



Usability and Optimization Improvements in MOOSE

Sept. 2022

Nuclear Energy Advanced Modeling and Simulation M2 Milestone, September 2022

Alexander Lindsay¹, Roy Stogner¹, Guillaume Giudicelli¹, Derek Gaston¹, and Cody Permann¹

¹COMPUTATIONAL FRAMEWORKS



*INL is a U.S. Department of Energy National Laboratory
operated by Battelle Energy Alliance, LLC*

DISCLAIMER

This information was prepared as an account of work sponsored by an agency of the U.S. Government. Neither the U.S. Government nor any agency thereof, nor any of their employees, makes any warranty, expressed or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness, of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. References herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise, do not necessarily constitute or imply its endorsement, recommendation, or favoring by the U.S. Government or any agency thereof. The views and opinions of authors expressed herein do not necessarily state or reflect those of the U.S. Government or any agency thereof.

Usability and Optimization Improvements in MOOSE

**Nuclear Energy Advanced Modeling and Simulation M2 Milestone,
September 2022**

**Alexander Lindsay¹, Roy Stogner¹, Guillaume Giudicelli¹, Derek Gaston¹, and Cody
Permann¹**

¹COMPUTATIONAL FRAMEWORKS

Sept. 2022

**Idaho National Laboratory
Computational Frameworks
Idaho Falls, Idaho 83415**

<http://www.inl.gov>

**Prepared for the
U.S. Department of Energy
Office of Nuclear Energy
Under DOE Idaho Operations Office
Contract DE-AC07-05ID14517**

Page intentionally left blank

SUMMARY

The Multiphysics Object-Oriented Simulation Environment (MOOSE) framework is a foundational capability used by the Nuclear Energy Advanced Modeling and Simulation (NEAMS) program to create over 15 different simulation tools for advanced nuclear reactors. Due to MOOSE's broad use, improvements to the framework in support of modeling and simulation goals are critical to the program. Such improvements can take many forms, including optimization, improved user experience, streamlined application programming interfaces (APIs), parallelism, and new capabilities. The work transcribed in this report was conducted in direct support of the simulation tools and has already been deployed. The capabilities were implemented in the same order as they are covered in this report: increased support of face variables for neutronics applications, arbitrary spatial and temporal evaluation of material properties, and the addition of a triangular meshing library in libMesh. These three additions are fundamental capabilities that will be leveraged by many NEAMS applications.

Page intentionally left blank

CONTENTS

SUMMARY	iii
1 Introduction	1
1.1 MOOSE	1
1.2 libMesh	2
2 Lower-Dimensional Variables	2
3 Arbitrary Location Material Property Evaluation	5
4 Native 2D Delaunay Triangulator	11
5 Conclusion	16
REFERENCES	17

FIGURES

Figure 1. Solution projection unit test, onto a mesh of 8 HEX27 elements, colored by a second-order SIDE_HIERARCHIC variable. Values are biquadratic on each QUAD9 element side, discontinuous between sides.	3
Figure 2. Potential inter-element flux interpretation of higher-order SIDE_HIERARCHIC data on TRI6 elements.	4
Figure 3. Solution projection unit test, visualized with a “Crinkle Clip” filter applied to remove some exterior faces from the scene and thus make interior face values visible.	5
Figure 4. Steady-state results for a high-temperature reactor (i.e., the HTR-PM) model. a) Fluid temperature. b) Solid temperature. c) Y-component of the superficial velocity.	9
Figure 5. Results at 200,000 seconds in a pressurized loss-of-forced-cooling accident scenario for the HTR-PM. a) Fluid temperature. b) Solid temperature. c) Y-component of the superficial velocity.	10
Figure 6. 1D flow channel porosity jump example, both with and without the porosity smoothing enabled by the functor materials system.	11
Figure 7. Graded mesh generation: a mesh with holes, generated using a spatially varying area function followed by a smoothing pass.	14
Figure 8. Nested mesh generation: a quad mesh inside a series of triangulated diamonds and squares.	16

TABLES

Table 1. Description of spatial argument types for functor evaluations.	7
--	---

ACRONYMS

FE	Finite Element
FV	Finite Volume
MOOSE	Multiphysics Object-Oriented Simulation Environment
NEAMS	Nuclear Energy Advanced Modeling and Simulation
TA	Technical Area

Page intentionally left blank

1. Introduction

The NEAMS program aims to develop simulation tools in support of the nuclear industry. These tools are meant to accelerate reactor design, licensing, demonstration, and deployment. The program is split into five Technical Areas (TAs): Fuel Performance, Thermal Fluids, Structural Materials and Chemistry, Reactor Physics, and Multiphysics Applications. The “physics” TAs involve delivering simulation tools to vendors, laboratories, and licensing authorities, whereas the Multiphysics Applications TA creates foundational capabilities for the program and exercises the tools in order to ensure their usability for the intended purposes.

Reactors are inherently multiphysical: heat conduction, neutronics, solid mechanics, fluid flow, chemistry, and material evolution all combine to create a complex system that is difficult to simulate. To tackle that problem, the NEAMS program utilizes the MOOSE platform [1] to develop interoperable physics applications. Over 15 MOOSE-based physics applications are being developed within the program, with at least one in every TA. Each physics application focuses on a particular aspect of reactor simulation (e.g., Bison [2] for nuclear fuel performance and Griffin [3] for neutronics).

