INL/RPT-24-78792

Increased accuracy of multiphysics simulations through flexible execution, transient algorithms, and modular physics

June 2024

Nuclear Energy Advanced Modeling and Simulation Milestone, June 2024

Alexander Lindsay[1], Roy Stogner[1], Guillaume Giudicelli[1], Mengnan Li[1], Logan Harbour[1], and Cody Permann[1]

[1]*COMPUTATIONAL FRAMEWORKS*

Idaho National Laboratory

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

INL/RPT-24-78792

Increased accuracy of multiphysics simulations through flexible execution, transient algorithms, and modular physics

Nuclear Energy Advanced Modeling and Simulation Milestone, June 2024

Alexander Lindsay[1], **Roy Stogner**[1], **Guillaume Giudicelli**[1], **Mengnan Li**[1], **Logan Harbour**[1], **and Cody Permann**[1]

[1]**COMPUTATIONAL FRAMEWORKS**

June 2024

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 MOOSE framework is a foundational capability used by the 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 described in this report was conducted in direct support of NEAMS tools and includes: multiple time integrators in the same input file, initial design of framework `Components`, an input file `Application` block, extension of NetGen to 3D geometries in MOOSE, and deployment of executors in the multi-system paradigm. These five additions are fundamental capabilities that will be leveraged by many NEAMS applications.

Page intentionally left blank

CONTENTS

FIGURES

TABLES

ACRONYMS

API	Application Programming Interface
INL	Idaho National Laboratory
MMS	Method of Manufactured Solution
MOOSE	Multiphysics Object-Oriented Simulation Environment
NEAMS	Nuclear Energy Advanced Modeling and Simulation
PETSc	Portable, Extensible Toolkit for Scientific Computation
TA	Technical Area
TH	Thermal Hydraulics

Page intentionally left blank

1. INTRODUCTION

The Nuclear Energy Advanced Modeling and Simulation (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 Reactor Physics TA involves delivering simulation tools to vendors, laboratories, and licensing authorities, whereas the Multiphysics Applications TA entails creating foundational capabilities for the program and exercising the tools 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 this problem, the NEAMS program utilizes the Multiphysics Object-Oriented Simulation Environment (MOOSE) platform [1] from Idaho National Laboratory (INL) 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 added 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 is developed open-source software on GitHub [4] and utilizes the Lesser General Public License v2.1, thus allowing for a large degree of flexibility. It is an active project, with dozens of code modifications being merged to the main branch weekly.

The core of the platform is a pluggable C++ framework that enables scientists and engineers to specify all the details pertaining to 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. Among the more important modules are the solid mechanics, heat transfer, Navier-Stokes, 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 library [5]. libMesh is an open-source software library 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., Portable, Extensible Toolkit for Scientific Computation (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 15 years of MOOSE development, partly due to INL having employed multiple libMesh developers over the years.

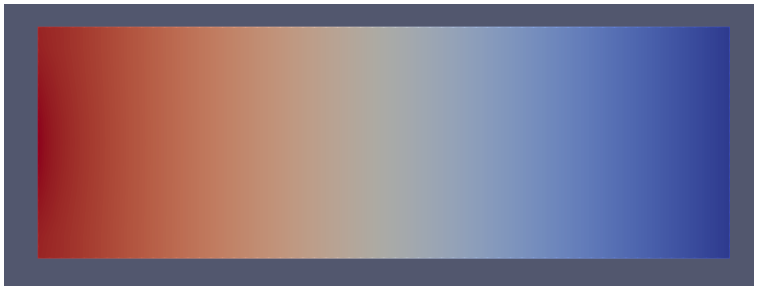
2. Multiple Time Integrators

A user first ran into an issue with monolithic time integration in October 2021. In this case, the issue arose from using Crank-Nicolson time integration for the incompressible Navier-Stokes equations, in which there is no time derivative term for the pressure variable. Without a time derivative term for the pressure, it is easy to satisfy the Crank-Nicolson scheme applied to the momentum equation:

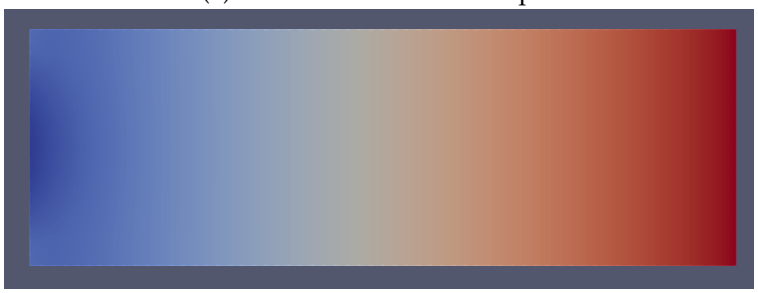
$$\frac{u_{n+1}-u_n}{\Delta t} = \frac{1}{2}\left[F_{n+1}\left(u,\vec{r},t,\nabla u,\nabla^2 u,\nabla p\right) + F_n\left(u,\vec{r},t,\nabla u,\nabla^2 u,\nabla p\right)\right] \quad (1)$$

$$= \frac{1}{2}\left[-\nabla\cdot(\rho u\otimes u-\mu\nabla u)-\nabla p\right]_{n+1} + \frac{1}{2}\left[-\nabla\cdot(\rho u\otimes u-\mu\nabla u)-\nabla p\right]_n \quad (2)$$

by having ∇p switch sign from time step to time step, as shown in 1, which is not physical. This kind of problem can be avoided by applying different time integration schemes to different variables/equations—in this case, by applying something other than Crank-Nicolson to the momentum equation. In addition to its application in the Thermal Hydraulics (TH) area, multiple time integrators are also relevant to fuels and any other areas in which dynamic thermomechanics calculations may be performed. In the thermomechanics case, a user may want to apply Newmark-Beta time integration to the displacement fields, while applying implicit Euler or BDF2 to the energy/temperature equation.



(a) Pressure field at timestep n



(b) Pressure field at timestep $n+1$

Figure 1: Illustration of an oscillating pressure gradient when performing Crank-Nicolson time integration for the Navier-Stokes momentum equation.

To meet this user need, during the last fiscal year we added the ability to apply multiple time integration schemes in a single input. This is made possible by an expansion of subvector capabilities in libMesh, including the addition of the `get_subvector` and `restore_subvector`

Application Programming Interfaces (APIs). The API implementations for the PETSc vector type is as follows:

```
template <typename T>
std::unique_ptr<NumericVector<T>>
PetscVector<T>::get_subvector(const std::vector<numeric_index_type> & rows)
{
  // Construct index set
  WrappedPetsc<IS> parent_is;
  auto ierr = ISCreateGeneral(this->comm().get(),
                              cast_int<PetscInt>(rows.size()),
                              numeric_petsc_cast(rows.data()),
                              PETSC_USE_POINTER,
                              parent_is.get());
  LIBMESH_CHKERR(ierr);

  Vec subvec;
  ierr = VecGetSubVector(_vec, parent_is, &subvec);
  LIBMESH_CHKERR(ierr);

  return std::make_unique<PetscVector<T>>(subvec, this->comm());
}

template <typename T>
void
PetscVector<T>::restore_subvector(NumericVector<T> && subvector,
                                  const std::vector<numeric_index_type> & rows)
{
  auto * const petsc_subvector = cast_ptr<PetscVector<T> *>(&subvector);

  // Construct index set
  WrappedPetsc<IS> parent_is;
  auto ierr = ISCreateGeneral(this->comm().get(),
                              cast_int<PetscInt>(rows.size()),
                              numeric_petsc_cast(rows.data()),
                              PETSC_USE_POINTER,
                              parent_is.get());
  LIBMESH_CHKERR(ierr);

  Vec subvec = petsc_subvector->vec();
  ierr = VecRestoreSubVector(_vec, parent_is, &subvec);
  LIBMESH_CHKERR(ierr);
```

```
}
```

