectors and solvers.

2.4. *Handling an always wider variety of mesh*

While initially designed only for standard unstructured meshes, the Arcane framework has been gradually enriched with:

(i) degenerate hexahedral cells,
(ii) generated Cartesian or honeycomb meshes,
(iii) Adaptive Mesh Refinement (AMR), both for unstructured meshes and Cartesian meshes,
(iv) polyhedral meshes,
(v) uniform mesh refinement, to generate a finer mesh.

All these mesh types or features are available in parallel. Degenerate heaxahedral cells, see Figure 2, have been added to handle complex soil geometries, see Figure 3. Such cells can be refined by the Arcane AMR procedure.

A mesh can be loaded by the platform under various formats (e.g. vtk, gmsh, med) or directly generated from a few input data. For instance, 2-D or 3-D Cartesian meshes as well as 2-D or 3-D honeycomb meshes can be created by just specifying the sizes of the domain and the number of cells.

For Cartesian meshes, Arcane provides a dedicated API giving access to neighbour elements in each direction. Cartesian meshes also come along with a dedicated block-refined AMR procedure.

Polyhedral meshes can currently be defined through the vtk format. This enables, for instance, the use of Voronoi meshes, see Figure 4. To handle such meshes, Arcane relies on the internal library Neo. This library is designed to handle complex polyhedral meshes that can possibly

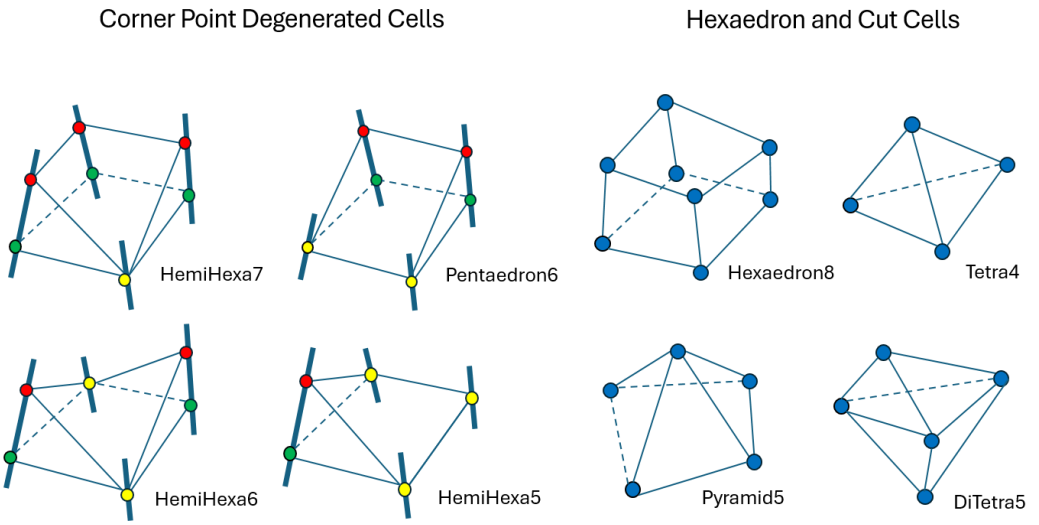


Figure 2. Degenerate hexahedral cells occurring in geological meshes.

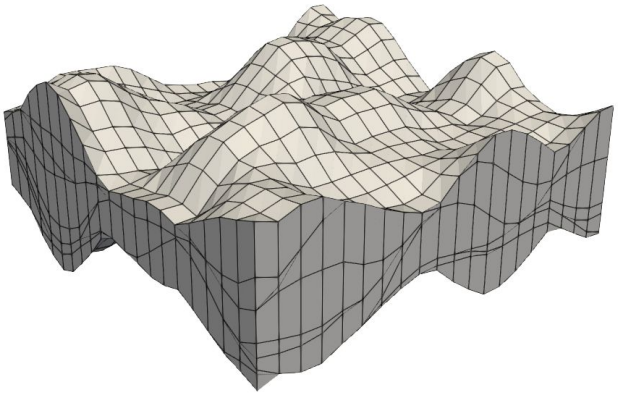


Figure 3. An example of geological mesh.

evolve during the simulation.[1] One of its interesting features is the way it asynchronously handles the mesh evolution by means of a dependency graph.

2.5. *A recent API for computing on accelerators*

Recently, a new API has been introduced in Arcane to handle accelerators with the following characteristics:

(i) it supports Cuda, HIP and SYCL backends;
(ii) it allows to use dynamically different types of GPU and multi-threaded CPU;
(iii) it ensures that the mesh connectivities and variables are available on the accelerator.

[1] Evolving meshes can, for instance, occur in geosciences when simulating the evolution of a sedimentary basin during geological times.

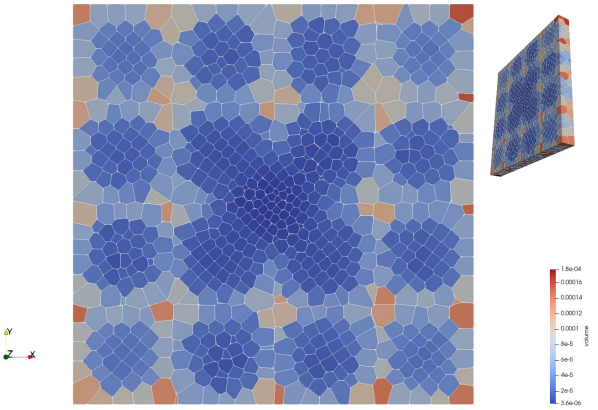


Figure 4. An example of Voronoi mesh colored by cell volumes.

```
// My mesh cell variables
Arcane::VariableCellReal m_a;
Arcane::VariableCellReal m_b;
Arcane::VariableCellReal m_c;

void compute_cpu_only()
{
 // given by Arcane
 Arcane::IMesh* mesh;









 // classical enumerate cell
 ENUMERATE_
    (Cell,icell,mesh->allCells()) {
  Real a = m_a(icell);
  Real b = m_b(icell);
  m_c(icell) = math::sqrt(a*a+b*b);
 };
}
```

Listing 3. Cell loop classical CPU version.

```
// My mesh cell variables
Arcane::VariableCellReal m_a;
Arcane::VariableCellReal m_b;
Arcane::VariableCellReal m_c;

void compute_cpu_or_gpu()
{
 // given by Arcane
 Arcane::IMesh* mesh;

 // define runner and queue
 // executing resource
 Arcane::Accelerator::Runner runner;
 // execution flow
 auto queue = makeQueue(runner);
 // computing kernel
 auto command = makeCommand(queue);

 // take for the device algorithm
 // 'a' and 'b' as inputs
 auto in_a = viewIn(command,m_a);
 auto in_b = viewIn(command,m_b);
 // 'c' as output
 auto out_c = viewOut(command,m_c);

 // Launch compute on device
 command << RUNCOMMAND_ENUMERATE
    (Cell,icell,mesh->allCells()) {
  Real a = in_a(icell);
  Real b = in_n(icell)
  out_c(icell) = math::sqrt(a*a+b*b);
 };
}
```

Listing 4. Cell loop accelerator version.

The API was inspired by the SYCL API from Khronos group [8]. As SYCL and Kokkos [9], it handles different architectures (NVIDIA, AMD, Intel). Its runtime-based approach allows for instance to address CPU (sequential or using multi-threading) or GPU or both with a same executable. The Arcane Accelerator API aims at providing the basic concepts from Arcane on GPU. The API allows to write classical mesh-based loops on GPU with few differences from the CPU version. In Listings 3 and 4 we compare a classical mesh compute loop on CPU and its implementation with accelerator API. The differences between both codes come only from a

dedicated setup needed on GPU, but the core of the loop itself is identical, see lines 28–31. In the GPU version, it is necessary to define a runner to specify the resource used for execution (line 13), a queue on this runner to manage the execution flow (line 15) and a computing kernel as a command (line 17). To manage variable access on specific accelerator devices, this API uses specific views of variables with access rules: In, Out or Inout, see lines 21–24.