It is therefore critical to the program that MOOSE continues to be enhanced and supported. Capabilities and optimizations made to the framework are instantly available to all NEAMS applications, making for a large return on investment.

1.1 MOOSE

The MOOSE platform enables rapid production of massively parallel, multiscale, multiphysics simulation tools based on finite element, finite volume, and discrete ordinate discretizations. This platform was developed as open-source software on GitHub [4] and utilizes the Lesser General Public License v2.1, thus allowing for a large amount of flexibility. It is an active project, with dozens of code modifications being merged weekly.

The core of the platform is a pluggable C++ framework that enables scientists and engineers to specify all the details of their simulations. Certain interfaces include finite element/volume terms, boundary conditions, material properties, initial conditions, and point sources. By modularizing numerical simulation tools, MOOSE allows for an enormous amount of reuse and flexibility.

Beyond its core framework, the MOOSE platform also provides myriad supporting technologies for application development. This includes a build system, a testing system for both regression and unit testing, an automatic documentation system, visualization tools, and many physics modules. The physics modules are a set of common physics utilizable by application developers. Some of the more important modules are the solid mechanics, heat conduction, fluid flow, chemistry, and phase-field modules. All this automation and reuse accelerates application development within the NEAMS program.

1.2 libMesh

Underpinning the MOOSE framework is the libMesh finite element framework [5]. libMesh is an open-source software originally developed at the University of Texas at Austin’s CFDLab. It provides an enormous amount of numerical support, including the mesh data structures, element discretizations, shape-function evaluations, numerical integration, and interfaces to various linear and nonlinear solvers (e.g., PETSc) [6]. MOOSE and libMesh were developed in lockstep with each other, with any change to one being immediately tested against the other. This symbiotic relationship has worked extremely well over the 14 years of MOOSE development, partly due to Idaho National Laboratory having employed multiple libMesh developers over the years.

2. Lower-Dimensional Variables

libMesh now supports a `SIDE.HIERARCHIC` finite element type, which implements a hierarchic space of discontinuous polynomial (in “master space”) basis functions defined specifically on element sides rather than in element interiors. When a user instantiates it with polynomial order p , this basis is the product of, on each element side S , the tensor product polynomial space $Q^p(S)$. Support for triangular-sided elements with $P^p(S)$ function spaces has begun, with initial support for tetrahedra now merged and passing tests, and support for pyramid and prism elements to be added in future work. On each side S of a d -dimensional element, the function space is parameterized using the same $d - 1$ D local basis and local basis ordering as used by `HIERARCHIC` elements of that type. Global-to-local basis ordering is permuted so that the same basis functions are chosen regardless of which neighboring d -dimensional element they are evaluated from. This

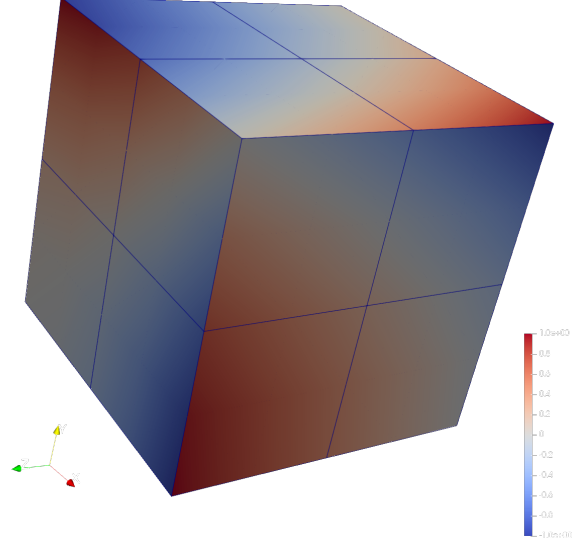


Figure 1: Solution projection unit test, onto a mesh of 8 HEX27 elements, colored by a second-order SIDE_HIERARCHIC variable. Values are biquadratic on each QUAD9 element side, discontinuous between sides.

finite element type is of the new SIDE_DISCONTINUOUS continuity type, meaning that values on any domain-internal side S are shared between the two elements whose boundary intersection is S , but the values are discontinuous between different sides meeting at the same mesh edge (or in 2D, vertex), as shown in the example given in Fig. 1. Values on element interiors are undefined, but for maximum robustness against floating-point error, values are returned based on a query point's position within the cone connecting an element centroid to an element side, as evaluated in the “master space,” where quadrature points and basis functions are naturally defined.

The initial use case for this finite element basis was simply an easier-to-use and more automated form of data storage: in the base case of $p = 0$, the basis is piecewise constant, with a single real-valued (or complex-valued, in those configurations) datum associated with each side of each variable. By adding some number of SIDE_HIERARCHIC variables to an explicit auxiliary system associated with a mesh, an application code obtains data storage for the many variables on each mesh side. Unlike application-level storage, however, these variables are automatically kept in sync via mesh distribution and redistribution. Values are made available to kernels at each side quadrature point, just as with any other solution or auxiliary variable, requiring no C++ data structure coding to manage.