Listing 1: libMesh `get_subvector` and `restore_subvector` implementations

Additionally, the `create_subvector` API was expanded to allow for passing only local vector indices to create the parallel subvector:

```
template <typename T>
void PetscVector<T>::create_subvector(NumericVector<T> & subvector,
                                      const std::vector<numeric_index_type> & rows,
                                      const bool supplying_global_rows) const
{
  parallel_object_only();

  libmesh_error_msg_if(
      subvector.type() == GHOSTED,
      "We do not support scattering parallel information to ghosts for subvectors");

  this->_restore_array();

  // PETSc data structures
  PetscErrorCode ierr = 0;

  // Make sure the passed in subvector is really a PetscVector
  PetscVector<T> * petsc_subvector = cast_ptr<PetscVector<T> *>(&subvector);

  // If the petsc_subvector is already initialized, we assume that the
  // user has already allocated the *correct* amount of space for it.
  // If not, we use the appropriate PETSc routines to initialize it.
  if (!petsc_subvector->initialized())
    {
      libmesh_assert(petsc_subvector->_type == AUTOMATIC || petsc_subvector->_type ==
    PARALLEL);

      if (supplying_global_rows)
        // Initialize the petsc_subvector to have enough space to hold
        // the entries which will be scattered into it.  Note: such an
        // init() function (where we let PETSc decide the number of local
        // entries) is not currently offered by the PetscVector
        // class.  Should we differentiate here between sequential and
        // parallel vector creation based on this->n_processors() ?
        ierr = VecCreateMPI(this->comm().get(),
```

```
                            PETSC_DECIDE,                    // n_local
                            cast_int<PetscInt>(rows.size()), // n_global
                            &(petsc_subvector->_vec));
      else
        ierr = VecCreateMPI(this->comm().get(),
                            cast_int<PetscInt>(rows.size()),
                            PETSC_DETERMINE,
                            &(petsc_subvector->_vec));
      LIBMESH_CHKERR(ierr);

      ierr = VecSetFromOptions (petsc_subvector->_vec);
      LIBMESH_CHKERR(ierr);

      // We created a parallel vector
      petsc_subvector->_type = PARALLEL;

      // Mark the subvector as initialized
      petsc_subvector->_is_initialized = true;
    }
  else
    {
      petsc_subvector->_restore_array();
    }

  std::vector<PetscInt> idx(rows.size());
  if (supplying_global_rows)
    std::iota (idx.begin(), idx.end(), 0);
  else
    {
      PetscInt start;
      ierr = VecGetOwnershipRange(petsc_subvector->_vec, &start, nullptr);
      LIBMESH_CHKERR(ierr);
      std::iota (idx.begin(), idx.end(), start);
    }

  // Construct index sets
  WrappedPetsc<IS> parent_is;
  ierr = ISCreateGeneral(this->comm().get(),
                         cast_int<PetscInt>(rows.size()),
                         numeric_petsc_cast(rows.data()),
                         PETSC_USE_POINTER,
                         parent_is.get());
```

```
  LIBMESH_CHKERR(ierr);

  WrappedPetsc<IS> subvector_is;
  ierr = ISCreateGeneral(this->comm().get(),
                         cast_int<PetscInt>(rows.size()),
                         idx.data(),
                         PETSC_USE_POINTER,
                         subvector_is.get());
  LIBMESH_CHKERR(ierr);

  // Construct the scatter object
  WrappedPetsc<VecScatter> scatter;
  ierr = VecScatterCreate(this->_vec,
                          parent_is,
                          petsc_subvector->_vec,
                          subvector_is,
                          scatter.get()); LIBMESH_CHKERR(ierr);

  // Actually perform the scatter
  VecScatterBeginEnd(this->comm(), scatter, this->_vec, petsc_subvector->_vec, INSERT_VALUES
    , SCATTER_FORWARD);

  petsc_subvector->_is_closed = true;
}
```

Listing 2: libMesh `create_subvector` implementation

The method parameter `supplying_global_rows = false` corresponds to the new local indices capability. These new APIs were leveraged in MOOSE time integrator classes (e.g., `ImplicitEuler`) to act on only the variable degrees of freedom to which the integrator is being applied:

```
void
ImplicitEuler::computeTimeDerivatives()
{
  NumericVector<Number> & u_dot = *_sys.solutionUDot();
  if (!_var_restriction)
  {
    u_dot = *_solution;
    computeTimeDerivativeHelper(u_dot, _solution_old);
  }
```

```
  else
  {
    auto u_dot_sub = u_dot.get_subvector(_local_indices);
    _solution->create_subvector(*_solution_sub, _local_indices, false);
    _solution_old.create_subvector(*_solution_old_sub, _local_indices, false);
    *u_dot_sub = *_solution_sub;
    computeTimeDerivativeHelper(*u_dot_sub, *_solution_old_sub);
    u_dot.restore_subvector(std::move(*u_dot_sub), _local_indices);
    // Scatter info needed for ghosts
    u_dot.close();
  }

  for (const auto i : index_range(_du_dot_du))
    if (integratesVar(i))
      _du_dot_du[i] = 1.0 / _dt;
}

void
ImplicitEuler::postResidual(NumericVector<Number> & residual)
{
  if (!_var_restriction)
  {
    residual += _Re_time;
    residual += _Re_non_time;
    residual.close();
  }
  else
  {
    auto residual_sub = residual.get_subvector(_local_indices);
    auto re_time_sub = _Re_time.get_subvector(_local_indices);
    auto re_non_time_sub = _Re_non_time.get_subvector(_local_indices);
    *residual_sub += *re_time_sub;
    *residual_sub += *re_non_time_sub;
    residual.restore_subvector(std::move(*residual_sub), _local_indices);
    _Re_time.restore_subvector(std::move(*re_time_sub), _local_indices);
    _Re_non_time.restore_subvector(std::move(*re_non_time_sub), _local_indices);
  }
}
```

Listing 3: Use of libMesh subvector APIs within a MOOSE time integrator

The `_local_indices` data member corresponds to the local (i.e., owned by this processor) degrees

of freedom to which the time integrator is being applied. Setting up multiple time integrators in an input file is simple and straightforward. In the below example, we apply `CrankNicolson` time integration to the variable `u`, and `ImplicitEuler` time integration to the variable `v`.

```
[Executioner]
  type = Transient
  solve_type = 'NEWTON'
  dt = 1
  end_time = 3
  [TimeIntegrators]
    [cn]
      type = CrankNicolson
      variables = 'u'
    []
    [ie]
      type = ImplicitEuler
      variables = 'v'
    []
  []
[]
```

Listing 4: Demonstration of multiple time integrators in a MOOSE input

We performed a Method of Manufactured Solution (MMS) study to verify that the multiple time integrators were performing as expected. As shown in Figure 2, we observed the expected first- and second-order convergence rates of the error in the L_2 norm—with respect to timestep size—for Crank-Nicolson and implicit Euler, respectively.

3. Reworking Components for Multiphysics Analyses

MOOSE should be largely customizable by any user possessing basic C++ proficiency, following a 12-hour video tutorial or a 2-day training session. Thus, it is valuable to have a very simplified input syntax in which user-listed objects in the input file can easily be found inside the source directory under the same name. The input file mirrors the object-oriented-programming principles used to organize the code, which can then be modified or duplicated at will to meet the user needs.

But laying bare each part of every equation comes at the price of lengthy input files. Many

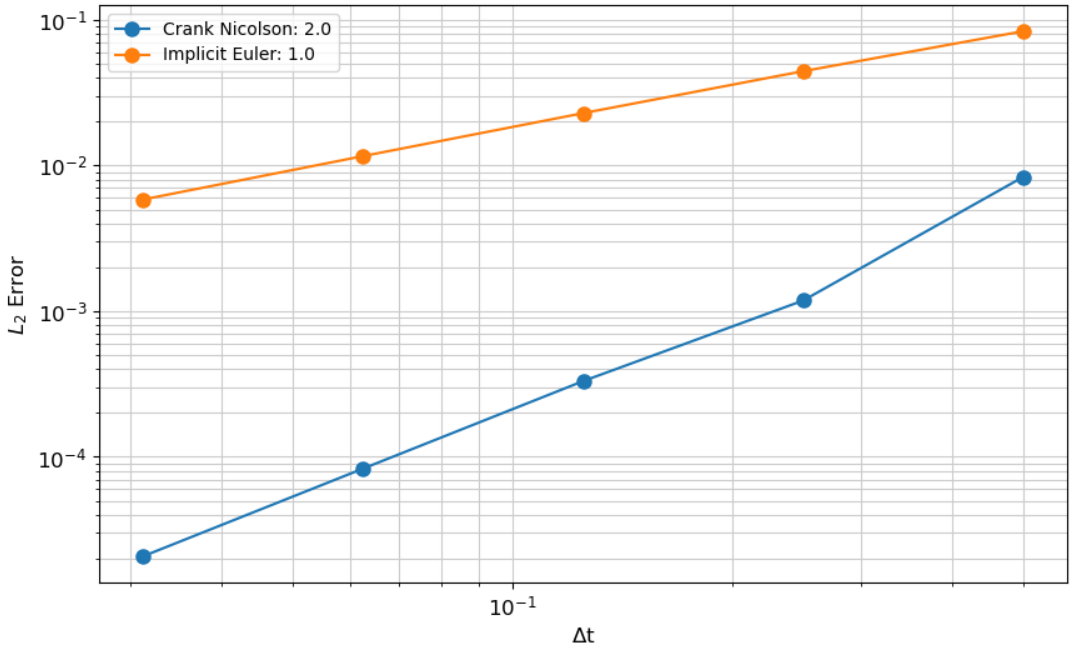


Figure 2: MMS plot of the temporal convergence rates for the implicit Euler and Crank-Nicolson time integration schemes, as applied in the same input file.

inputs on the Virtual Test Bed [7], for example, are several hundreds of lines long. With greater length comes increased potential for error, as numerous parts of the inputs should have parameters that are consistent with each other. Furthermore, large sections of those composable inputs need not be modified for most cases. To remedy such issues stemming from input length, the `Action` system was introduced.

`Actions` coded in the application or the module of interest perform a set of setup operations from a reduced number of parameters, that are consistently forwarded to the objects created by the `Actions`. Most input files created with Griffin, Pronghorn, and Bison use `Actions`—specifically, the `TransportSystems`, `NavierStokesFV`, and `QuasiStaticSolidMechanics` actions, respectively. These classes inherit the `Action` base class. The core of the equations, the kernels, and, in some cases, the boundary conditions—as well as several utilities—are programmatically created with minimal user input.

TH codes such as SAM and RELAP-7, as well as the TH module in MOOSE, are structured differently. The 1D flow equations are pre-implemented in the applications, and the complexity

lies instead in describing the TH loops, their geometry, and their composition. The user input focuses on describing the flow loops. Each element of the flow loop is represented by a `Component`. `Components` then add the relevant equations to the simulation.