This API manages Arcane variables on items (Cell, Node...), and gives access to the mesh connectivity. It also handles specific multi-dimensional array of dimension one to four. To submit command and process loop on target device we use specific loop macro based on C++ 11 lambda functions and specific loop macro, see Listing 5.

```
1  // pseudo-code describing the structure of the RUNCOMMAND_ API
2  command<<RUNCOMMAND_"LoopType"(args lambda consistent with "LoopType")
3   {
4     /* body lambda code to process on target device */
5   };
6
7  // two examples of the RUNCOMMAND_ API
8
9  // loop over mesh items (Cells, Nodes...).
10 // example for Nodes:
11 RUNCOMMAND_ENUMERATE(Node,inode,mesh()->allNodes()){};
12
13 // loop over multi-dimensional array of size one to four.
14 // example for one dimension:
15 RUNCOMMAND_LOOP1(iter, nb_value){};
```

Listing 5. Arcane API to execute parallel loops on a target device.

Finally, it also provides some high level functionalities like reductions and algorithms such as filtering, partitioning or sorting. It can also automatically profile the accelerator loops.

2.6. *From Arcane to a simulation application*

To build a numerical application upon the Arcane framework, the application developer will create *Modules* to implement the different physical models simulated. The entry-points of a *Module* structure the various computing steps of the model and are dynamically plugged into the application time-loop through a scheduler defined, by means of an xml `.config` file, by the application end-user.

Numerical and cross-functional tools are usually implemented in Arcane *Services*, so that they can be shared between *Modules* or even applications. These *Services* may handle numerical schemes, mesh reader/writer, etc. The library Alien, for instance, offers a collection of Arcane *Services* plugging different solver libraries, gathered under a unique interface.

Examples showing how to build an application upon Arcane are available in the platform repository [6] (tutorial or samples directories) or in mini applications arcane-benchs repository [10]. Two proxy-apps are also available in the Arcane Framework page on GitHub:

(i) the finite element proxy-app ArcaneFEM [11];
(ii) the porous media flow proxy-app ShArc [4], cf. Sections 4 and 5.

3. The ArcNum framework numerical tools

In this section, we present the ArcNum framework itself: a set of software components provided to facilitate the work of application developers. These components are located at the interface

between Arcane's kernel and the simulation application. These tools are for example plugged into the proxy-app ShArc [4], used in Sections 4 and 5 of this article.

The components of the framework ArcNum, namely *GUMP*, *Law Framework* and *AuDi*, are respectively dedicated to physical model handling, to the registration and evaluation of physical laws, and to automatic differentiation. They provide the needed tools to build the numerical core of an application in a highly readable and long-term maintainable way.

3.1. *GUMP data model*

For a given physical problem, GUMP (Generated Unified Manager Properties) allows to define the corresponding data model in an xml-style format file. This xml model has a very simple grammar defined in the `gump.xsd` scheme file. This grammar makes it possible to define a hierarchical structure of entities as shown in Listing 6 and, thus, to define all the physical properties carried by each entity of the model as in Listing 7. When the program is compiled, this file is parsed to generate C++ code that can be used by the application's developer. Even if most of the properties are defined in this xml model file, it is still possible to add additional properties within the application code.

```
<gump>
  <model namespace="ArcRes">
    <entity name="System">
      <contains>
        <entity name="FluidSubSystem" unique="true" />
      </contains>
    </entity>
    <entity name="FluidSubSystem">
      <contains>
        <entity name="FluidPhase" />
      </contains>
    </entity>
    <entity name="FluidPhase">
      <contains>
        <entity name="Species" />
      </contains>
    </entity>
    <entity name="Species">
      <contains>
        <entity name="Component" />
      </contains>
    </entity>
  </model>
</gump>
```

Listing 6. Example of an instance of a GUMP tree of entities.

```
 <entity name="FluidPhase">
  <supports>
    <property name="Viscosity" dim="scalar" type="real" />
    <property name="Density" dim="scalar" type="real" />
    <property name="CapillaryPressure" dim="scalar" type="real" />
  </supports>
</entity>
```

Listing 7. Example of GUMP properties associated to an entity.

This model is instantiated at runtime. In general, this action is carried out using a dedicated Arcane *Service*, see Section 2.2 and [1] for the description of Arcane *Services*. This *Service* allows the user to specify its model in the simulation input file, as shown in Listing 8.

```
<physical-model>
  <system name="UserSystem">
    <name>System</name>
    <fluid-system>
        <name>Fluid</name>
        <fluid-phase>
            <name>Water</name>
            <species>H2O</species>
        </fluid-phase>
        <fluid-phase>
          <name>Gas</name>
          <species>CO2</species>
        </fluid-phase>
    </fluid-system>
  </system>
</physical-model>
```

Listing 8. Example of an input simulation file instantiating a GUMP model.

This approach aims at bringing flexibility to flow models, for instance. Each property of the system is referred by a specific name called "xpath". It consists in specifying the property name with the path to the entity carrying it as a prefix. As an example, the input data of an initial condition *Service* is printed in Listing 9. The pressure property is uniquely designed through the path `[System]System::Pressure` and, here, its value is set to `1.38E+7`Pa.

```
<initial-condition name="Constant">
  <property>[System]System::Pressure</property>
  <condition>
    <value>1.38E+7</value>
  </condition>
</initial-condition>
```

Listing 9. Example of an input simulation file where xpath properties are used for initial conditions.

The generated C++ API also allows the application's developer to iterate over the model entities and manipulate the associated properties. An example of user code, parsing the physical model defined in Listing 9, is given in Listing 10. This code snippet prints the different entities of the physical model and shows how to access to the system pressure property by using the xpath we mentioned earlier.

```
ENUMERATE_SUBSYSTEM(isubsystem, system.subSystems()) {
    info() << *isubsystem; // prints: Fluid
    ENUMERATE_PHASE(iphase, isubsystem->phases()) {
      info() << *iphase;  // prints: Water Gas
      ENUMERATE_SPECIES(ispecies, iphase->species()) {
        info() << *ispecies; // prints: H20 CO2
      }
    }
}
// Access System Pressure
auto system_pressure = ArcRes::XPath::property(system,"System::Pressure");
```

Listing 10. C++ API to parse the physical system and access to the properties.

By defining a path to access to the different physical entities, GUMP provides a way to define the input and output properties of the law framework, described in the following section.

3.2. *Law framework*

When solving systems of partial differential equations, we frequently have to evaluate physical laws in groups of mesh items (cells, nodes...). For instance, in the context of porous-media flow simulations, these laws may correspond to petrophysical empirical relationships such as the relative permeabilities or the capillary pressures. The proposed law framework allows users to easily define new laws for their application.

In our framework, a law is simply a function which calculates output properties from input ones defined on groups of mesh items. It also evaluates the derivatives of the output properties with respect to the input ones.

In the law API, properties can be scalar, multi-scalar or vector-based. The access to a scalar law is simply done using its property name. It is for instance the case of the porosity. A multi-scalar law can be retrieved using several indexes of properties. Thus, the access to the molar fraction of a component is done using the name of the component and the name of the fluid phase.

The definition of a new law is carried out in three steps. First, the user defines the input and output properties of the law in an xml-style file (see Listing 11). This xml file is parsed at compilation time to generate a C++ header in the build directory. This header contains two classes, the `lawName::signature` defined in the xml file and the `lawName::law` class managing the evaluations of the law on different supports: mesh, mesh groups, item vectors or scalars. Second, the user has to implement the function which will be used to evaluate the law on a single mesh item (see Listing 12).

```
<law name="FluidDensityLawType">
  <input name="temperature" type="real" dim="scalar" />
  <output name="fluidDensity" type="real" dim="scalar" />
</law>
```

Listing 11. Example of xml file defining the law signature.

```
void FluidDensityLaw::eval(const Real T, Real& rho, Real& drho_dt)const
{
    rho = T + exp(T);
    drho_dt = 1 + exp(T);
}
```

Listing 12. User evaluation method corresponding to the xml definition.