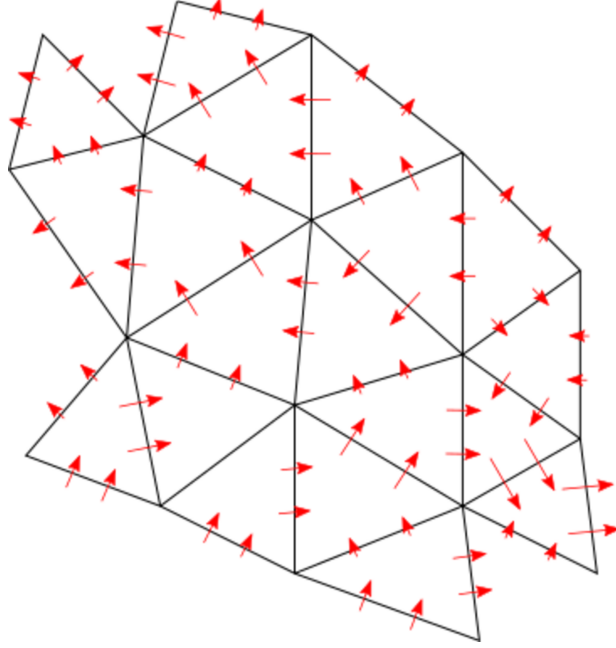


Figure 2: Potential inter-element flux interpretation of higher-order `SIDE_HIERARCHIC` data on TRI6 elements.

For future use cases, the ability to elevate the polynomial degree of these variables also becomes important. Simple discontinuous-Galerkin-type finite element methods have always been supported in libMesh via a few discontinuous L_2 finite element types, but more sophisticated stabilized formulations (e.g., employing inter-element flux variables or jump variables defined on element sides) are now attainable as well. The sketch in Fig. 2 shows one potential interpretation, in which scalar flux magnitudes stored as a `SIDE_HIERARCHIC` variable are interpreted to have a vector direction (based on the normal vectors computed on element sides) and sign (based on an element ordering rule) to represent the normal component of a flux between elements, then evaluated on two quadrature points on each element side.

One limitation of the current architecture is that side-discontinuous variables cannot currently be evaluated on element side boundaries (vertices in 2-D, vertices and edges in 3-D); for safety purposes, the code returns NaN whenever such an evaluation is requested. This limits the choice of quadrature rules that are usable when integrating these variables on element sides. The Gaussian quadrature rules, which libMesh uses by default and which are most appropriate for high-order accurate integration of smooth functions, perform fine, but the trapezoidal and Simpson rules that are popular for some special applications are not yet usable here.

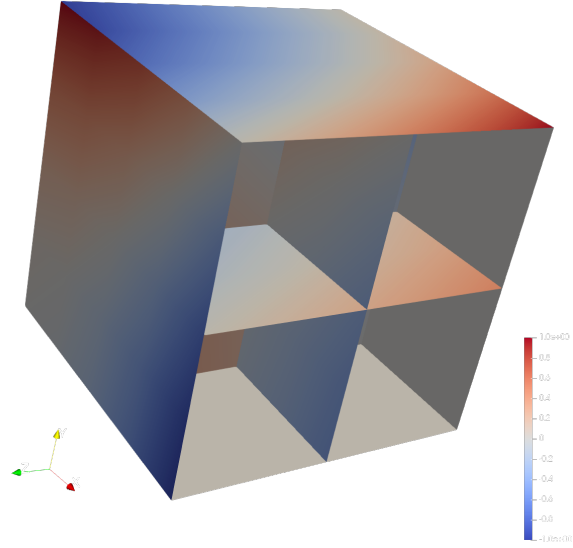


Figure 3: Solution projection unit test, visualized with a “Crinkle Clip” filter applied to remove some exterior faces from the scene and thus make interior face values visible.

libMesh support has been added for visualizing these variables via ExodusII output files. When “added_sides” support is enabled on the libMesh ExodusII writer object, fictitious side elements are added (with a lower-dimensional element type, in a separate block), and as with ordinary elements, these fictitious elements are also output along with all solution variables evaluated on all nodes. Using visualization software and filtering by element block allows users to easily see side variables on exterior sides. Adding a clipping plane to a filter pipeline, as in Fig. 3, allows users to see interior side values as well.

Extensive unit testing of these new spaces, as well as the ExodusII output of various mixed system variable configurations, have been added to the libMesh test suite.

3. Arbitrary Location Material Property Evaluation

The Finite Element (FE) method usually features a small element stencil with respect to the formation of sparsity patterns. Continuous Galerkin, for example, only involves the current element, while the discontinuous Galerkin method involves the current element and its neighbors. Given the small stencil size, it is memory and programmatically efficient to pre-initialize data for use in physics kernels and boundary conditions. The Finite Volume (FV) method, on

the other hand, almost always involves degrees of freedom on neighbors of neighbors, and not uncommonly involves degrees of freedom on neighbors of neighbors of neighbors. This significantly larger stencil size (compared to the FE method) makes it cumbersome to pre-initialize the requisite data, both from a programmatic (code maintenance) and memory point of view. To avoid this issue, we created an on-the-fly evaluation system in MOOSE, which is usable by both FV code (the primary user) and FE code.