This current design has three distinct flaws. The first is that `Components` are not derived from the `Action` class, but from the `MooseObject` class. Thus, they do not inherit the dependency resolution capabilities of `Actions`, nor do they have the same setup order as the rest of MOOSE. This forces the use of a custom `Problem` class (`Problem` handles the setup), preventing fully coupled single-input simulations involving other physics for which a custom `Problem` is used for other reasons. The second flaw is that `Components` are all hard-wired to an equation, meaning that adding a new term or equation requires creating a new component. Even if we only consider flow physics, this approach is unsustainable. The number of phases, the advection of scalars, the discretization method, the turbulence models, the presence of a porous medium—all these model selections introduce a combinatorial need for distinct components. Finally, all the major `Actions` that define equations in the applications implement similar techniques, and this duplication of code increases the maintenance burden and generally prevents programmatic coupling of these equations from being generalized to all equations.

During this fiscal year, we sought to improve on this design. After careful consideration and discussions with several modeling teams, a split between `Components` and `Physics` was decided upon. `Components` will describe the system geometry and the relevant operation quantities, whereas `Physics` will instantiate the equations. If this approach proves successful, complex systems should be described with several `Components`, with the relationship between the `Components` being either automatically detected or specified by the user. Complex multiphysics problems will be indicated by specifying several `Physics`, without explicitly defining the kernels. The interaction between the different `Physics`, either on the same `Component` or on neighbor `Components`, should be automatically handled whenever possible.

3.1 Physics

The first step of this effort entailed creating a common class for `Actions` that define equations. Following consultations, the name `Physics` was chosen for the syntax that will host the equations. The `PhysicsBase` class will serve as the base class. It provides APIs for defining equations, which

are listed in the header below. The signature of each routine is presented, as well as code comments for developers using them.

```
public:
  /// Forwards from the action tasks to the implemented addXYZ() in the derived classes
  /// If you need more than these:
  /// - register your action to the new task using
  ///   registerMooseAction("AppName", ActionClass, "task_name");
  /// - override actOnAdditionalTasks and add your additional work there
  virtual void act() override final;

  /// Routine to add additional setup work on additional registered tasks to a Physics
  virtual void actOnAdditionalTasks() {}

  /**
   * @brief Add new blocks to the Physics
   * @param blocks list of blocks to add to the physics
   */
  void addBlocks(const std::vector<SubdomainName> & blocks);

  /// Return the blocks this physics is defined on
  const std::vector<SubdomainName> & blocks() const { return _blocks; }
  /**
   * @brief Get a Physics from the ActionWarehouse with the requested type and name
   * @param phys_name name of the Physics to retrieve
   * @param allow_fail whether to allow returning a nullptr if the physics does not exist
   */
  template <typename T>
  const T * getCoupledPhysics(const PhysicsName & phys_name, const bool allow_fail = false)
    const;
  /// Get all Physics from the ActionWarehouse with the requested type
  template <typename T>
  const std::vector<T *> getCoupledPhysics(const bool allow_fail = false) const;
  /// Return whether the Physics is solved using a transient
  bool isTransient() const;
  /// Return the maximum dimension of the blocks the Physics is active on
  unsigned int dimension() const;
```

Listing 5: Member functions for the `PhysicsBase` class

The interfaces in `PhysicsBase` will grow with the needs of the derived objects, when those needs are identified to be useful to other `Physics` classes. There are overlapping APIs provided

by PhysicsBase and interfaces in MOOSE created for MooseObject. Future work should consolidate the two by making the interfaces more flexible. Interfaces in MooseObjects such as BlockRestrictable are designed to be initialized at construction, with const (constant) members throughout the entire simulation runs. This is insufficient for Physics, which should interact with Components during the setup phase and may be assigned additional subdomains post-construction.

An additional helper class was created to check user-inputted parameters in a standardized manner. This class is currently templated based on the class it "helps," thus enabling it to be used outside the Physics system. It provides APIs for checking the size of vectors, maps, and enumeration parameters, and for reporting informative errors. The helper class gives greater detail on any identified informative errors than does the typically implemented parameter check in large, equation-defining Actions.

```
/// Check in debug mode that this parameter has been added to the validParams
/// @param param parameter that should be defined
template <typename T>
void assertParamDefined(const std::string & param) const;
/// Check that two parameters are either both set or both not set
/// @param param1 first parameter to check
/// @param param2 second parameter to check
void checkParamsBothSetOrNotSet(const std::string & param1, const std::string & param2)
  const;
/// Check that a parameter is set only if the first one is set to true
/// @param param1 first parameter to check, check the second if true
/// @param param2 second parameter to check, that should be set if first one is true
void checkSecondParamSetOnlyIfFirstOneTrue(const std::string & param1,
                                           const std::string & param2) const;
/// Check that a parameter is set only if the first one is set
/// @param param1 first parameter to check, check the second if set
/// @param param2 second parameter to check, that should be set if first one is set
void checkSecondParamSetOnlyIfFirstOneSet(const std::string & param1,
                                          const std::string & param2) const;
/// Check that the two vector parameters are of the same length
/// @param param1 first vector parameter to compare the size of
/// @param param2 second vector parameter to compare the size of
template <typename T, typename S>
void checkVectorParamsSameLength(const std::string & param1, const std::string & param2)
  const;
```

```
24  /// Check that this vector parameter (with name defined in \p param1) has the same length
      as the MultiMooseEnum (with name defined in \p param2)
25  /// @param param1 vector parameter to compare the size of
26  /// @param param2 multiMooseEnum parameter to compare the size of
27  template <typename T>
28  void checkVectorParamAndMultiMooseEnumLength(const std::string & param1,
29                                               const std::string & param2) const;
30  /// Check that the two-D vectors have exactly the same length in both dimensions
31  /// @param param1 first two-D vector parameter to check the dimensions of
32  /// @param param2 second two-D vector parameter to check the dimensions of
33  template <typename T, typename S>
34  void checkTwoDVectorParamsSameLength(const std::string & param1,
35                                       const std::string & param2) const;
36  /// Check that there is no overlap between the items in each vector parameters
37  /// Each vector parameter should also have unique items
38  /// @param param_vecs vector of parameters that should not overlap with each other
39  template <typename T>
40  void checkVectorParamsNoOverlap(const std::vector<std::string> & param_vecs) const;
41  /// Check that each inner vector of a two-D vector parameter are the same size as another
      one-D vector parameter
42  /// @param param1 two-D vector parameter to check the dimensions of
43  /// @param param2 one-D vector parameter to set the desired size
44  template <typename T, typename S>
45  void checkTwoDVectorParamInnerSameLengthAsOneDVector(const std::string & param1,
46                                                       const std::string & param2) const;
47  /// Check that the size of a two-D vector parameter matches the size of a MultiMooseEnum
      parameter
48  /// @param param1 two-D vector parameter to check the unrolled size of
49  /// @param param2 MultiMooseEnum parameter to set the desired size
50  template <typename T>
51  void checkTwoDVectorParamMultiMooseEnumSameLength(const std::string & param1,
52                                                    const std::string & param2,
53                                                    const bool error_for_param2) const;
54  /// Check that the user did not pass an empty vector
55  /// @param param1 vector parameter that should not be empty
56  template <typename T>
57  void checkVectorParamNotEmpty(const std::string & param1) const;
58  /// Check that two vector parameters are the same length if both are set
59  /// @param param1 first vector parameter to check the size of
60  /// @param param2 second vector parameter to check the size of
61  template <typename T, typename S>
62  void checkVectorParamsSameLengthIfSet(const std::string & param1,
```

```
                                    const std::string & param2,
                                    const bool ignore_empty_default_param2 = false)
  const;
/// Check that a vector parameter is the same length as two others combined
/// @param param1 vector parameter that provides the target size
/// @param param2 vector parameter that provides one term in the combined size
/// @param param3 vector parameter that provides one term in the combined size
template <typename T, typename S, typename U>
void checkVectorParamLengthSameAsCombinedOthers(const std::string & param1,
                                                const std::string & param2,
                                                const std::string & param3) const;

/// Check if the user commited errors during the definition of block-wise parameters
/// @param block_param_name the name of the parameter that provides the groups of blocks
/// @param parameter_names vector of the names of the parameters that are defined on a per
  -block basis
template <typename T>
void checkBlockwiseConsistency(const std::string & block_param_name,
                               const std::vector<std::string> & parameter_names) const;
/// Return whether two parameters are consistent
/// @param other_param InputParameters object from another object to check the 'param_name
  ' parameter in
/// @param param_name the name of the parameter to check for consistency
template <typename T>
bool parameterConsistent(const InputParameters & other_param,
                         const std::string & param_name) const;
/// Emits a warning if two parameters are not equal to each other
/// @param other_param InputParameters object from another object to check the 'param_name
  ' parameter in
/// @param param_name the name of the parameter to check for consistency
template <typename T>
void warnInconsistent(const InputParameters & parameters, const std::string & param_name)
  const;
/// Error messages for parameters that should depend on another parameter
/// @param param1 the parameter has not been set to the desired value (for logging
  purposes)
/// @param value_not_set the desired value (for logging purposes)
/// @param dependent_params all the parameters that should not have been since 'param1'
  was not set to 'value_not_set'
void errorDependentParameter(const std::string & param1,
                             const std::string & value_not_set,
```

```
97                                  const std::vector<std::string> & dependent_params) const;
```

Listing 6: Utility routines for the parameter checking interface using by `Physics`

A number of `Physics` were then created to cover a large panel of needs, ensuring that the design was sound and that the base class provided enough of the necessary APIs to define equations in MOOSE. The effort began with the diffusion equation. A simple `DiffusionPhysicsBase` class served as the base class for several derived classes, each of which implement different discretizations, continuous Galerkin, and finite volume for now, of the diffusion equation, as seen in the inheritance diagram in Fig. 3. These physics can handle the automatic definition of:

- The diffusion Laplacian term, as well as a source term (if requested)
- A time derivative term in a transient problem
- Flux (Neumann) and value (Dirichlet) boundary conditions
- Multi-grid default preconditioning using hypre.