At last, to be able to use a new law in an input `.arc` file, an arcane *Service* has to be created. This *Service* allows the user to choose the input and output properties at runtime. Listing 13 shows an example of `.arc` input file and Listing 14 is the implementation of the corresponding service. In this implementation, let us note that the two classes, generated in the header file, are instantiated: the `lawName::law` object taking in arguments the law evaluation function `FluidDensityLaw::eval` and the `law::signature` instance. At this stage the law is configured and added to the `function_mng` object (see last line of Listing 14). This last `Law::FunctionManager` object is used as a "law-database". When the application is initialized, each *Service*, which instantiates a law, adds it to the `Law::FunctionManager` object. The API also offers a `Law::FunctionEvaluator` object which allows one to select a subset of the available laws. This object is used to evaluate the selected laws on a specific support, at each time step,

for instance. Let us note that it is only at this stage that the evaluation support is specified. An example of law evaluation is given in Listing 15.

```
   <law name="FluidDensityLawConfig">
      <output>
        <fluid-density>[Phase]Water::Density</fluid_density>
      </output>
      <input>
        <temperature>[System]System::Pressure</temperature>
      </input>
      <parameters>
        <initial-temperature>0</initial-temperature>
      </parameters>
    </law>
```

Listing 13. Example of input simulation file configuring the law input and output.

```
#include "FluidDensityLawType_law.h" // generated from xml file
#include "FluidDensityLaw.h" // user definition of the law (cf. Listing 12)
void
FluidDensityLawConfigService::
configure(Law::FunctionManager& function_mng, ArcRes::System& system)
{
    // Initialize arguments from .arc input file
    auto fluid_density = options()->output().fluidDensity();
    auto temperature = options()->input().temperature();

    // Initialize parameters from .arc input file
    auto initial_temperature = options()->parameters().initialTemperature();

    // Set law arguments
    FluidDensityLawType::Signature signature;
    signature.fluidDensity = ArcRes::XPath::property(system, fluid_density);
    signature.temperature  = ArcRes::XPath::property(system, temperature);

    // Set law parameters
    FluidDensityLaw law;
    law.setParameters(initial_temperature);

    // Create law
    auto function = std::make_shared<FluidDensityLawType::Function>(
        signature, law, &FluidDensityLaw::eval);

    // Add law to function manager
    function_mng << function;
}
```

Listing 14. Example of C++ code to create a law and register it in the function manager.

```
// Prepare law evaluation
Law::FunctionManager function_mng; // Build and filled by the application
Law::FunctionEvaluator function_evaluator(function_mng);

IVariableMng variable_mng; // Given by the application
auto variable_accessor = variable_mng.variableAccessor<Cell>();

// Evaluate laws registered in function_mng
function_evaluator.evaluate(accessor, allCells(), Law::eWithDerivative);

// Access to law outputs
auto system_pressure = ArcRes::XPath::property(system,"System::Pressure");
auto computed_pressure = accessor.values(system_pressure);
auto computed_pressure_derivatives = accessor.derivatives(system_pressure);
```

Listing 15. Example of C++ code to evaluate the registered laws on a given support.

To conclude, the workflow to define and evaluate a physical law consists of four steps:

(i) definition of the law inputs and outputs in an xml-style file (Listing 11);
(ii) computation of its values and derivatives through a user-defined function (Listing 12);
(iii) definition of a configuration *Service* to instantiate and register the law object (Listing 14);
(iv) call to the law evaluator on a given support (Listing 15).

At runtime, users only have to complete the `.arc` file to set the actual inputs, outputs and parameters of the law, see Listing 13. After evaluating the laws, their values and derivatives can be read for any item included in the support defined in the evaluator. This functionality is a key point of the API since it provides all necessary data to build the matrices for the linear system assembly. These data are then gathered in *Contribution* objects which are described in the sequel.

3.3. *Contributions*

The resolution of nonlinear problems such as $\mathbf{F}(\mathbf{u}) = 0$ is often carried out thanks to a Newton's method. Each iteration k requires the construction of a linearized problem involving the Jacobian matrix of $\mathbf{F}$ and the residual vector $\mathbf{F}(\mathbf{u}^k)$. The *Contribution* API provides a convenient way to assemble the Jacobian and the residual terms, to compute automatically their derivatives to be stored in the matrix, and to add all these values in Alien systems.

3.3.1. *Contributions — Accessors*

For any GUMP property, the *Contribution* framework allows to recover its value and derivatives with respect to the main unknowns of the problem. When computing fluxes on a face, the derivatives often depend on various unknowns located around this face. This set of local elements is called a stencil. In the case of a diffusion flux discretized with the finite-volume two-point flux, this stencil includes the cells that share this face. To make the assembly of such fluxes easier in our framework, *Contributions* can be accessed according to the stencil of a given face. Let us give an example. In the Listing 16, lines 7–12 illustrate how to get *Contributions* related to three GUMP properties which are `ArcRes::Pressure`, `ArcRes::CapillaryPressure` and `ArcRes::FluidDensity`. These *Contributions* depend on an unknown manager that specifies the variables defined in each cell. This manager here appears twice since it is used with a two-point stencil. This stencil is initialized for each face on line 17. The following lines 18–25 show how to get the values for one stencil cell (back or front to the face).

```
1  _buildFluxInternal(ArcNum::Vector& residual, ArcNum::Matrix& jacobian)
2  {
3      // unknown manager of the system (List of unknown properties)
4      const auto& um = unknownsManager();
5
6      // contribution wrapper by property
7      auto P = Law::contribution<ArcRes::Pressure>(domain(),
8               functionMng(),um,um,system());
9      auto Pc = Law::contribution<ArcRes::CapillaryPressure>(domain(),
10              functionMng(),um,um,system().fluidSubSystem().phases());
11     auto Rho = Law::contribution<ArcRes::FluidDensity>(domain(),
12              functionMng(),um,um,system().fluidSubSystem().phases());
13
14     Arcane::FaceGroup inner_faces = mesh()->allCells().innerFaceGroup();
15     // Arcane loop on faces
16     ENUMERATE_FACE(iface,inner_faces) {
17         ArcNum::TwoPointsStencil stencil(iface);
18         const Law::Cell& cell_k = stencil.back();
19         const Law::Cell& cell_l = stencil.front();
```

```
        // GUMP loop on fluid phases
        ENUMERATE_PHASE(iphase, fluid.phases()) {
            // Acces contribution by mesh item [cell_k] or [cell_l]
            const auto Pk = P[cell_k];
            const auto Pl = P[cell_l];
            const auto Pck = Pc[iphase][cell_k];
            // idem for Pcl, Rhok and Rhol
            ...
        }
    }
}
```

Listing 16. Example of C++ code to access to the *Contributions*.

3.3.2. *Contributions — Combination and automatic differentiation with AuDi*

New *Contributions* can be created as a result of operations, for operators such as `*`, `/`, `+`, `-`, between several existing *Contributions*. An illustration is given in Listing 17. Here, the *Contribution* `grad_kl` is computed from the *Contributions* defined in the Listing 16. These operations make use of an automatic differentiation library, implemented in our framework and called AuDi.

```
_buildFluxInternal(ArcNum::Vector& residual, ArcNum::Matrix& jacobian)
{
    ...
        ENUMERATE_PHASE(iphase, fluid.phases()) {
            ...
            const auto grad_kl = ((Rhok + Rhol) / 2) *
            (Pk + Pck - Pl - Pcl);
            ...
            const auto flux_kl = ...// computed from grad_kl
        }
    }
}
```

Listing 17. Example of operations on *Contributions* computing values and derivatives.

AuDi is a C++ library based on generic programming techniques. It uses expression templates [12,13] and operator overloading. Operator overloading is a classical C++ technique that enables to define and implement operators on any object. It is applied in our case to implement the operators `+,-,/,*` on contributions. On the other hand, expression templates prevent the runtime from creating unnecessary variables or temporary objects and thus offer good CPU and memory performance in critical calculation sections such as flux evaluations.