The Functor class lies at the root of the on-the-fly evaluation system. Functors have three different APIs that user code can call: `operator()`, `gradient`, and `dot`. These represent values, spatial gradients, and time derivatives, respectively. These APIs may be called with seven different spatial arguments and one temporal argument. The different kinds of spatial arguments, which are either `struct` or `std::tuple`, are summarized in Table 1. The single temporal argument is an unsigned integer that reflects the temporal “state,” with 0 corresponding to the current time, 1 corresponding to the previous time, etc.

If the user has requested that cached data be stored, and cached data are available for the calling spatial and temporal arguments, then the three aforementioned APIs (i.e., `operator()`, `gradient`, and `dot`) return the cached data. If caching was not requested or there are no previous evaluations cached for the calling arguments, the public APIs redirect to protected `evaluate`, `evaluateGradient`, and `evaluateDot` routines, which must be implemented by Functor-derived classes. Users commonly interact with three types of functors: variables (`MooseVariableField`), functions (`Function`), and functor material properties (`FunctorMaterialProperty`). `FunctorMaterialProperty` is actually an alias for the class template `PiecewiseByBlockLambdaFunctor`. As is consistent with the latter name, a functor material property describes its evaluation by using a C++ lambda. Functor material properties are added to the simulation through classes derived from `FunctorMaterial`—analogous to how traditional pre-initialized material properties are added through classes derived from `Material`. An example of functor material property addition is shown in Listing 1. The template argument provided to `addFunctorProperty` defines the return type of the functor material property; the first function argument to `addFunctorProperty` defines the global simulation name of the functor material property for the purpose of retrieval by consumers. The second function argument to `addFunctorProperty` is the lambda describing the functor material property evaluation. In the

Argument Name	Discretization Types	Description
ElemArg	FV	Corresponds to an element-average evaluation of the functor for the provided element.
ElemFromFaceArg	FV	Evaluates the functor at a face with <i>sidedness</i> (e.g., for a FV variable, this will return the element average value on either the + or - side of the face, depending on the argument). A continuous Function will return the value of the function evaluated right at the face position. Functor material property evaluations are comprised of variable and function evaluations.
FaceArg	FV	Evaluates the functor by performing a specified interpolation. For example, the interpolation may be of CentralDifference type, in which case a FV variable would be linearly interpolated between element/cell-centered values to the face.
SingleSidedFaceArg	FV	Akin to FaceArg, except that only data on one side of the face are considered. Consequently, this type of spatial argument is appropriate when an extrapolation is desired, as opposed to interpolation.
ElemQpArg	FV, FE	Evaluates the functor at a specified element quadrature point with a specified quadrature rule. This is appropriate for both FE and FV discretizations, since the FV variables in MOOSE are backed by a constant monomial FE type.
ElemSideQpArg	FV, FE	Very similar to ElemQpArg, but this argument type also includes element-side data. This type of argument makes sense in the context of boundary conditions, discontinuous Galerkin kernels, and interface kernels.
ElemPointArg	FV, FE	This argument type is composed of an element and a point within the element. The functor is evaluated using a two-term Taylor expansion based on the evaluation of both the element/cell-centered value and gradient. This type of argument is used in FV mortar constraints.

Table 1: Description of spatial argument types for functor evaluations.

example of Listing 1, we are creating a viscosity with a name defined by the parameter value of `NS::mu`. Evaluation of `NS::mu` corresponds to the product of a rampdown functor evaluation and fluid properties evaluation based on the current values of pressure and temperature functors. We forward the incoming spatial and temporal arguments `r` and `t` to all functors used within a functor material property evaluation.

```

1 const auto & mu = addFunctorProperty<ADReal>(
2     NS::mu,
3     [this](const auto & r, const auto & t) -> ADReal
4     { return _mu_rampdown(r, t) * _fluid.mu_from_p_T(_pressure(r, t), _T_fluid(r, t)); });

```

Listing 1: Example of functor material property addition, featuring a function functor (μ) and two variable functors (i.e., pressure and temperature), all chained to obtain the initialization viscosity.

Creation of the functor system has enabled a whole new class of simulation capabilities in MOOSE’s Navier-Stokes module: weakly compressible fluid flow. Previously, with density as a traditional material property, it was impossible to evaluate the density on arbitrary cells necessary for creating a Rhie-Chow interpolation of velocity. Outcomes from the new functor-enabled weakly compressible simulation capabilities are seen in Figure 4 and Figure 5, which show the results of steady-state and transient accident scenarios, respectively, for a high-temperature reactor (i.e., the HTR-PM) model.

The functor system also provides an elegant solution for handling discontinuities in the porosity applied to homogenized porous media simulations. These simulations are used to predict coolant flow in pebble-bed gas- or salt-cooled reactors, and are also deployed for conducting porous media homogenization on parts used in sodium fast reactor cores. These simulations almost always involve discontinuities in the porosity of the porous medium being used (e.g., at transitions between porous and unobstructed fluid flow regions, as well as at transitions between the pebbled regions and other homogenized regions such as the reflector). Discontinuities in the porosity cause oscillations in the fluid flow solutions, as Rhie-Chow interpolations of the velocities to the discontinuities assume a linear variation of pressure. The pressure is, in practice, discontinuous at the interface.