An input file that utilizes the diffusion `Physics` is reproduced in Listing 7. Note the discretization selected in the block nested under the name "`Physics`". The `Functor` system is leveraged so that variables, functions, postprocessors, and functor material properties can all be used to set boundary condition values and fluxes.

```
1  [Mesh]
2    [cmg]
3      type = CartesianMeshGenerator
4      dim = 2
5      dx = '1 2'
6      dy = '2 1'
7      ix = '2 3'
8      iy = '3 2'
9    []
10 []
11
12 [Physics]
13   [Diffusion]
14     [FiniteVolume]
```

```
      [diff]
        source_functor = 2

        # Boundary conditions
        neumann_boundaries = left right top'
        boundary_fluxes = '1 flux_pp flux_function'
        dirichlet_boundaries = 'left
        boundary_values = '2
      []
    []
  []
[]

[Executioner]
  type = Transient
  solve_type = NEWTON
  num_steps = 10
[]
```

Listing 7: Example diffusion simulation using the `Physics` syntax

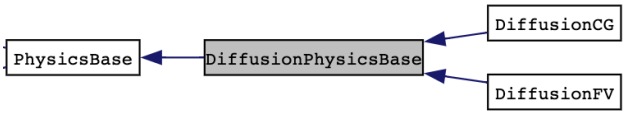


Figure 3: Inheritance diagram for diffusion `Physics`.

After completing the initial example, which can be used to model the diffusion of species in a medium, several classes of heat conduction were defined. The kernels and boundary conditions used in the heat conduction equations were similar, but not sufficiently so to justify an inheritance between `HeatConductionPhysicsBase` and `DiffusionPhysicsBase`. The inheritance structure was modeled similarly to handle both discretizations. The input files and capabilities were similar to the diffusion `Physics`.

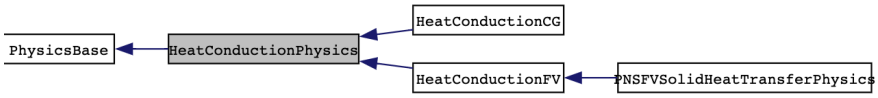


Figure 4: Inheritance diagram for heat conduction `Physics`.

The Physics syntax was then deployed to the Navier-Stokes module, entailing the unique challenge of having to define and couple a high number of equations for:

- The Navier-Stokes flow equations, mass, and momentum
- The conservation of energy in the fluid
- The conservation of energy in the porous media solid phase
- The conservation of advected quantities such as precursors in molten salt reactors
- The turbulence equations, depending on the model chosen.

This presents a significant challenge for creating inter-connected Physics classes, which helps to select a viable design. Previously, these equations were defined using a single Action that was 5,000 lines of code long and instantiated the mass, momentum, fluid energy, and scalar advection equations. The Action was continuously improved, widely utilized, and preferred over a Kernel-based syntax—in the same way that users preferred the solid mechanics Actions over specifying Kernels.

Instead of utilizing a monolithic Action, the problem setup was divided into the following five Physics classes, all of which leverage the finite volume discretization, with weakly compressible flow:

- WCNSFVFlowPhysics for the mass and momentum equations
- WCNSFVFluidHeatTransferPhysics for the fluid energy equation
- PNSFVSolidHeatTransferPhysics for the porous media solid-phase energy equation
- WCNSFVScalarTransportPhysics for the scalar transport equation
- WCNSFVTurbulencePhysics for the turbulence models / equations.

All these Physics classes are capable of accounting for both free and porous media flow, apart from WCNSFVScalarTransportPhysics, which is only implemented for free flow. Deployment of the other four Physics demonstrates how different Physics can be coupled together. The fluid energy equation requires the velocity names and interpolation methods from the flow physics.

The porous media solid phase energy equation must use the same material properties for the heat transfer coefficient as the fluid energy equation. The turbulence `Physics` creates additional diffusion terms in the flow physics, fluid energy, and scalar transport `Physics`. The viscosity in the boundary conditions for the flow equations must include the turbulent contribution. All these interactions are handled by the `Physics` without introducing the potential for erroneous inputs from the user.

The following are additional `Physics` currently in the final stages of the review process before being accepted into the Navier Stokes module:

- A k-ϵ model implemented in the `WCNSFVTurbulencePhysics`
- `WCNSVTwoPhaseMixturePhysics` for the advection of a dispersed phase and the creation of an immobile solid phase.

The simplification afforded by `Physics` deployment is best understood by comparing inputs with the regular kernel-based syntax and this new syntax. A channel input with a drift flux model with the kernel syntax (Listing 8) and with the `Physics` syntax (Listing 9) is reproduced here. Repetition of parameters can be seen in the kernels block in Listing 8. The hundreds of lines afford an equal number of opportunities for mistakes. While the kernels do carry out some parameter checks, they are generally not comparing their parameters to other kernels. The `Physics` set many of the parameters themselves, and include many additional checks between parameters. Several of the functor materials in the kernel syntax provide utilities that can be automated in the `Physics`.

```
mu = 1.0
rho = 10.0
mu_d = 0.1
rho_d = 1.0
l = 2
U = 1
dp = 0.01
inlet_phase_2 = 0.1
advected_interp_method = 'average'
velocity_interp_method = 'rc'

[GlobalParams]
  rhie_chow_user_object = 'rc'
  density_interp_method = 'average'
  mu_interp_method = 'average'
[]

[UserObjects]
  [rc]
```

```
    type = INSFVRhieChowInterpolator
    u = vel_x
    v = vel_y
    pressure = pressure
  []
[]

[Mesh]
  [gen]
    type = GeneratedMeshGenerator
    dim = 2
    xmin = 0
    xmax = '${fparse l * 5}'
    ymin = '${fparse -l / 2}'
    ymax = '${fparse l / 2}'
    nx = 10
    ny = 4
  []
  uniform_refine = 0
[]

[Variables]
  [vel_x]
    type = INSFVVelocityVariable
    initial_condition = 0
  []
  [vel_y]
    type = INSFVVelocityVariable
    initial_condition = 0
  []
  [pressure]
    type = INSFVPressureVariable
  []
  [phase_2]
    type = INSFVScalarFieldVariable
  []
[]

[FVKernels]
  [mass]
    type = INSFVMassAdvection
    variable = pressure
    advected_interp_method = ${advected_interp_method}
    velocity_interp_method = ${velocity_interp_method}
    rho = 'rho_mixture'
  []

  [u_advection]
    type = INSFVMomentumAdvection
    variable = vel_x
    advected_interp_method = ${advected_interp_method}
    velocity_interp_method = ${velocity_interp_method}
    rho = 'rho_mixture'
    momentum_component = 'x'
  []
  [u_drift]
    type = WCNSFV2PMomentumDriftFlux
    variable = vel_x
    rho_d = ${rho_d}
    fd = 'phase_2'
    u_slip = 'vel_slip_x'
    v_slip = 'vel_slip_y'
```

```
    momentum_component = 'x'
  []
  [u_viscosity]
    type = INSFVMomentumDiffusion
    variable = vel_x
    mu = 'mu_mixture'
    limit_interpolation = true
    momentum_component = 'x'
  []
  [u_pressure]
    type = INSFVMomentumPressure
    variable = vel_x
    momentum_component = 'x'
    pressure = pressure
  []
  [u_friction]
    type = PINSFVMomentumFriction
    Darcy_name = Darcy_coefficient_vec
    is_porous_medium = false
    momentum_component = x
    mu = mu_mixture
    rho = rho_mixture
    variable = vel_x
  []

  [v_advection]
    type = INSFVMomentumAdvection
    variable = vel_y
    advected_interp_method = ${advected_interp_method}
    velocity_interp_method = ${velocity_interp_method}
    rho = 'rho_mixture'
    momentum_component = 'y'
  []
  [v_drift]
    type = WCNSFV2PMomentumDriftFlux
    variable = vel_y
    rho_d = ${rho_d}
    fd = 'phase_2'
    u_slip = 'vel_slip_x'
    v_slip = 'vel_slip_y'
    momentum_component = 'x'
  []
  [v_viscosity]
    type = INSFVMomentumDiffusion
    variable = vel_y
    mu = 'mu_mixture'
    limit_interpolation = true
    momentum_component = 'y'
  []
  [v_pressure]
    type = INSFVMomentumPressure
    variable = vel_y
    momentum_component = 'y'
    pressure = pressure
  []
  [v_friction]
    type = PINSFVMomentumFriction
    Darcy_name = Darcy_coefficient_vec
    is_porous_medium = false
    momentum_component = y
    mu = mu_mixture
    rho = rho_mixture
```

```
    variable = vel_y
  []

  [phase_2_advection]
    type = INSFVScalarFieldAdvection
    variable = phase_2
    u_slip = 'vel_slip_x'
    v_slip = 'vel_slip_y'
    velocity_interp_method = ${velocity_interp_method}
    advected_interp_method = 'upwind'
  []
  [phase_2_src]
    type = NSFVMixturePhaseInterface
    variable = phase_2
    phase_coupled = phase_1
    alpha = 0.1
  []
[]

[FVBCs]
  [inlet-u]
    type = INSFVInletVelocityBC
    boundary = 'left'
    variable = vel_x
    functor = '${U}'
  []
  [inlet-v]
    type = INSFVInletVelocityBC
    boundary = 'left'
    variable = vel_y
    functor = '0'
  []
  [walls-u]
    type = INSFVNoSlipWallBC
    boundary = 'top bottom'
    variable = vel_x
    function = 0
  []
  [walls-v]
    type = INSFVNoSlipWallBC
    boundary = 'top bottom'
    variable = vel_y
    function = 0
  []
  [outlet_p]
    type = INSFVOutletPressureBC
    boundary = 'right'
    variable = pressure
    function = '0'
  []
  [inlet_phase_2]
    type = FVDirichletBC
    boundary = 'left'
    variable = phase_2
    value = ${inlet_phase_2}
  []
[]

[AuxVariables]
  [drag_coefficient]
    type = MooseVariableFVReal
  []
```

```
  [rho_mixture_var]
    type = MooseVariableFVReal
  []
  [mu_mixture_var]
    type = MooseVariableFVReal
  []
[]

[AuxKernels]
  [populate_cd]
    type = FunctorAux
    variable = drag_coefficient
    functor = 'Darcy_coefficient'
  []
  [populate_rho_mixture_var]
    type = FunctorAux
    variable = rho_mixture_var
    functor = 'rho_mixture'
  []
  [populate_mu_mixture_var]
    type = FunctorAux
    variable = mu_mixture_var
    functor = 'mu_mixture'
  []
[]

[FunctorMaterials]
  [populate_u_slip]
    type = WCNSFV2PSlipVelocityFunctorMaterial
    slip_velocity_name = 'vel_slip_x'
    momentum_component = 'x'
    u = 'vel_x'
    v = 'vel_y'
    rho = ${rho}
    mu = 'mu_mixture'
    rho_d = ${rho_d}
    particle_diameter = ${dp}
    linear_coef_name = 'Darcy_coefficient'
    outputs = 'out'
    output_properties = 'vel_slip_x'
  []
  [populate_v_slip]
    type = WCNSFV2PSlipVelocityFunctorMaterial
    slip_velocity_name = 'vel_slip_y'
    momentum_component = 'y'
    u = 'vel_x'
    v = 'vel_y'
    rho = ${rho}
    mu = 'mu_mixture'
    rho_d = ${rho_d}
    particle_diameter = ${dp}
    linear_coef_name = 'Darcy_coefficient'
    outputs = 'out'
    output_properties = 'vel_slip_y'
  []
  [compute_phase_1]
    type = ADParsedFunctorMaterial
    property_name = phase_1
    functor_names = 'phase_2'
    expression = '1 - phase_2'
    outputs = 'out'
    output_properties = 'phase_1'
```

```
  []
  [CD]
    type = NSFVDispersePhaseDragFunctorMaterial
    rho = 'rho_mixture'
    mu = mu_mixture
    u = 'vel_x'
    v = 'vel_y'
    particle_diameter = ${dp}
  []
  [mixing_material]
    type = NSFVMixtureFunctorMaterial
    phase_2_names = '${rho} ${mu}'
    phase_1_names = '${rho_d} ${mu_d}'
    prop_names = 'rho_mixture mu_mixture'
    phase_1_fraction = 'phase_2'
  []
[]

[Executioner]
  type = Steady
  solve_type = 'NEWTON'
  nl_rel_tol = 1e-10
[]

[Preconditioning]
  [SMP]
    type = SMP
    full = true
    petsc_options_iname = '-pc_type -pc_factor_shift_type'
    petsc_options_value = 'lu       NONZERO'
  []
[]

[Outputs]
  print_linear_residuals = true
  print_nonlinear_residuals = true
  [out]
    type = Exodus
    hide = 'Re'
  []
  [perf]
    type = PerfGraphOutput
  []
[]

[Postprocessors]
  [Re]
    type = ParsedPostprocessor
    expression = '${rho} * ${l} * ${U}'
  []
[]
```