3.3.3. *Contributions — Alien's linear system assembly*

Finally, adding *Contributions* to the Jacobian matrix or to the residual, defined by means of Alien's matrix and vector (see Section 2.3), can simply be done by using the operators `+=` or `-=`. Listing 18 shows an example where a finite-volume flux contribution `flux_lk` is inserted into the lines related to the equation `iequation` of both stencil cells. Let us notice that the insertion of the derivatives into the corresponding matrix columns is here done in a very compact way thanks to the use of the `stencil` object.

```
_buildFluxInternal(ArcNum::Vector& residual, ArcNum::Matrix& jacobian)
{
    ...
        ENUMERATE_PHASE(iphase, fluid.phases()) {
```

```
            ...
            const Arcane::Integer iequation = iphase.index();
            if (cell_k.isOwn()) {
                residual[iequation][cell_k]          += flux_kl;
                jacobian[iequation][cell_k][stencil] += flux_kl;
            }
            if (cell_l.isOwn()) {
                residual[iequation][cell_l]          -= flux_kl;
                jacobian[iequation][cell_l][stencil] -= flux_kl;
            }
        }
    }
}
```

Listing 18. Example of C++ code to assemble a Jacobian matrix using the *Contributions*.

4. First examples with ShArc, an Arcane-based demonstrator

In this section, we present numerical illustrations derived from two porous-media models implemented in ShArc: a single-phase thermohaline convection model and an isothermal two-phase flow model. We first introduce the common notations and subsequently provide detailed descriptions of both models.

4.1. *Notations*

The physical quantities used in both examples are introduced in this section. In the sequel, we also indicate their units where M stands for mass, L for length, T for time and Θ for temperature. The numerical values are given in the SI system. Vectors in $\mathbb{R}^3$ are denoted with bold letters and tensors in $\mathbb{R}^3 \times \mathbb{R}^3$ with blackboard-bold style letters.

We denote by $\phi[-]$ the porosity of the porous medium Ω, $\mathbb{K}[L^2]$ its permeability tensor. For a fluid phase p, $P_p[ML^{-1}T^{-2}]$ denotes its pressure, $\mu_p[ML^{-1}T^{-1}]$ the viscosity and $\rho_p[ML^{-3}]$ the mass density. We denote by $\boldsymbol{v}_p$ Darcy's velocity which is defined by

$$\boldsymbol{v}_p(P_p) = -\frac{\mathbb{K}}{\mu_p}(\nabla P_p - \rho_p \boldsymbol{g}) \tag{1}$$

where $\boldsymbol{g} = -g\nabla z[LT^{-2}]$ denotes the gravity vector.

4.2. *Thermohaline convection*

For the first model, we consider the transport of salt by water through the porous medium Ω. This problem, known as thermohaline convection [14], can be modelled by the following partial differential equations:

- the water mass balance,
$$\phi\partial_t\rho_\mathrm{w} + \mathrm{div}(\rho_\mathrm{w}\boldsymbol{v}_\mathrm{w}) = 0; \tag{2}$$
- the salt mass balance,
$$\phi\partial_t C + \mathrm{div}(\boldsymbol{F}_C) = 0, \tag{3}$$
where $\boldsymbol{F}_C = C\boldsymbol{v}_\mathrm{w} - \phi\mathbb{D}\nabla C$, $\mathbb{D}[L^2T^{-1}]$ is the porous-medium molecular diffusivity, $C[ML^{-3}]$ the mass concentration;

- the heat balance,

$$\partial_t\big((\phi c_{\mathrm{w}}\rho_{\mathrm{w}} + (1-\phi)\rho_{\mathrm{s}}c_{\mathrm{s}})\theta\big) + \mathrm{div}(\boldsymbol{F}_\theta) = 0 \tag{4}$$

where $\theta[\Theta]$ is the temperature,

$$\boldsymbol{F}_\theta = \rho_{\mathrm{w}}\theta\,\boldsymbol{v}_{\mathrm{w}} - \big((1-\phi)\mathbb{L}_{\mathrm{s}} + \phi\mathbb{L}_{\mathrm{w}}\big)\nabla\theta \tag{5}$$

and, for $p \in \{\mathrm{w},\mathrm{s}\}$ (the index s standing for salt), c_p is the heat capacity $[L^2T^{-2}\Theta^{-1}]$ and $\mathbb{L}_{\mathrm{p}}[MLT^{-3}\Theta^{-1}]$ the heat conductivity.

The boundary of Ω is partitioned into $\partial\Omega = \Gamma_{\mathrm{top}} \cup \Gamma_{\mathrm{bottom}} \cup \Gamma_{\mathrm{vertical}}$ where Γ_{top} is the highest part of the border, Γ_{bottom} the lowest one and $\Gamma_{\mathrm{vertical}}$ the remaining one. We denote by $\boldsymbol{n}$ the unit normal vector outwards to Ω. On the boundaries, we fix the following conditions:

$$\boldsymbol{v}_w \cdot \boldsymbol{n} = 0 \qquad \text{on } \partial\Omega, \tag{6}$$

$$\chi = \chi_{\mathrm{p}} \qquad \text{on } \Gamma_p,\ \chi \in \{C,\theta\},\ p \in \{\mathrm{top},\mathrm{bottom}\}, \tag{7}$$

$$\boldsymbol{F}_\chi \cdot \boldsymbol{n} = 0 \qquad \text{on } \Gamma_{\mathrm{vertical}},\ \chi \in \{C,\theta\}. \tag{8}$$

Initial conditions are also defined by zero for C and θ.

4.2.1. *Numerical setting*

We consider a thermohaline convection problem inspired from a test case proposed in [14]. Here the 2D space domain $\Omega = (0,300)\times(0,300)$ represents a homogeneous and isotropic aquifer where

- $\theta_{\mathrm{bottom}} = 200$, $\theta_{\mathrm{top}} = 0$, $C_{\mathrm{bottom}} = 10$, $C_{\mathrm{top}} = 0$;
- $\phi = 0.1$, $\mathbb{K} = 1e-13$, $\mathbb{D} = 1e-13$;
- $\rho_{\mathrm{s}} = 2700$, $c_{\mathrm{s}} = 1180$, $\mathbb{L}_{\mathrm{s}} = 2$;
- $c_{\mathrm{w}} = 4200$, $\lambda_{\mathrm{w}} = 0.65$;
- ρ_{w} is given by

$$\rho_{\mathrm{w}} = \rho_{\mathrm{w},0}\big(1 - \beta_\theta(\theta-\theta_0) + \beta_C(C-C_0)\big),$$

with $\rho_{\mathrm{w},0} = 1000$, $\beta_\theta = 3.37e-4$, $\beta_c = 6.38e-4$, $\theta_0 = 0$, $C_0 = 0$, $\mu_{\mathrm{w}} = 1e-3$.

A cell-centered finite volume discretization [15] is applied on the system (1)–(8), with an implicit Euler time stepping, a two-point flux approximation (TPFA) for the diffusive terms and upwinding for the convective ones. We perform the simulation on a grid discretized with 100×100 uniform cells. The initial time step is $\tau^0 = 8.64\times10^5\,\mathrm{s}$. At each time step we apply Newton's method [16] to solve the corresponding non-linear system. The linear system obtained at each Newton iteration is solved using an LU solver. If a divergence of the Newton algorithm occurs we divide the time step by 2. Otherwise, the time step is increased by multiplying it by 1.5. Figure 5 depicts the evolution of temperature and salt concentration at different time steps.

4.2.2. *Implementation within ShArc*

This section illustrates how the numerical tools presented in Section 3 are used to implement in ShArc the resolution of this thermohaline convection system.

Using GUMP (see Section 3.1), temperature and salt concentration are declared into the physical model by means of an xml file. An example of a section from this file is shown in Listing 19.

```
1    <property name="Temperature" dim="scalar" type="real" />
2    <property name="Concentration" dim="scalar" type="real" />
```

Listing 19. Definition of salt concentration and temperature GUMP properties.