Using functors, the porosity functor can be replaced with chained averages of local and neighbor porosities. This is referred to as porosity smoothing. Each “layer” of porosity smoothing

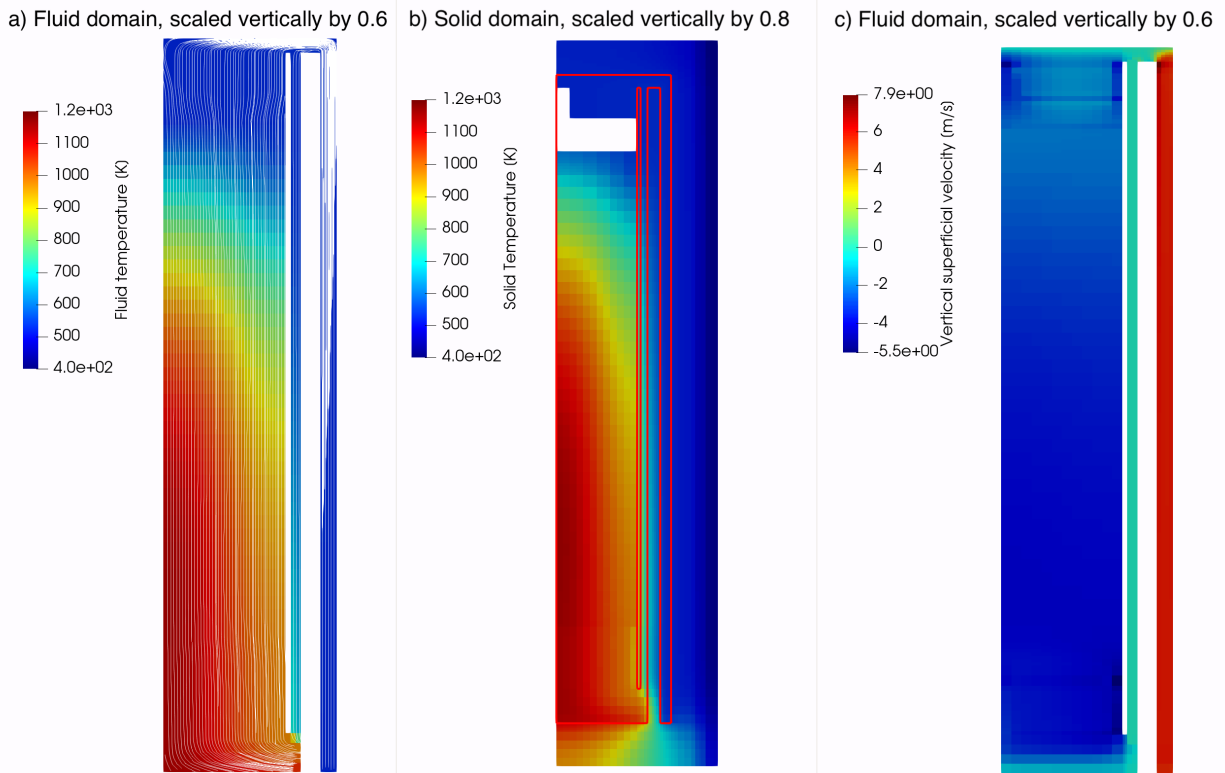
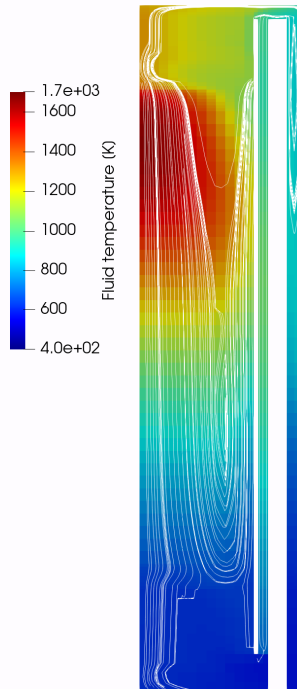
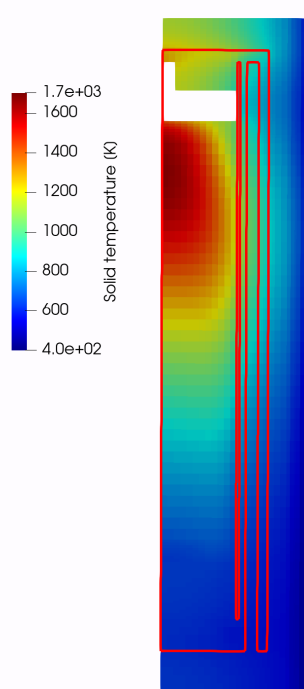


Figure 4: Steady-state results for a high-temperature reactor (i.e., the HTR-PM) model. a) Fluid temperature. b) Solid temperature. c) Y-component of the superficial velocity.

a) Fluid domain, scaled vertically by 0.6
Time = 200,000 sec



b) Solid domain, scaled vertically by 0.8



c) Fluid domain, scaled vertically by 0.6

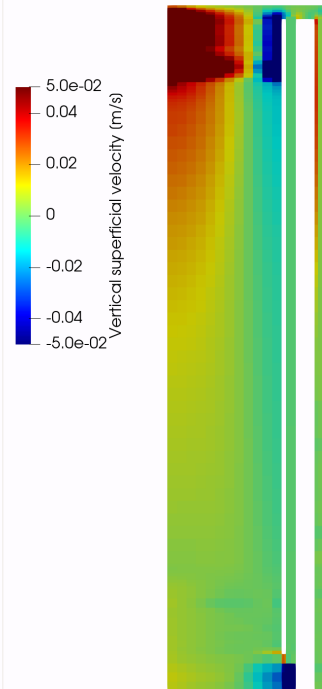


Figure 5: Results at 200,000 seconds in a pressurized loss-of-forced-cooling accident scenario for the HTR-PM. a) Fluid temperature. b) Solid temperature. c) Y-component of the superficial velocity.