Listing 8: Full MOOSE syntax.

```
mu = 1.0
rho = 10.0
mu_d = 0.1
rho_d = 1.0
l = 2
```

```
U = 1
dp = 0.01
inlet_phase_2 = 0.1
advected_interp_method = 'average'
velocity_interp_method = 'rc'

# not used
cp = 1
k = 1
cp_d = 1
k_d = 1

[Mesh]
  [gen]
    type = GeneratedMeshGenerator
    dim = 2
    xmin = 0
    xmax = '${fparse l * 5}'
    ymin = '${fparse -l / 2}'
    ymax = '${fparse l / 2}'
    nx = 10
    ny = 4
  []
  uniform_refine = 0
[]

[Physics]
  [NavierStokes]
    [Flow]
      [flow]
        compressibility = 'incompressible'

        density = 'rho_mixture'
        dynamic_viscosity = 'mu_mixture'

        # Initial conditions
        initial_velocity = '0 0 0'
        initial_pressure = 0

        # Boundary conditions
        inlet_boundaries = 'left'
        momentum_inlet_types = 'fixed-velocity'
```

```
        momentum_inlet_functors = '${U} 0'

        wall_boundaries = 'top bottom'
        momentum_wall_types = 'noslip noslip'

        outlet_boundaries = 'right'
        momentum_outlet_types = 'fixed-pressure'
        pressure_functors = '0'

        # Friction is done in drift flux term
        friction_types = "Darcy"
        friction_coeffs = "Darcy_vec"
        standard_friction_formulation = true

        mass_advection_interpolation = '${advected_interp_method}'
        momentum_advection_interpolation = '${advected_interp_method}'
        velocity_interpolation = '${velocity_interp_method}'
        mu_interp_method = 'average'
      []
    []
    [TwoPhaseMixture]
      [mixture]
        liquid_phase_fraction_name = 'phase_1'
        dispersed_phase_fraction_name = 'phase_2'

        # Phase transport equation
        add_phase_transport_equation = true
        alpha_exc = 0.1
        phase_advection_interpolation = 'upwind'
        # see flow for inlet boundaries
        phase_fraction_inlet_type = 'fixed-value'
        phase_fraction_inlet_functors = '${inlet_phase_2}'

        # Drift flux parameters
        add_drift_flux_momentum_terms = true
        density_interp_method = 'average'
        slip_linear_friction_name = 'Darcy'

        # Base phase material properties
        first_phase_density_name = ${rho}
        first_phase_viscosity_name = ${mu}
        first_phase_specific_heat_name = ${cp}
```

```
        first_phase_thermal_conductivity_name = ${k}

        use_dispersed_phase_drag_model = true
        particle_diameter = 0.01

        # Other phase material properties
        other_phase_density_name = ${rho_d}
        other_phase_viscosity_name = ${mu_d}
        other_phase_specific_heat_name = ${cp_d}
        other_phase_thermal_conductivity_name = ${k_d}
        output_all_properties = true
      []
    []
  []
[]

[FunctorMaterials]
  [CD]
    type = NSFVDispersePhaseDragFunctorMaterial
    drag_coef_name = 'Darcy'
    rho = 'rho_mixture'
    mu = mu_mixture
    u = 'vel_x'
    v = 'vel_y'
    particle_diameter = ${dp}
    outputs = 'all'
    output_properties = 'Darcy_coefficient'
  []
[]

[Executioner]
  type = Steady
  solve_type = 'NEWTON'
  nl_rel_tol = 1e-10
[]

[Preconditioning]
  [SMP]
    type = SMP
    full = true
    petsc_options_iname = '-pc_type -pc_factor_shift_type'
    petsc_options_value = 'lu       NONZERO'
```

```
  []
[]

[Outputs]
  print_linear_residuals = true
  print_nonlinear_residuals = true
  dofmap = true
  [out]
    type = Exodus
    hide = 'Re'
  []
  [perf]
    type = PerfGraphOutput
  []
[]

[Postprocessors]
  [Re]
    type = ParsedPostprocessor
    expression = '${rho} * ${l} * ${U}'
  []
[]
```

Listing 9: Physics syntax.

In this work, the syntax and nesting equations under each `Physics` block were extended to the solid mechanics equation, as this provided an opportunity to simultaneously address two separate issues:

- Renaming the Master action to the more physically meaningful name Quasi-Static Physics
- Renaming the Tensor Mechanics module to the more commonly expected name Solid Mechanics.

Combining these two changes in a single pull request avoided drawing out the transition period and requiring users to change the syntax in their input file twice. The deprecated Tensor Mechanics module and Master action remained functional after the transition.

3.2 Components

With `Physics` describing the equations, `Components` is relied on to describe the physical system being modeled. `Components` should be composable so as to form a complex system from its composite parts. Ideally, `Components` should not prescribe a single `Physics`, but instead allow users to select which equations should be solved on which part of the system. `Components` should be able to provide their specificities (e.g., the names of the boundary connections with other `Components`) so as to facilitate automatic setup of `Physics`. The current situation and intended component capabilities are summarized in Listing 10 and Listing 11.

```
[Components]
    [pipe]
      type = PipeFlow1D
      length = 3
      ...
  []
[]
```

Listing 10: Current `Components` syntax: the flow model and discretization are hard-coded. They are known for a given application—such as SAM or RELAP7—that consistently uses the same one in all components.

```
[Components]
    [pipe]
      type = Pipe
      physics = '1D_finite_volume'
      length = 3
      ...
  []
[]
```

Listing 11: Planned `Components` syntax: the flow model and discretization are specified by the user through the selection of a `Physics`. Shared discretization parameters are specified in the `Physics` rather than the `Component`.