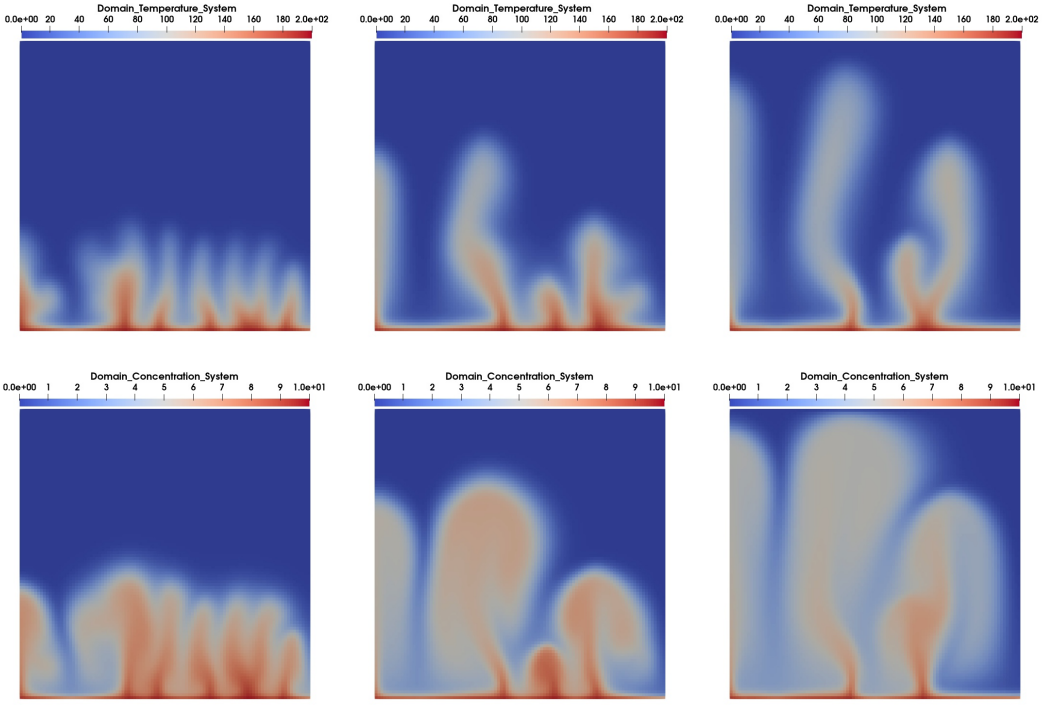


Figure 5. Evolution of the temperature (top) and salt concentration (bottom) after 28 years (left), 34 years (middle), and 40 years (right).

The law framework API, described in Section 3.2, allows to create the different laws related to the flow problem which is here considered. In our example, the fluid density ρ_w is defined by the code given in Listing 20. This piece of code will be evaluated in each mesh cell.

```
void FluidDensityLaw::eval(const Real T, const Real C, Real& rho,
                           Real& drho_dt, Real& drho_dc ) const
{
    Real factor = (1.0 - m_betat*(T - m_t0) + m_betac*(C - m_c0));
    rho = m_rho0 * factor;
    drho_dt = -1.0 * m_rho0 * m_betat;
    drho_dc = m_rho0 * m_betac;
}
```

Listing 20. C++ code computing fluid density law.

The parameters of the law ρ_w are then specified within the input file like in Listing 21. Note that the previous generated properties here appear as inputs of the law.

```
    <law name="FluidDensityLawConfig">
      <output>
        <fluid-density>[Phase]Water::FluidDensity</fluid-density>
      </output>
      <input>
        <concentration>[System]System::Concentration</concentration>
        <temperature>[System]System::Temperature</temperature>
      </input>
```

```
      <parameters>
        <rho0>1000</rho0>
        <betat>3.37e-4</betat>
        <betac>6.38e-4</betac>
        <t0>0</t0>
        <c0>0</c0>
      </parameters>
    </law>
```

Listing 21. Input simulation file defining fluid density law inputs, output and parameters.

When assembling the Jacobian matrix, contributions (see Section 3.3) provide the local values and derivatives of the variables and laws according to the primary unknowns of the system (here temperature and concentration) within the scheme stencil. Thus for a face σ between two cells K and L, the contributions contain the derivatives with respect to the unknowns of these two cells. Examples on how to get such contributions are given in Listing 22.

```
auto T = Law::contribution<ArcRes::Temperature>(domain(), ...,system());

auto phi = Law::values<ArcRes::VolumeFraction>(domain(),fluid);
auto mu = Law::contribution<ArcRes::Viscosity>(domain(), ...,fluid.phases());

auto rho = Law::contribution<ArcRes::Density>(domain(), ...,fluid.phases());
auto rhof = Law::contribution<FluidDensity>(domain(), ...,fluid.phases());

auto lambda_f = Law::contribution<HeatConductivity>( ..., fluid.phases());
auto lambda_s = Law::contribution<HeatConductivity>( ..., solid.phases());
```

Listing 22. C++ code retrieving the contributions for the evaluation of the thermal flux.

Finally, Listing 23 shows how to compute the discrete version of the thermal flux (5) and add it to the residual vector and to the Jacobian matrix.

```
const auto darcy_kl = (permeability / mu[fluid.phase(0)][cell_k])
        * tau[iface] * (P[cell_k] - P[cell_l] + rho_kl * dgz_kl);

const auto cond_s = ( (1.0-phi[K]) * L_s[solid.phase(0)][cell_k] +
                           phi[K]  * L_f[fluid.phase(0)][cell_k] );

const auto flux_t_kl =  darcy_kl
        * (one(darcy_kl >= 0)*T[cell_k]*rho[fluid.phase(0)][cell_k]
         + one(darcy_kl <  0)*T[cell_l]*rho[fluid.phase(0)][cell_l])
         + cond_s * tau[iface] * (T[cell_k] - T[cell_l]);

residual[iequation][cell_k]          += flux_t_kl;
jacobian[iequation][cell_k][stencil] += flux_t_kl;
```

Listing 23. C++ code building the upwind TPFA heat flux (5) using the contributions.

4.3. *Two-phase flow model*

We now consider an isothermal immiscible two-phase model of water and gas [17]. Each phase has only one component: H_2O and CO_2. Phase changes of these components are not here taken into account. Consequently, we use the indices "w" and "g" to refer indifferently to the corresponding phase or component. The difference between both phase pressures is equal to the capillary pressure:

$$P_g - P_w = P_{c_{g,w}}, \tag{9}$$

where $P_{c_{g,w}}[ML^{-1}T^{-2}]$ is a function of the gas saturation. Consequently, the model unknowns are the water pressure P_w and the saturations S_p, $p \in \{g, w\}$. Writing the conservation of the volume of each phase leads to the following system:

$$\phi \partial_t(\rho_w S_w) + \mathrm{div}(\rho_w k_{r,w} \boldsymbol{v}_w) = q_w, \tag{10}$$

$$\phi \partial_t(\rho_g S_g) + \mathrm{div}(\rho_g k_{r,g} \boldsymbol{v}_g) = q_g, \tag{11}$$

$$S_w + S_g = 1, \tag{12}$$

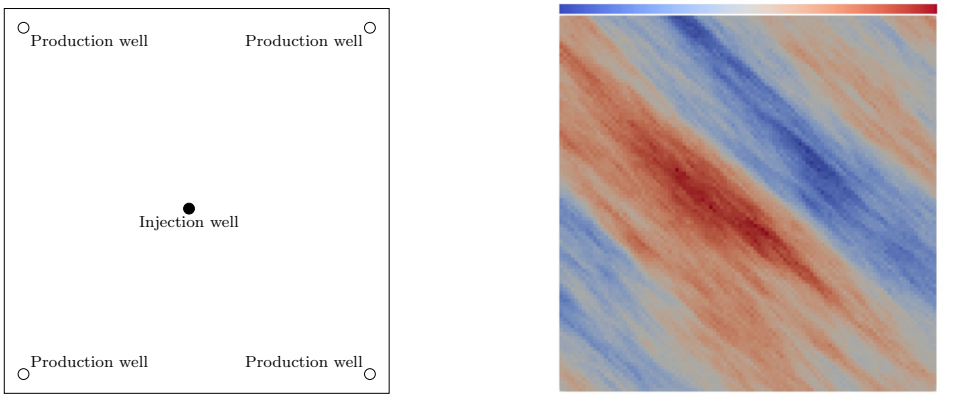
where $k_{r,p}(S_p)[-]$ stands for the relative permeability and $q_p[ML^{-3}T^{-1}]$ are sink or source terms which are non-zero where well perforations are present. No-flow boundary conditions are used for both phases. Initial conditions are given for S_g.

For further details about the two-phase flow model and the more general multiphase compositional flows in porous media we refer to [18–21]. More numerical experiments on practical problems illustrated on other Arcane-based IFPEN simulators with advanced resolution methods (adaptive mesh refinement, adaptive solvers, multilevel refinement, mixed multiscale finite elements) on general meshes can be found in [22–29].