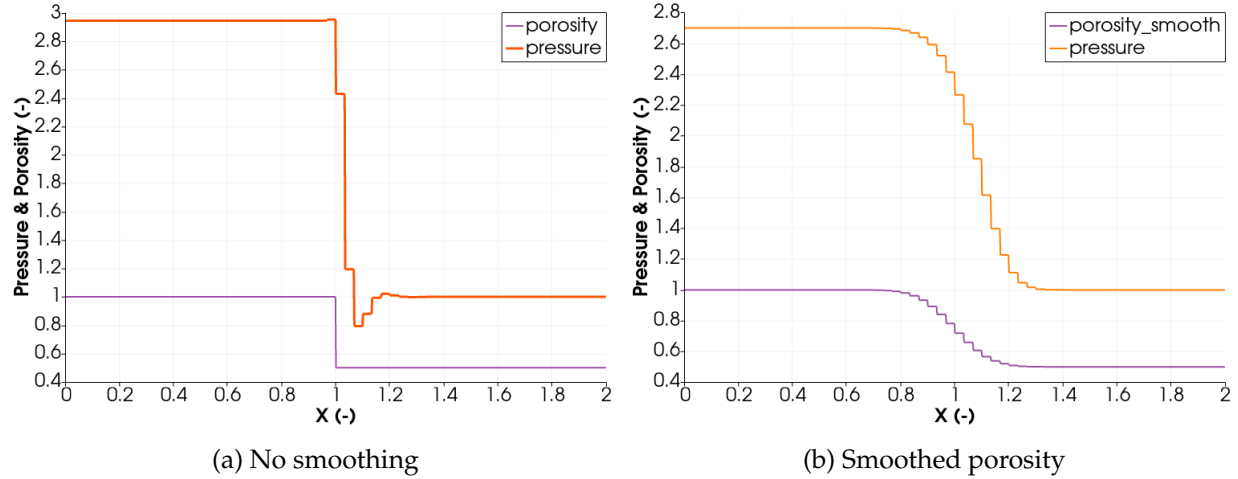


Figure 6: 1D flow channel porosity jump example, both with and without the porosity smoothing enabled by the functor materials system.

is an additional averaging of porosity, increasing the stencil to more cells around the porosity jump. This porosity smoothing may be performed with a function or a functor material property for the porosity, and may not involve dependence on local variables, as the stencil would require additional ghosting of variable values.

Figure 6 shows the porosity profile, along with the pressure and velocity solutions, both with and without porosity smoothing. Porosity smoothing removes oscillations by smoothing the discontinuity. A high number of porosity smoothing layers may be used to completely remove the oscillations, or additional techniques (e.g., friction correction and special treatments of the Rhie-Chow coefficients near the discontinuities) may be employed.

4. Native 2D Delaunay Triangulator

The libMesh library now incorporates Poly2Tri, a small open-source Delaunay triangulation library, as an optional sub-module. Its basic triangulation capabilities were initially incorporated into the MOOSE reactors module, with new MeshGenerator classes that create such triangulations around—and stitch them together with—other user-constructed mesh geometries, thus enabling the simple creation of a reactor cross-section mesh for extrusion into a simulation domain with hexahedra and triangular prism elements.

libMesh’s support for mesh stitching has also been broadened somewhat, allowing DistributedMesh objects to be stitched together when they have been only temporarily replicated

between processors, with the library automatically handling any needed communication if the meshes are in a distributed state when the stitching is requested. This was prompted to support workflows such as the above, where stitching is done on a relatively small mesh (for which the replicated-mesh stitching algorithm in libMesh is affordable); however, the eventual simulation will be performed on a much larger (extruded to 3D and/or refined to higher fidelity) mesh. After the stitching is complete but before the mesh is enlarged, distribution of the mesh object while leaving only local elements and a thin layer of “ghost” elements in memory on each processor can be accomplished, making the enlargement much more CPU efficient and the subsequent simulation much more memory efficient.

Within libMesh, the old interface to Triangle (a popular triangulation library, unfortunately unusable with many MOOSE and other types of applications, due to licensing conflicts) was refactored to allow for multiple concrete implementations of the same abstract interface, and an implementation using Poly2Tri at the subclass level was written to finally enable usage of this interface within MOOSE and other applications. The pre-existing APIs for specifying boundary segments, holes, and other triangulation features are now also implemented in the new subclass, giving other libMesh + Triangle users a migration path to the newer code. The expanded C++ interface and new Poly2Tri-based implementation are maintained within the libMesh library, and a number of new features (along with error handling for invalid inputs, as well as associated test coverage for both) have been added and are maintained there. During development of the Poly2Tri-based triangulator, a numerical tolerance issue was discovered with the Poly2Tri library itself. The bug, a fix, and a new test to avoid regressions were all communicated upstream and eventually merged into the code. Another unit test for the bug is retained within libMesh.