Due to the overlap in names between `Components` and the components contained in SAM and the TH module, the `Components` were nested under the `ComponentAction` syntax. Once the system has been sufficiently matured, the syntax can be renamed.

3.2.1 Initial efforts

After creating `Physics` for Navier-Stokes equations, several component designs were explored within the current `Component` system. Namely, the following `Components` were created:

- A `FileMeshPhysicsComponent` that enables physics to be created on a component, thanks to

the utilization of simple utilities. It essentially loads a mesh and adds its blocks to a `Physics` domain. This `Component` may be contributed to MOOSE while the new `Component` design is being created.

- A `FileMeshWCNSFVComponent` that enabled all the weakly compressible finite volume `Physics` to be specified. All parameters of these `Physics` can be specified directly on the component.

The following junctions were also created:

- `StitchedMeshJunction`, which stitches two meshes together
- `FileMeshWCNSFVFlowJunction`, which joins two components that have the `WCNSFV` physics active on them. This allows for continuity of the conservation equations across both components.

The components and junctions were demonstrated on a simple use case involving two pipes, one of which fed into the other. Several options were shown regarding how to connect the two components. Both performed similarly, with the less verbose syntax separating the physics parameters away from the component.

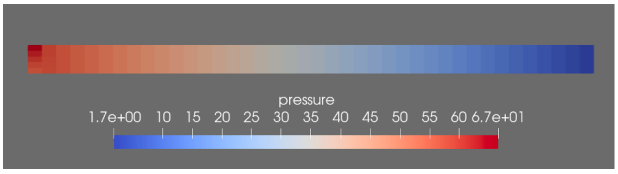


Figure 5: Single `Physics`, stitch mesh junction, flow parameters defined in the `Physics`.

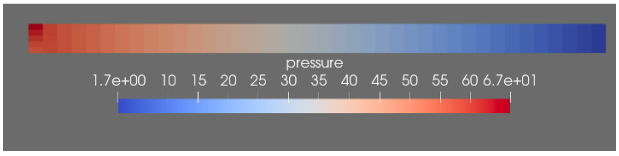


Figure 7: Single `Physics`, stitch mesh junction, parameters defined on each flow component.

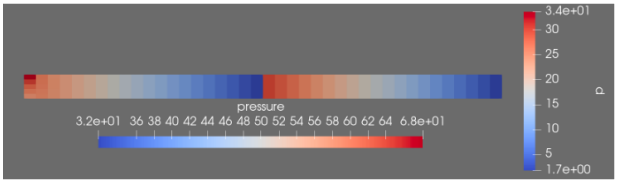


Figure 6: One `Physics` per component, stitch mesh junction, joined with Dirichlet boundaries. This setup is not conservative.

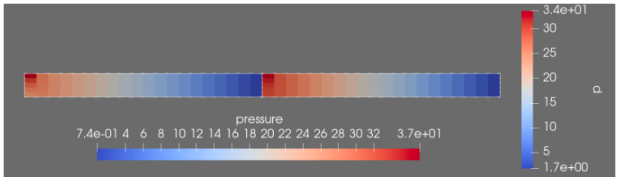


Figure 8: One `Physics` per component, `FileMeshWCNSFVFlowJunction` imposing the average outlet values of one pipe to the inlet and vice versa.

3.2.2 Deployment in the Tritium Migration Analysis Program (TMAP)

Following this initial exploration, along with the deployment of several `Physics` in the Navier-Stokes module, we sought to take an initial step toward solving 0D equations coupled with multi-D equations with `Components` and `Physics` in Tritium Migration Analysis Program (TMAP) v8 [8], a MOOSE-based application that solves for tritium transport in fusion devices. It also solves rate equations and migration/diffusion equations in a number of systems such as the breeder blanket and heat exchangers. Having different equations on different components—fully coupled in a single input file—represents a valuable test of the capabilities of the new approach.

First, we implemented a 0D enclosure component. In TMAP, an enclosure is a part of the system where tritium accumulates and decays. It can be connected to other parts of the system where tritium may be diffusing or advecting. The next component was a 1D structure on which tritium diffuses. Both components only represented the geometry. Multi-dimensional systems can be represented with a `MeshGeneratorComponent`, which uses the mesh created by a mesh generator.

The `Physics` utilized were the `PointSpeciesTrappingPhysics` and `FieldSpeciesTrappingPhysics` for trapping (created in TMAP) and a `SpeciesMigrationPhysics`, inherited from the `DiffusionCG Physics`, for tritium diffusion. The names of the `Physics` have not been defined in consultation with the rest of the TMAP team. They are therefore likely to change in the near future. The code has similarly not been integrated in the main TMAP branch, so modifications may occur.

The distinction between `Point` and `Field` is not in the spirit of keeping `Physics` agnostic from the `Components` they are defined on. However, the code was greatly simplified as the kernels used for a differential equation defined globally and differential equations defined on each node are different.

These `Components` and `Physics` suffice to represent all the verification cases in TMAP8. The input files for a kernel and a `Physics` syntax for verification case 1a are shown in Listings 12 and 13, respectively. Some of the postprocessing in the kernel-based syntax is automatically created by the `Physics`. Furthermore, certain other sections in the inputs are here omitted for the sake of brevity.

```
1 # Verification Problem #1a from TMAP4/TMAP7 V&V document
```

```
# Tritium diffusion through SiC layer with depleting source at 2100 C.
# No Soret effect, solubility, or trapping included.

# Physical Constants
# Note that we do NOT use the same number of digits as in TMAP4/TMAP7.
# This is to be consistent with PhysicalConstant.h
kb = 1.380649e-23 # Boltzmann constant J/K
R = 8.31446261815324 # Gas constant J/mol/K

# Data used in TMAP4/TMAP7 case
length_unit = 1e6 # conversion from meters to microns
temperature = 2373 # K
initial_pressure = 1e6 # Pa
volume_enclosure = '${fparse 5.20e-11*length_unit^3}' # microns^3
surface_area = '${fparse 2.16e-6*length_unit^2}' # microns^2
diffusivity_SiC = '${fparse 1.58e-4*exp(-308000.0/(R*temperature))*length_unit^2}' # microns^2/s
solubility_constant = '${fparse  7.244e22/(temperature * length_unit^3)}' # atoms/microns^3/Pa = atoms*s^2/m^2/kg
slab_thickness = '${fparse  3.30e-5*length_unit}' # microns

# Useful equations/conversions
concentration_to_pressure_conversion_factor = '${fparse kb*temperature*length_unit^3}' # J = Pa*microns^3

[Mesh]
  type = GeneratedMesh
  dim = 1
  nx = 150
  xmax = '${slab_thickness}'
[]

[Variables]
  # concentration in the SiC layer in atoms/microns^3
  [u]
  []
  # pressure of the enclosure in Pa
  [v]
    family = SCALAR
    order = FIRST
    initial_condition = '${fparse initial_pressure}'
  []
```

```
[]

[Kernels]
  [diff]
    type = MatDiffusion
    variable = u
    diffusivity = '${diffusivity_SiC}'
  []
  [time]
    type = TimeDerivative
    variable = u
  []
[]

[ScalarKernels]
  [time]
    type = ODETimeDerivative
    variable = v
  []
  [flux_sink]
    type = EnclosureSinkScalarKernel
    variable = v
    flux = scaled_flux_surface_left
    surface_area = '${surface_area}'
    volume = '${volume_enclosure}'
    concentration_to_pressure_conversion_factor = '${
    concentration_to_pressure_conversion_factor}'
  []
[]

[BCs]
  # The concentration on the outer boundary of the SiC layer is kept at 0
  [right]
    type = DirichletBC
    value = 0
    variable = u
    boundary = 'right'
  []
  # The surface of the slab in contact with the source is assumed to be in equilibrium with
    the source enclosure
  [left]
    type = EquilibriumBC
```

```
    variable = u
    enclosure_scalar_var = v
    boundary = 'left'
    Ko = '${solubility_constant}'
    temp = ${temperature}
  []
[]

[Postprocessors]
  # flux of tritium through the outer SiC surface - compare to TMAP7
  [flux_surface_right]
    type = SideDiffusiveFluxIntegral
    variable = u
    diffusivity = '${diffusivity_SiC}'
    boundary = 'right'
    execute_on = 'initial nonlinear linear timestep_end'
    outputs = 'console csv exodus'
  []
  # flux of tritium through the surface of SiC layer in contact with enclosure
  [flux_surface_left]
    type = SideDiffusiveFluxIntegral
    variable = u
    diffusivity = '${diffusivity_SiC}'
    boundary = 'left'
    execute_on = 'initial nonlinear linear timestep_end'
    outputs = ''
  []
  # scale the flux to get inward direction
  [scaled_flux_surface_left]
    type = ScalePostprocessor
    scaling_factor = -1
    value = flux_surface_left
    execute_on = 'initial nonlinear linear timestep_end'
    outputs = 'console csv exodus'
  []
[]
```