4.3.1. *Numerical setting*

We consider the injection of CO_2 into a porous rock initially saturated with water. The spatial domain $\Omega = (0, 1010) \times (0, 1010)$ is discretized with a grid composed of 101×101 uniform cells. The process is organized following a five-spot pattern, see Figure 6(a), where one injection well is located in the middle of the domain and producers are in the four corners (these producers are here only used to create gradients of pressure through the domain for the purpose of our test). Peaceman well index models [17] are used to compute the discrete values of q_w and q_g where these wells are perforated (well radius is set to 0.5 and the skin factor is zero). This injection-production process is simulated during $t_F = 10$ years with an initial time step $\tau^0 = 8.64 \times 10^3$ s, which is equal to 0.1 days. If one time step iteration is completed, the time step is increased by a factor of 1.5. We divide it by 2 if a divergence of the Newton algorithm occurs.

The injection pressure is set constant in time and equal to $P_{inj} = 5.6e7$. In the same way, the producer pressure is fixed to $P_{pro} = 2.7e7$.



(a) Five-spot pattern **(b)** Permeability

Figure 6. Test-case configuration of Section 4.3.

The problem parameters are chosen as follows:

- $\phi = 0.1$, $\mathbb{K}$ is given by a geostatistical realization shown in Figure 6(b);
- $\rho_{\mathrm{w}} = 1025$, $\mu_{\mathrm{w}} = 1e-3$;
- $\rho_{\mathrm{g}} = 1.89$, $\mu_{\mathrm{g}} = 1.42e-5$;
- a Brooks–Corey model [30] was used for the petrophysical properties:
  - the relative permeability of a phase $p \in \{\mathrm{w}, \mathrm{g}\}$ is given by

$$k_{\mathrm{r},p}(S_p) = \begin{cases} 1 & \text{if } S_p \geq 1, \\ \left(\frac{S_p - S_p^{\mathrm{res}}}{1 - S_p^{\mathrm{res}}}\right)^n & \text{if } S_p^{\mathrm{res}} < S_p < 1, \\ 0 & \text{if } S_p \leq S_p^{\mathrm{res}}, \end{cases} \tag{13}$$

    where the residual saturations are respectively given by $S_{\mathrm{w}}^{\mathrm{res}} = 0.2$ and $S_{\mathrm{g}}^{\mathrm{res}} = 0.1$ and $n = 1$;
  - $P_{c_{\mathrm{g,w}}}$ is given by

$$P_{c_{\mathrm{g,w}}}(S_{\mathrm{g}}) = P_e \cdot (S_e)^m, \qquad S_e = 1 - \frac{S_{\mathrm{g}} - S_{\mathrm{g}}^{\mathrm{res}}}{1 - S_{\mathrm{w}}^{\mathrm{res}} - S_{\mathrm{g}}^{\mathrm{res}}},$$

    with $P_e = 8.73e5$, $m = -\frac{1}{2.89}$.

As for the previous test case, a two-point finite volume scheme with upwinding in space and the backward Euler scheme in time are applied to discretize the system (1), (10)–(12). Newton's method [16] along with LU solver is used to solve the resulting nonlinear system.

Figure 7 shows the evolution of the water saturation at three different time steps and, in particular, the faster front displacement towards the upper left and lower right corners where larger permeability values are present, see Figure 6(b) too.

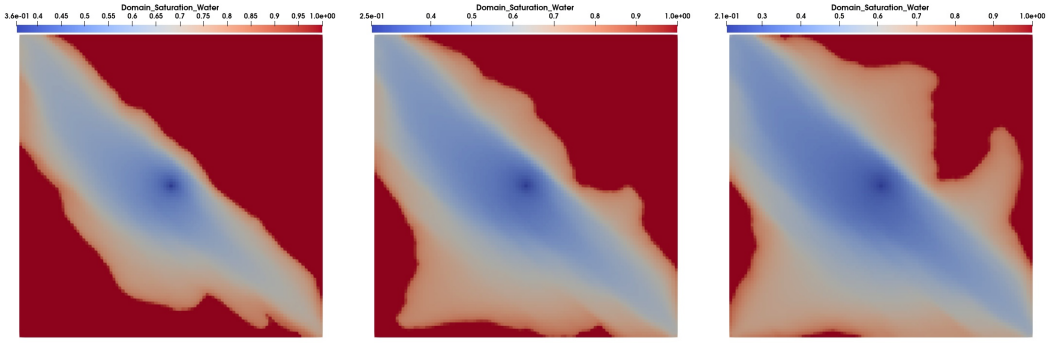


Figure 7. Water saturation at 0.5 year (left), 1 year (middle), and 1.5 year (right).

4.3.2. *Implementation within ShArc*

As for the previous example, this section gives an illustration of the use of the GUMP data model and of the law framework to define the capillary pressure: Listing 24 shows the line defining this property in GUMP, Listing 25 the C++ code defining the law, and Listing 26 the definition of the parameters of this law in the input file. At last, Listings 27 and 28 detail the construction of the *Contributions* required for the phase fluxes, their evaluation and their storage into the matrix and into the right-hand side of the linear system.

```
<property name="CapillaryPressure" dim="scalar" type="real" />
```

Listing 24. Definition of capillary pressure GUMP property.

```
void CapillaryPressureLaw::eval(const Real S, Real& Pc, Real& dPc_dS)const
{
    Real Se = Arcane::math::min(1 - (S - m_Sr)/(1 - m_Sr_ref -m_Sr), 1.0);
    Real dSe_dS = - 1.0/(1 - m_Sr_ref -m_Sr);
    Real alpha = - 1./m_lambda;
    Pc = m_Pe * pow(Se, alpha);
    dPc_dS = alpha * m_Pe * dSe_dS * pow(Se, alpha - 1);
}
```

Listing 25. C++ code computing the capillary pressure law

```
    <law name="CapillaryPressureLawConfig">
      <output>
        <capillary-pressure>[Phase]Gas::CapillaryPressure</capillary-pressure>
      </output>
      <input>
        <saturation>[Phase]Gas::Saturation</saturation>
      </input>
      <parameters>
        <Pe>8.73e5</Pe>
        <Sr-ref>0.2</Sr-ref>
        <Sr>0.1</Sr>
        <lambda>2.89</lambda>
      </parameters>
    </law>
```

Listing 26. Input simulation file defining the capillary pressure law parameters.

```
const Arcane::VariableFaceReal& T = m_transmissivities["T"];
auto P  = Law::contribution<ArcRes::Pressure>(...,system());
auto Pc = Law::contribution<ArcRes::CapillaryPressure>(...,fluid.phases());

auto mu  = Law::contribution<ArcRes::Viscosity>(...,fluid.phases());
auto rho = Law::contribution<ArcRes::Density>(...,fluid.phases());
auto kr  = Law::contribution<ArcRes::RelativePermeability>(...,fluid.phases());
```

Listing 27. C++ code retrieving the *Contributions* for the evaluation of the phase fluxes.

```
const auto grad_kl = T[iface] * (P[cell_k] + Pc[iphase][cell_k]
                                 - P[cell_l] - Pc[iphase][cell_l]);

const auto mobility_k =
        rho[iphase][cell_k] * kr[iphase][cell_k] / mu[iphase][cell_k];

const auto mobility_l =
        rho[iphase][cell_l] * kr[iphase][cell_l] / mu[iphase][cell_l];

const auto flux_kl = (audi::value(grad_kl)>=0) ? mobility_k*grad_kl
                                               : mobility_l*grad_kl ;

residual[iequation][cell_k]           += flux_kl;
jacobian[iequation][cell_k][stencil] += flux_kl;
```

Listing 28. C++ code building an upwind TPFA phase flux using the *Contributions*.

5. Massively parallel simulations on a simplified two-phase flow model

5.1. *Tests on* Irene *supercomputer*

We now present the first scalability tests performed on the Rome partition of *Irene* supercomputer [31]. The Rome partition contains 2292 dual-processor AMD Rome (Epyc) nodes at 2.6 GHz with 64 cores per processor, for a total of 293,376 computing cores. All nodes are connected through a HDR-100 Infiniband network.