The original abstract interface for triangulation holes was used with only minor extensions, but a number of additional `Hole` subclasses and subclass constructors were developed for both internal and external use. An `AffineHole` shim makes copying and tiling of a specified hole easier. An `ArbitraryHole(const Hole&)` constructor enables refinement of hole boundaries, even if the initial hole was a user-defined subclass rather than a fixed library class. Automatic refinement of hole boundaries can be individually allowed or disallowed. A `MeshedHole` allows a hole to be automatically defined based on elements in an existing mesh object: 1D edges, boundary sides of 2D elements, or subsets of both, as specified using sets of subdomain or boundary IDs. And in the

base class, a new `Hole::contains(Point)` method enables a great deal of consistency verification within the triangulator code.

Although `Poly2Tri` itself only supports the Delaunay triangulation of a fixed set of points and boundaries, the `libMesh` code was written to support additional capabilities applied both before and after the `Poly2Tri` triangulation is called, as well as within iterative loops of triangulation steps. Before a point set is triangulated, users can call `set_interpolate_boundary_points()` on the triangulator to conduct uniform interpolation of points into the interior of their mesh boundary segments, making it easy for them to specify polygonal boundary geometry independently of boundary mesh fineness. After a point set is triangulated, `libMesh`-style boundary ID values are set (a distinct value on the outer boundary and another on each hole). Users then have the option of applying steps of Laplacian smoothing to the triangulation, potentially improving the shape quality of the final triangles.

Most importantly, if users cannot specify a priori the point cloud they wish to use for triangle vertices in the final triangulation, they can specify a scalar value for the `TriangulatorInterface` `desired_area()` or an evaluable C++ `FunctionBase` for the triangulator's `desired_area_function()`. In either case, the `libMesh` triangulator will adaptively insert new Steiner points until the user criterion is met. After a `Poly2Tri` triangulation completes, the `libMesh` triangulator loops through the resulting elements, testing each against the user area criterion. Each time a triangle with excessive area is found, the `libMesh` triangulator follows the best practices given in the Delaunay refinement literature: it finds the large element's circumcircle-center point and, if necessary, projects that point to a domain boundary to find the final location for a new Steiner point to insert. The `libMesh` triangulator then deletes all elements with circumcircles containing the Steiner point, and retriangulates the resulting Delaunay cavity, simultaneously reconstructing any outer-boundary segmentation or hole definitions for Steiner points on boundary edges.

Repeating this triangulation-and-refinement sequence until all elements in a triangulation satisfy the user criteria ultimately leads to the creation of the final refined mesh. Fig. 7 shows an example mesh generated using a desired-area function that increases linearly from the bottom-left toward the top-right of the domain.

This triangulation capability is intended for more direct user interaction than that afforded by many `libMesh`-level C++-developer-only APIs. Thus, even in optimized builds, a great deal

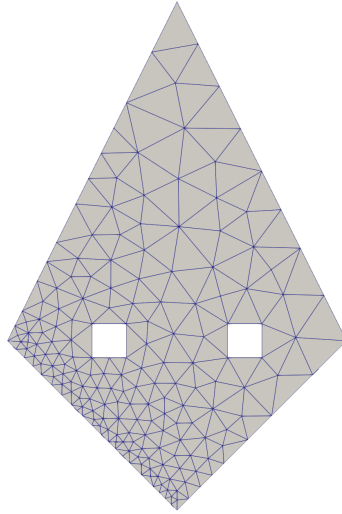


Figure 7: Graded mesh generation: a mesh with holes, generated using a spatially varying area function followed by a smoothing pass.

of error-checking is conducted on API inputs. Points outside the X-Y plane are not directly supported (though meshes in other planes can, of course, be created by applying a transformation to an X-Y mesh after it is triangulated), disconnected outer boundaries are not supported (though disconnected meshes can be constructed by merging the output of multiple connected triangulations), outer boundary definitions that do not form closed loops are not supported, overlapping input points are not supported, intersections between hole boundaries are not supported, and non-positive area criteria are not supported. When retriangulating an existing mesh, inverted elements are not supported. Using either a 1D-embedded-in-2D mesh (based on its Edge elements) or a 2D mesh (based on its domain boundary sides) to define a triangulation domain boundary or a triangulation hole boundary is supported, but use of a mesh featuring a combination of the two is not. In nearly all these cases, the new code always attempts to detect unsupported use cases and return user-friendly error messages appropriate to the unsupported input; in one case (i.e., hole intersections, which can be asymptotically costly to determine), the verification is optional. New unit test coverage includes the most likely error scenarios, not just valid triangulations.

Although the C++ interface to these raw features belonged at the libMesh level, properly

integrating them into MOOSE required that new `MeshGenerator` subclasses be created to expose the input specifications and triangulation options to users. The new `PolyLineMeshGenerator` is general enough to create an arbitrary polyline embedded in 3D, but using it to create a looped polyline in 2D is the easiest way to manually specify an irregular polygon for use as a domain boundary or hole boundary. The new `Poly2TriMeshGenerator` is the MOOSE interface to `libMesh` triangulation, designed to integrate more naturally with the MOOSE `MeshGenerator` system. As with other sophisticated generators, `Poly2TriMeshGenerator` is expected to be able to work within a dependency graph—taking inputs from one or more preceding generators, using common idioms for common generation options, and producing an output that is usable by a subsequent generator. Here, the outer boundary and any hole boundaries are specified solely via meshes from input generators. In addition to adding input parameters for the `libMesh` features discussed above, the MOOSE interface adds the capabilities to specify outer-boundary subsets by name rather than just numerical ID, to set arbitrary subdomain and boundary names on the output triangulation, and to specify a desired area function by using a parsed function string.