Listing 12: TMAP input with traditional MOOSE syntax.

```
# Verification Problem #1a from TMAP4/TMAP7 V&V document
# Tritium diffusion through SiC layer with depleting source at 2100 C.
# No Soret effect, solubility, or trapping included.
```

```
4
5  # Physical Constants
6  # Note that we do NOT use the same number of digits as in TMAP4/TMAP7.
7  # This is to be consistent with PhysicalConstant.h
8  kb = 1.380649e-23 # Boltzmann constant J/K
9  R = 8.31446261815324 # Gas constant J/mol/K
10
11 # Data used in TMAP4/TMAP7 case
12 length_unit = 1e6 # conversion from meters to microns
13 temperature = 2373 # K
14 initial_pressure = 1e6 # Pa
15 volume_enclosure = '${fparse 5.20e-11*length_unit^3}' # microns^3
16 surface_area = '${fparse 2.16e-6*length_unit^2}' # microns^2
17 diffusivity_SiC = '${fparse 1.58e-4*exp(-308000.0/(R*temperature))*length_unit^2}' # microns
       ^2/s
18 solubility_constant = '${fparse  7.244e22/(temperature * length_unit^3)}' # atoms/microns^3/
       Pa = atoms*s^2/m^2/kg
19 slab_thickness = '${fparse  3.30e-5*length_unit}' # microns
20
21 # Useful equations/conversions
22 concentration_to_pressure_conversion_factor = '${fparse kb*temperature*length_unit^3}' # J =
        Pa*microns^3
23 pressure_unit = 1 # number of pressure units in a Pascal
24
25 [Physics]
26   [TMAP8]
27     [SpeciesTrapping]
28       [0d_trapping]
29         species = 'v'
30         equilibrium_constants = ${solubility_constant}
31
32         # These parameters can be passed for each component here in lieu of fetching them
       from the components
33         # initial_values = '${initial_pressure}'
34         # temperatures = ${temperature}
35
36         verbose = true
37
38         # If the initial pressure had not been scaled (=1 right now)
39         pressure_unit_scaling = ${pressure_unit}
40         # Volume and area have been pre-scaled
41         length_unit_scaling = 1
```

```
      []
    []
  []
  [Diffusion]
    [ContinuousGalerkin]
      [multi-D]
        variable_name = 'u'
        diffusivity_matprop = ${diffusivity_SiC}

        # To help coupling to trapping
        compute_diffusive_fluxes_on = 'structure_left'

        dirichlet_boundaries = 'structure_right'
        boundary_values = '0'
      []
    []
  []
[]

[SystemComponents]
  [structure]
    type = Structure1D
    species = 'u'
    diffusivities = 'D_u'
    physics = 'multi-D'

    # Geometry
    nx = 150
    xmax = ${slab_thickness}
    length_unit_scaling = 1
  []

  [enc]
    type = Enclosure0D
    species = 'v'
    physics = '0d_trapping'

    # Conditions
    temperature = ${temperature}
    species_initial_pressures = '${initial_pressure}'

    # Geometry
```

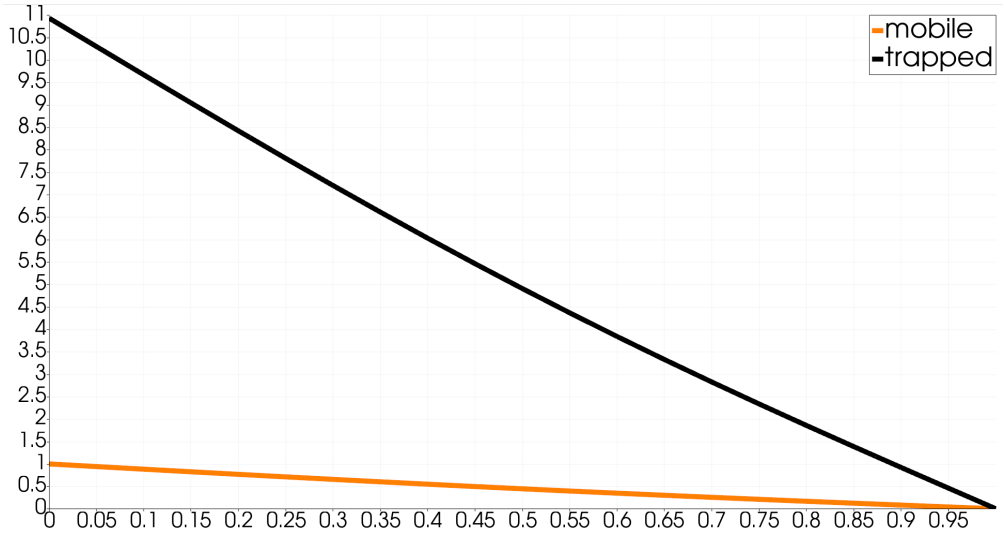


Figure 9: Verification case 1d: diffusion of a mobile species with volumetric trapping. Arbitrary units

```
    surface_area = 2.16e-6
    volume = 5.2e-11

    # Connection to structures
    connected_structure = 'structure'
    boundary = 'structure_left'
  []
[]
```

Listing 13: TMAP input with `Physics/Component` syntax. The names used for the syntax are not final

Verification cases 1b, 1c, 1d, 1e, 1fa-d, 1g, 1gc and validation case 2a were all implemented with those `Physics` and `Components`. Selected results are presented in Figures 9 and 10. The results are identical to those obtained with the previous kernel-based syntax.

3.2.3 Future efforts

Follow-on efforts will seek to deploy the new `Components` to the TH module. This will serve to demonstrate the following:

- Junctions between components. Flow components can live on disjoint meshes (e.g., when modeling arrays of flow channels that all join in a plenum).

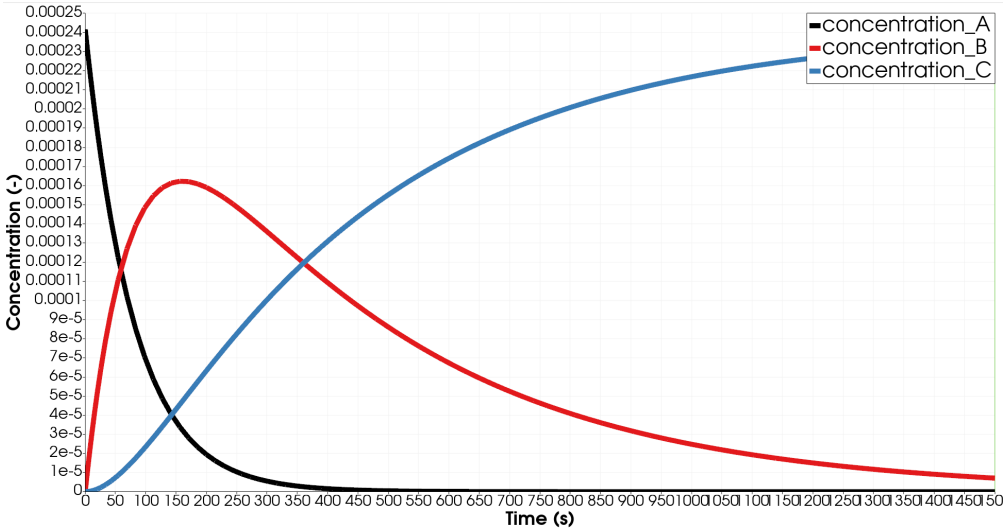


Figure 10: Verification case 1g-c: time evolution of a three species diffusion reaction problem. Species 'a' and 'b' react to form species 'ab'

- Coupling different flow discretizations together into a single input in a fully coupled and conservative manner. Alternative setups—notably using domain-overlapping coupling—can also be deployed using the `Components` syntax.

4. Specifying Application Information in the Input

A new application system with a new syntax block called the `Application` block was introduced into MOOSE to explicitly specify the application type used for the input file. Previously, there was no information in MOOSE or MOOSE-based application input files that helped to distinguish between different applications, leading to potential confusion for users. For instance, in the Virtual Test Bed [7], many reactor examples use input files from various codes, but apart from the comments line, there is no way for users to mark the application type for each input file. The `Application` block now enables users to differentiate those input files by explicitly providing the application type. Furthermore, the `Application` block is parsed prior to constructing the MOOSE application, thus enabling early detection of mismatches between the registered and the specified application types and thereby preventing incorrect simulation setups. For simulations using MultiApps for physics coupling, the child application type listed in the parent app input will be checked against the application type in the child-app input file so as to ensure application interoperability. When combined applications are employed (e.g.,

BlueCRAB), users can opt to run only one of the dependent applications by specifying its type in the `Application` block. This is essential when different applications in a single multiphysics simulation feature competing syntax. This capability used to be performed with the command line option `--app <AppName>`. The `Application` block provides an alternative way to specify the application type. Placing the command line option in the input file improves the reproducibility of complex multiphysics simulations.