The physical model used for these tests is similar to the one presented in Section 4.3. As a first attempt, it has been deliberately simplified in order to first test the code on larger grids than the ones usually used with other Arcane applications. This simplified model differs in terms of boundary conditions, rock and fluid properties. Namely, here:

- $\Omega = (0, L) \times (0, L) \times (0, H)$ with $L = 80000$, $H = 1000$, $t_F = 100$ days;
- the relative permeability of a phase $p \in \{w, g\}$ is given by (13) with $n = 2$ and $S_w^{res} = S_g^{res} = 0$, $P_{c_{g,w}} = 0$;
- $\phi = 0.2$, $\mathbb{K} = e^{f(x,y,z)}$ with

$$f(x, y, z) = -30 + 3 * \sin(2 * \pi * x/L) * \sin(2 * \pi * y/L) + y/L;$$

- $\rho_w = \rho_g = 1000$, $\mu_w = \mu_g = 10^{-3}$;
- $q_w = q_g = 0$, no-flow condition are used on $\partial\Omega$ except on $x = 0$ and $x = L$ where $P\big|_{x=0} = 8.10^7$, $S_g\big|_{x=0} = 1$, $P\big|_{x=L} = 10^7$.

The computation domain Ω is discretized with a Cartesian grid of size $N^2 \times N_z$ with $N = 2048$ and $N_z = 100$. The time step evolution was chosen according to the evolution of Newton's iterations starting with an initial time step of one day with a maximal value equal to 10 days. The linear systems were solved with a BiCGStab method preconditioned with CprAMG (Hypre Boomer AMG + block Jacobi ILU0). Note that the defined grid and model lead to ≈ 1.258 billion equations at each Newton iteration. Starting from 10 nodes, MPI parallel simulations were carried out up to 128 nodes (16,384 cores) without changing the grid size (strong scalability study). Figure 8 shows the scalability of ShArc's time loop from 10 up to 128 nodes (blue curve) and the theoretical linear performance compared to the 10-node run (red curve) which is the lowest number of nodes that can run this test case. These results were obtained as part of the Genci *Grands Challenges 2024* [32]. We observe a super-linear scalability of ShArc performances up to 100 nodes due to better cache usage as per core problem size decreases. On this range of nodes, this study has highlighted a parallel efficiency of ShArc's time loop at around 90%.

Secondly, simulations have also been carried out at the limit of the Alien platform's capacity:[2] a grid containing 700 million of elements or so was distributed on 256 nodes (32,768 cores and 2.1 billion equations on the whole). These tests performed on Irene supercomputer have also provided an opportunity to compare the performances of Alien's MCGSolver in both MPI and MPI-OpenMP contexts. Further details are given in [33]. The input data file of these case can be found on GitHub [34].

5.2. *Tests on IFPEN* Ener440 *supercomputer*

In this second series of parallel tests, we take the model a step further introducing a capillary pressure P_c, given in the description below. The tests are executed on IFPEN *Ener440* supercomputer, composed of 240 computing nodes with dual Intel Skylake-X G-6140 processors clocked at 2.3 GHz. Each node holds 2×18 computing cores so this supercomputer provides a total of 8640 computing cores. All nodes are connected with an Intel Omni-Path high speed network.

[2] Due to the equation indexing with 32-bit integers.

Figure 8. Scalability of ShArc's time loop from 10 up to 128 nodes (blue curve) and the theoretical linear performance compared to the 10-node run (red curve).

The physical model used for these tests is close to the model used with *Irene* supercomputer, but with a capillary pressure P_c. It has the following characteristics:

- $\Omega = (0,L)\times(0,L)\times(0,H)$ with $L = 20000$, $H = 100$, $t_\mathrm{F} = 1000$ days;
- $\rho_\mathrm{w} = \rho_\mathrm{g} = 1000$, $\mu_\mathrm{w} = 10^{-3}, \mu_\mathrm{g} = 10^{-4}$;
- $q_\mathrm{w} = q_\mathrm{g} = 0$, no-flow condition are used on $\partial\Omega$ except on $x = 0$ and $x = L$ where $P\big|_{x=0} = 6.10^7$, $S_\mathrm{g}\big|_{x=0} = 1$, $P\big|_{x=L} = 10^7$;
- $P_{c_\mathrm{g,w}}$ is given by

$$P_{c_\mathrm{g,w}}(S_\mathrm{g}) = P_e\cdot(S_e)^m, \qquad S_e = 1 - \frac{S_\mathrm{g} - S_\mathrm{g}^\mathrm{res}}{1 - S_\mathrm{w}^\mathrm{res} - S_\mathrm{g}^\mathrm{res}},$$

with $P_e = 10^5$, $m = -1$, $S_\mathrm{w}^\mathrm{res} = 0.2$ and $S_\mathrm{g}^\mathrm{res} = 0.1$.

The domain Ω is discretized with a Cartesian grid of size $N^2 \times N_z$ with $N = 2048$ and $N_z = 10$.

The performance results in Figure 9 are similar to those observed with *Irene* system but at a lower scale: N_z is reduced from 100 to 10 and tests are performed up to 20 nodes (720 cores). Relative efficiency, compared to the 3 nodes test, is > 94%. The test case is also available at [34].

6. Conclusion

The ArcNum framework presented in this article offers easy-to-use numerical tools significantly simplifying the writing of numerical methods within scientific computing codes. These numerical tools are built upon the Arcane platform, providing mesh management, data structures and parallelism. For linear system resolutions, they rely on the Alien library, giving access to a wide range of linear solvers. Used together, ArcNum, Arcane, and Alien offer a complete toolbox for building numerical simulation applications. This framework has been used to build the open

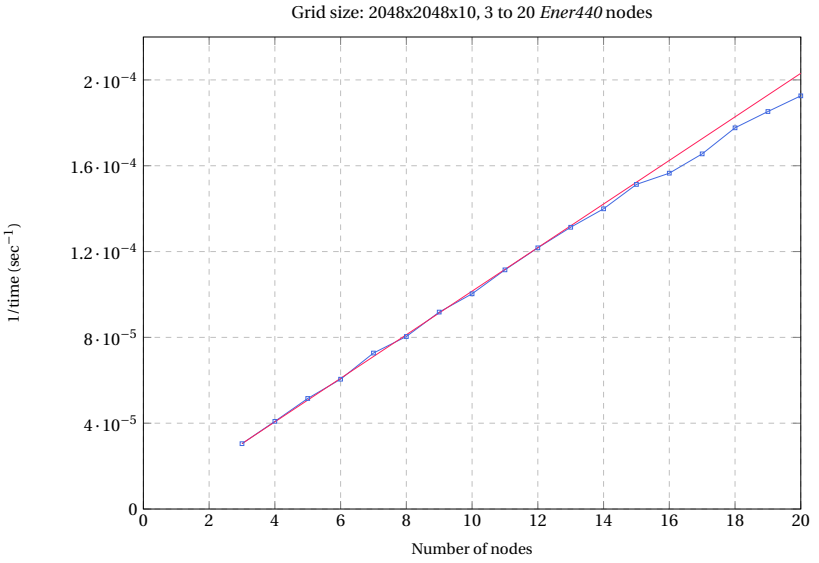


Figure 9. Scalability of ShArc's time loop from 3 up to 20 nodes (blue curve) and the theoretical linear performance compared to the 3-node run (red curve).

source proxy-app ShArc for Geosciences simulation. The two case studies presented in this publication and conducted with ShArc highlight the flexibility, development time savings and robustness of the ArcNum framework. A scalability study shows that this gain is not at the expense of parallel performance.

The ArcNum framework is also used within complex industrial applications at IFPEN. For example, it is the foundation of a compositional multiphase reactive transport simulator, coupled with mechanics. In this context, more advanced schemes, such as (non)linear multipoint finite volume and virtual element methods, have been implemented [35,36]. In this much more demanding context, the framework has been used with more complex data models, a great number of physical laws and varied assembly algorithms. It was therefore necessary to add new functionalities to ensure the needed expressiveness and maintain the performance requirement. More precisely, the following features were added to the framework for internal use at IFPEN:

- the ability to create a graph of laws to enable the composition of physical laws;
- the management of dynamic multi-point stencils;
- a sparse automatic differentiation.