Of greater interest is the capability in `Poly2TriMeshGenerator` to combine triangulation of a subdomain surrounding mesh-defined holes with the ability to “stitch” those holes into the final output mesh. For this to work, hole boundary refinement must not be enabled on a hole to be stitched (error checking verifies this), such that the final triangulation vertices and the hole mesh vertices on that boundary will overlap perfectly when stitching is performed. Custom mesh generators can then be used for individual components, thereby enabling the use of quads instead of triangles, anisotropic refinement or otherwise aligned edges, regular subblocks, or other a priori mesh features that are difficult to support via a more general triangulator. But these components can then be passed as hole inputs to the triangulator, which fills in the domain between and surrounding them. Triangulations can be nested indefinitely in this fashion—even, with a triangulation from one generator stitched into a hole in a triangulation from a subsequent generator, to create a mesh on a full domain while preserving complex boundary shapes internal to the domain. Each stage of such a triangulation can be given its own subdomain name, and each internal interface can be given its own boundary name. Fig. 8 shows an example of this from the MOOSE regression tests: an initial quad mesh given one subdomain, surrounded by nested triangle meshes with retained internal square- and diamond-shaped boundaries.

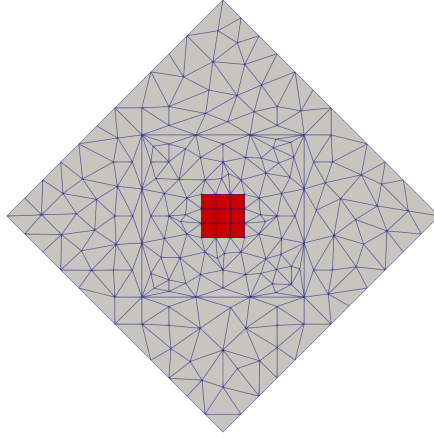


Figure 8: Nested mesh generation: a quad mesh inside a series of triangulated diamonds and squares.

5. Conclusion

This report highlighted three foundational capabilities added in support of NEAMS applications: lower dimensional variables, arbitrary on-the-fly material property evaluation, and native triangulation. Lower dimensional variables could be an important tool in neutronics and fluid flow simulations for describing flux terms. On-the-fly material property evaluation has already been used extensively in Pronghorn for constructing arbitrarily large finite volume stencils, enabling concepts like Rhie-Chow interpolation. The native triangulation capabilities have already been used in MOOSE's Reactor module and may be useful down the road for on-the-fly re-meshing.

REFERENCES

- [1] C. J. Permann, D. R. Gaston, D. Andrš, R. W. Carlsen, F. Kong, A. D. Lindsay, J. M. Miller, J. W. Peterson, A. E. Slaughter, R. H. Stogner, and R. C. Martineau, “MOOSE: Enabling massively parallel multiphysics simulation,” *SoftwareX*, vol. 11, p. 100430, 2020.
- [2] R. L. Williamson, J. D. Hales, S. R. Novascone, G. Pastore, K. A. Gamble, B. W. Spencer, W. Jiang, S. A. Pitts, A. Casagrande, D. Schwen, A. X. Zabriskie, A. Toptan, R. Gardner, C. Matthews, W. Liu, and H. Chen, “Bison: A flexible code for advanced simulation of the performance of multiple nuclear fuel forms,” *Nuclear Technology*, vol. 207, no. 7, pp. 954–980, 2021.
- [3] M. DeHart, F. N. Gleicher, V. Laboure, J. Ortensi, Z. Prince, S. Schunert, and Y. Wang, “Griffin user manual,” Tech. Rep. INL/EXT-19-54247, Idaho National Laboratory, 2020.
- [4] MOOSE Team, “Moose github page.” <https://github.com/idaholab/moose>, 2014.
- [5] B. S. Kirk, J. W. Peterson, R. H. Stogner, and G. F. Carey, “libMesh: a C++ library for parallel adaptive mesh refinement/coarsening simulations,” *Engineering with Computers*, vol. 22, no. 3-4, pp. 237–254, 2006.
- [6] S. Balay, S. Abhyankar, M. F. Adams, S. Benson, J. Brown, P. Brune, K. Buschelman, E. Constantinescu, L. Dalcin, A. Dener, V. Eijkhout, W. D. Gropp, V. Hapla, T. Isaac, P. Jolivet, D. Karpeev, D. Kaushik, M. G. Knepley, F. Kong, S. Kruger, D. A. May, L. C. McInnes, R. T. Mills, L. Mitchell, T. Munson, J. E. Roman, K. Rupp, P. Sanan, J. Sarich, B. F. Smith, S. Zampini, H. Zhang, H. Zhang, and J. Zhang, “PETSc/TAO users manual,” Tech. Rep. ANL-21/39 - Revision 3.17, Argonne National Laboratory, 2022.