The next step for this framework will concern the physical law package, where the usage of AI trained models and the ability to compute asynchronously on accelerators will be added.

Declaration of interests

The authors do not work for, advise, own shares in, or receive funds from any organization that could benefit from this article, and have declared no affiliations other than their research organizations.

References

[1] G. Grospellier and B. Lelandais, "The Arcane Development Framework", in *POOSC '09: Proceedings of the 8th Workshop on Parallel/High-Performance Object-Oriented Scientific Computing* (K. Davis, ed.), ACM Press, 2009, (11 pages).

[2] CEA/IFPEN, *The Arcane Framework organization*. Online at https://github.com/arcaneframework (accessed on November 3, 2025).

[3] J.-M. Gratien, C. Chevalier, T. Guignon, X. Tunc, P. Have and S. de Chaisemartin, "Evaluation of the performance portability layer of different linear solver packages with ALIEN, an open generic and extensible linear algebra framework", Conference paper : The 8th European Congress on Computational Methods in Applied Sciences and Engineering (ECCOMAS Congress 2022), 2022. Online at https://www.scipedia.com/public/gratien_et_al_2022a.

[4] CEA/IFPEN, *The ShArc proxy-app repository*. Online at https://github.com/arcaneframework/sharc.git (accessed on November 3, 2025).

[5] Inria, *The ShArc proxy-app installation guide repository*. Online at https://gitlab.inria.fr/numpex-pc5/wp2-co-design/proxy-sharc.git (accessed on November 3, 2025).

[6] CEA/IFPEN, *The Arcane platform repository*. Online at https://github.com/arcaneframework/framework.git (accessed on November 3, 2025).

[7] A. Anciaux-Sedrakian, P. Gottschling, J.-M. Gratien and T. Guignon, "Survey on Efficient Linear Solvers for Porous Media Flow Models on Recent Hardware Architectures", *Oil Gas Sci. Technol.* **69** (2014), pp. 753–766.

[8] The Khronos Group, *SYCL*, 2020 Specification. Online at https://www.khronos.org/sycl/ (accessed on November 3, 2025).

[9] H. C. Edwards, C. R. Trott and D. Sunderland, "Kokkos: Enabling manycore performance portability through polymorphic memory access patterns", *J. Parallel Distrib. Comput.* **74** (2014), no. 12, pp. 3202–3216.

[10] CEA/IFPEN, *Arcane benchs and mini apps repository*. Online at https://github.com/arcaneframework/arcane-benchs.git (accessed on November 3, 2025).

[11] CEA/IFPEN, *The ArcaneFEM proxy-app repository*. Online at https://github.com/arcaneframework/arcanefem.git (accessed on November 3, 2025).

[12] T. Veldhuizen, "Expression Templates", *C++ Rep.* **7** (1995), no. 5, pp. 26–31.

[13] D. Vandevoorde and N. Josuttis, *C++ Templates: The Complete Guide*, Addison-Wesley Publishing Group, 2002.

[14] H.-J. G. Diersch and O. Kolditz, "Coupled groundwater flow and transport: 2. Thermohaline and 3D convection systems", *Adv. Water Resources* **21** (1998), no. 5, pp. 401–425.

[15] R. Eymard, T. Gallouët and R. Herbin, "The finite volume method", in *Solution of Equation in $\mathbb{R}^n$ (Part 3), Techniques of Scientific Computing (Part 3)* (P. G. Ciarlet and J.-L. Lions, eds.), Handbook of Numerical Analysis, vol. 7, Elsevier, 2000, pp. 713–1020.

[16] C. T. Kelley, *Solving nonlinear equations with Newton's method*, Society for Industrial and Applied Mathematics, 2003.

[17] Z. Chen, *Reservoir simulation: mathematical techniques in oil recovery*, Society for Industrial and Applied Mathematics, 2007.

[18] G. Acs and E. Farkas, "General Purpose Compositional Model", *SPE J.* **25** (1985), no. 4, pp. 543–553.

[19] K. Aziz and A. Settari, *Petroleum Reservoir Simulation*, Applied Science Publishers, 1979.

[20] K. H. Coats, "An Equation of State Compositional Model", *SPE J.* **20** (1980), no. 5, pp. 363–376.

[21] L. C. Young and R. E. Stephenson, "A Generalized Compositional Approach for Reservoir Simulation", *SPE J.* **23** (1983), no. 5, pp. 727–742.

[22] A. Anciaux-Sedrakian, L. Grigori, Z. Jorti and S. Yousef, "Adaptive linear solution process for single-phase Darcy flow", *Oil Gas Sci. Technol.* **75** (2020), article no. 54 (11 pages).

[23] D. A. Di Pietro, E. Flauraud, M. Vohralík and S. Yousef, "A posteriori error estimates, stopping criteria, and adaptivity for multiphase compositional Darcy flows in porous media", *J. Comput. Phys.* **276** (2014), pp. 163–187.

[24] D. A. Di Pietro, M. Vohralík and S. Yousef, "An a posteriori-based, fully adaptive algorithm with adaptive stopping criteria and mesh refinement for thermal multiphase compositional flows in porous media", *Comput. Math. Appl.* **68** (2014), no. 12, Part B, pp. 2331–2347.

[25] J.-M. Gratien, O. Ricois and S. Yousef, "Reservoir Simulator Runtime Enhancement Based on a Posteriori Error Estimation Techniques", *Oil Gas Sci. Technol.* **71** (2016), no. 5, article no. 59 (11 pages).

[26] G. Mallik, M. Vohralík and S. Yousef, "Goal-oriented a posteriori error estimation for conforming and nonconforming approximations with inexact solvers", *J. Comput. Appl. Math.* **366** (2020), article no. 112367 (20 pages).

[27] M. Vohralík and S. Yousef, "A simple a posteriori estimate on general polytopal meshes with applications to complex porous media flows", *Comput. Methods Appl. Mech. Eng.* **331** (2018), pp. 728–760.

[28] S. Yousef, "A posteriori-based, local multilevel mesh refinement for the Darcy porous media flow problem", *J. Comput. Appl. Math.* **454** (2025), article no. 116162 (10 pages).

[29] M. A. Puscas, G. Enchéry and S. Desroziers, "Application of the mixed multiscale finite element method to parallel simulations of two-phase flows in porous media", *Oil Gas Sci. Technol.* **73** (2018), article no. 38 (14 pages).

[30] R. J. Brooks and A. T. Corey, *Hydraulic properties of porous media*, Hydrology Paper, Colorado State University, no. 3, 1964.

[31] CEA, "Supercomputer architecture", in *TGCC public documentation, version 2025-10-10.1612*, 2025. Online at https://www-hpc.cea.fr/tgcc-public/en/html/toc/fulldoc/supercomputer_architecture.html (accessed on November 3, 2025).

[32] GENCI, "Grands Challenges Scalaires Joliot-Curie, Adastra 2024", in *Grands Challenges*. Online at https://www.genci.fr/grands-challenges#ui-id-1 (accessed on November 3, 2025).

[33] A. Anciaux-Sedrakian, R. Gayno, T. Guignon and A. K. Mohamed El Maarouf, "Efficient high-fidelity simulations for the energy transition using ARM and x86 64 architectures", Conference paper: The 9th European Congress on Computational Methods in Applied Sciences and Engineering (ECCOMAS Congress 2024), 2024. Online at https://www.scipedia.com/public/Anciaux-Sedrakian_et_al_2024a.

[34] CEA/IFPEN, *Sharc: Two phase flow test cases*. Online at https://github.com/arcaneframework/sharc/tree/main/test/LargeScaleTwoPhaseFlowSimulation/ (accessed on November 3, 2025).

[35] M. Schneider, L. Agélas, G. Enchéry and B. Flemisch, "Convergence of nonlinear finite volume schemes for heterogeneous anisotropic diffusion on general meshes", *J. Comput. Phys.* **351** (2017), pp. 80–107.

[36] G. Enchéry and L. Agélas, "Coupling linear virtual element and non-linear finite volume methods for poroelasticity", *Comptes Rendus. Mécanique* **351** (2023), no. S1, pp. 395–410.