Model developers can simply add the `Application` block to the input file as the snippet below demonstrates. It is also compatible with previous input files in which no application type is specified. A default application type would be used in that case.

```
[Application]
  type = MooseTestApp
[]
```

During the application construction phase, the application system will check the whether the application type is consistent with the user-specified one. If not, the system will throw out a run-time error before running the simulation:

```
*** ERROR ***
'DummyApp' is not a registered application name.
```

An example with a `MultiApp` is provided to demonstrate the cross-checking capability with `Application` block. In the parent application input file, a user can specify the application type of a child app is set by using the `MultiApps/*/app_type` parameter, as shown below:

```
[MultiApps]
   [child]
     type = TransientMultiApp
     app_type = PronghornApp
     execute_on = timestep_end
     positions = '1 1 0'
     input_files = application_block_child.i
     output_in_position = true
```

```
    move_positions = '2 2 0'
  []
[]
```

In the child application input file, the application type can also be explicitly specified by using `Application/type` as follows. During the application construction phase, the application system will cross check the application type given in the parent application `MultiApp` block, with the application type stated in the child application `Application` block to ensure interoperability.

```
[Application]
  type = PronghornApp
[]

[Mesh]
  type = GeneratedMesh
  dim = 2
  nx = 10
  ny = 10
[]
```

If the application type is inconsistent within the parent app and child app input files, the following run-time error will be returned to remind users to double check the specified application type:

```
*** ERROR ***
The following error occurred in the MultiApp 'child' of type
   TransientMultiApp.

In the 'child0', 'DummyApp' is not a registered application. The
   registered application is named: 'PronghornApp'. Please double check
   the \texttt{Application} block to make sure the correct application is
   provided.
```

The current implementation allows for setting the application type via the command-line

option in the parent App (e.g., `Application/type=DummyApp`), but does not support direct setup of the child app type via the command-line (e.g., `child0:Application/type=DummyApp`). For future work, we will consider implementing this capability and expanding the `Application` block to include more options, in accordance with user feedback regarding this new application system.

5. Netgen Integration for Tetrahedral Mesh Generation

A previous MOOSE usability improvement, the `XYDelaunay` mesh generator, proved more popular than expected among MOOSE users, especially when used as a subgenerator within other custom mesh generators (e.g., those in the MOOSE Reactor module). Although MOOSE has always been able to compute on triangular meshes read from mesh files or generated on simple domains, `XYDelaunay` added the capability to generate triangular mesh components based on input boundaries and with optional input holes derived from other mesh generators. This workflow allowed many users to both metaphorically and literally fill in the gaps between other types of mesh generation. A typical mesh generation workflow might begin with generators designed to model a single circular layered pin, followed by generators that align one or more pin types into a larger assembly, but then rely on `XYDelaunay` to mesh the "negative space" in between the pins or subassemblies. Use of a general triangulation algorithm enables arbitrary domain shapes to be specified while still retaining control over the element shape quality and size—a significant advance over more structured "filler" options that would require unnaturally sized or stretched elements when faced with domains whose interior thicknesses varied significantly from place to place.

However, a significant drawback to the `XYDelaunay` generator is the "XY" restriction, which limits it to triangulation within boundaries defined in the XY plane. And though the output triangulation can be subsequently extruded into layers of prisms, translated in Z, rotated in a 3D fashion, and/or otherwise modified by a bijective map, the ultimate end product will still be either a triangulation or a sum of extrusions of triangulations. Although this is surprisingly useful for modeling reactor core assemblies designed around fuel pins, such a tool cannot be truly general without possessing the ability to mesh an arbitrary 3D domain—not just a 3D extrusion of a 2D domain.

An optional libMesh feature, integration with Tetgen, has proven valuable for other users with similar problems; however, this option is incompatible with the binary distribution of most MOOSE codes: Tetgen is only available under the AGPLv3 license, unless a commercial license is purchased. AGPLv3 is incompatible with non-open-source applications (e.g., the controlled applications in the MOOSE herd), and a closed-source license would require additional funding, thus proving too limiting for the redistribution involved in collaborations with MOOSE users in academia.

QHull, another tetrahedralization code currently offered through libMesh, has compatible licensing but does not handle non-convex point sets, as would be required in order for a MOOSE tetrahedralization mesh generator to be useful in the same sense as the MOOSE triangularization.

A quick survey of other tetrahedralization codes suggested that similar licensing issues applied to CGALmesh, DelPSC, Gmsh, GridEx, GRUMMP, LBIE, Mefisto, MeshKit, QualMesh, SolidMesh, T3D, and, to a slightly lesser extent, MMG. Other technical issues would make it too difficult to build a general tetrahedralizer around DistMesh, GTS, iso2mesh, QMG, Stellar, and SUS. And some lesser-known tetrahedralization codes were avoided due to lack of active development. Even MeshGenC++, a semi-active code from 2021, was discovered to have experienced a degree of "bit rot" (issues with newer compilers, though surmountable, were nonetheless worrisome).

Of the remaining candidates (i.e., LaGrit, Netgen, and OmegaH), we chose to investigate Netgen, due to its active user community, relatively simple tetrahedralization API, and past interest from other MOOSE developers and collaborators. Our only concern was that some other developers have had difficulties integrating Netgen with other build systems.

In our new Netgen-enabled libMesh, a tagged Netgen version from their version control repository was imported as a libMesh submodule. Software integration was indeed a difficulty, but any issues encountered with Netgen's CMake-based configuration were worked around in the libMesh autotools-based configuration. The Netgen CMake configuration was chosen so as to enable tetrahedralization yet disable many of the broader Netgen features that would otherwise introduce additional unused dependency libraries into the libMesh+MOOSE build. The additional dependencies required by Netgen were thus reduced to CMake for builds, plus an rpath-editing tool (operating system dependent) for installs.

As with `XYDelaunay`, the core of the MOOSE mesh generation capability was enabled by refactoring and adding features and a new backend to an older libMesh compatibility. In this case, the old libMesh Tetgen interface was converted into a `MeshTetInterface` abstract base class, with the accessors for controlling mesh generation parameters, a pure virtual "triangulate()" method for invoking the mesh generation, and a number of generalized tool subroutines useful for implementation-independent preparatory work. These include a converter to turn volumetric input meshes into surface meshes on the domain boundary to remesh, and a method to test the integrity of the boundary prior to the meshing (to provide users with more specific and more helpful error messages in the event such inputs are detected). Other non-meshing-specific tools created for this use case were a mesh volume calculator (for evaluating the results of unit tests), a mesh modifier to split hexes into tetrahedra (for creating inputs to unit tests, as well as for future use when stitching creations from hex-based workflows to tetrahedralizations), and a simple flood-fill algorithm (for separating input manifolds into their connected components).

As in `XYDelaunay`, the new tetrahedralization interface is designed to take inputs in one of two formats. A boundary may be specified directly via a mesh of lower-dimensional elements: edge elements for the boundary of a 2D domain, or tri elements for the boundary of a 3D domain. Otherwise, a boundary may be specified indirectly via the unpaired sides of a mesh of higher-dimensional elements—tri or quad elements in 2D; volume `Cell` elements in 3D. The 3D indirect case supports tet elements, possibly mixed with prism elements, pyramid elements (which only expose triangular sides to the mesh outer boundary) and/or hex elements, which do not intersect the mesh outer boundary. Direct boundary specification is generally more useful for receiving input from mesh generators that directly construct a boundary from a parametric representation. Indirect boundary specification is useful for remeshing an already-meshed domain, but is most useful for specifying the boundary of a "hole" in a triangulation or tetrahedralization in cases where the specification mesh is then to be retained and "stitched" into that hole. As in the 2D case, the new 3D code supports arbitrary numbers of holes within the triangulated mesh.

Unit tests for all of the above capabilities exist at the libMesh level, but the higher-level interface at the MOOSE level is still being written. Some delay was incurred in investigating the inner workings of Netgen, after it was discovered that Netgen was failing to produce certain types of unrefined Delaunay tetrahedralizations in an expected fashion.

In general, the Delaunay problem, which entails generating an optimal simplicial mesh to discretize a domain defined by a set of boundary faces and a set of boundary and internal vertices, is much harder in 3D than in 2D, so we went into this project expecting some moderately suboptimal results. In 2D, any triangulation that is "locally Delaunay" (meaning that no vertex not belonging to a triangle lies within the circumcircle of that triangle) is also "globally Delaunay" (maximizing the size of the smallest angles within the mesh, which for isotropic physics tends to produce the best finite element method solutions), thus such a mesh can be refined by a greedy descent algorithm, that simply performs "edge swap" operations whenever a pair of triangles violating the local Delaunay condition are found. In 3D, however, not only are such local operations more complicated (three tetrahedra may be turned into two, or vice-versa), they are not guaranteed on their own to produce a globally optimal mesh. This is part of the reason for our desire to rely more heavily on a tested third-party meshing library, as opposed to the code in the `XYDelaunay` case, which only relies on a library (Poly2Tri) for the coarsest discretizations and then performs custom refinement afterward. Even still, we did not expect perfectly optimal elements in the third-party output. What was most surprising, however, was that the tetrahedral meshing algorithm used in Netgen does not give us the expected vertices in its output. Even with all mesh optimization and smoothing turned off, it generates additional vertices, because it relies on an advancing-front algorithm—rather than a pure Delaunay algorithm—to generate its initial unoptimized mesh. This does not negatively affect the utility of the generated mesh for the above-described uses, since users of tetrahedral meshes for infill and stitching naturally expect to need additional interior vertices for mesh quality purposes. However, during the development process, we attracted attention from other users seeking a solution for generating meshes to use with the Pebble Tracking Transport algorithm, and it was disheartening to discover that Netgen would not be a viable backend for any method in which new vertex insertion is unacceptable. Compare the cutaway views of the meshes in Figure 11 and Figure 12 to see an example of the problem. In the former, a tetrahedralization of an octahedron embedded in a cube, the generated mesh is asymmetric, as expected, but even this asymmetry did not trigger any additional vertices from Netgen. In the latter, a tetrahedralization of a shell between two faceted spheres, most of the domain is given the desired coarse discretization, but at one point an extra vertex is inserted unnecessarily, as are a swath of unrequested refined elements around it. Any

future fix to this must begin by adding a third backend option to the Tetgen and Netgen options already available—a backend that uses a pure Delaunay algorithm rather than beginning with an advancing front method.

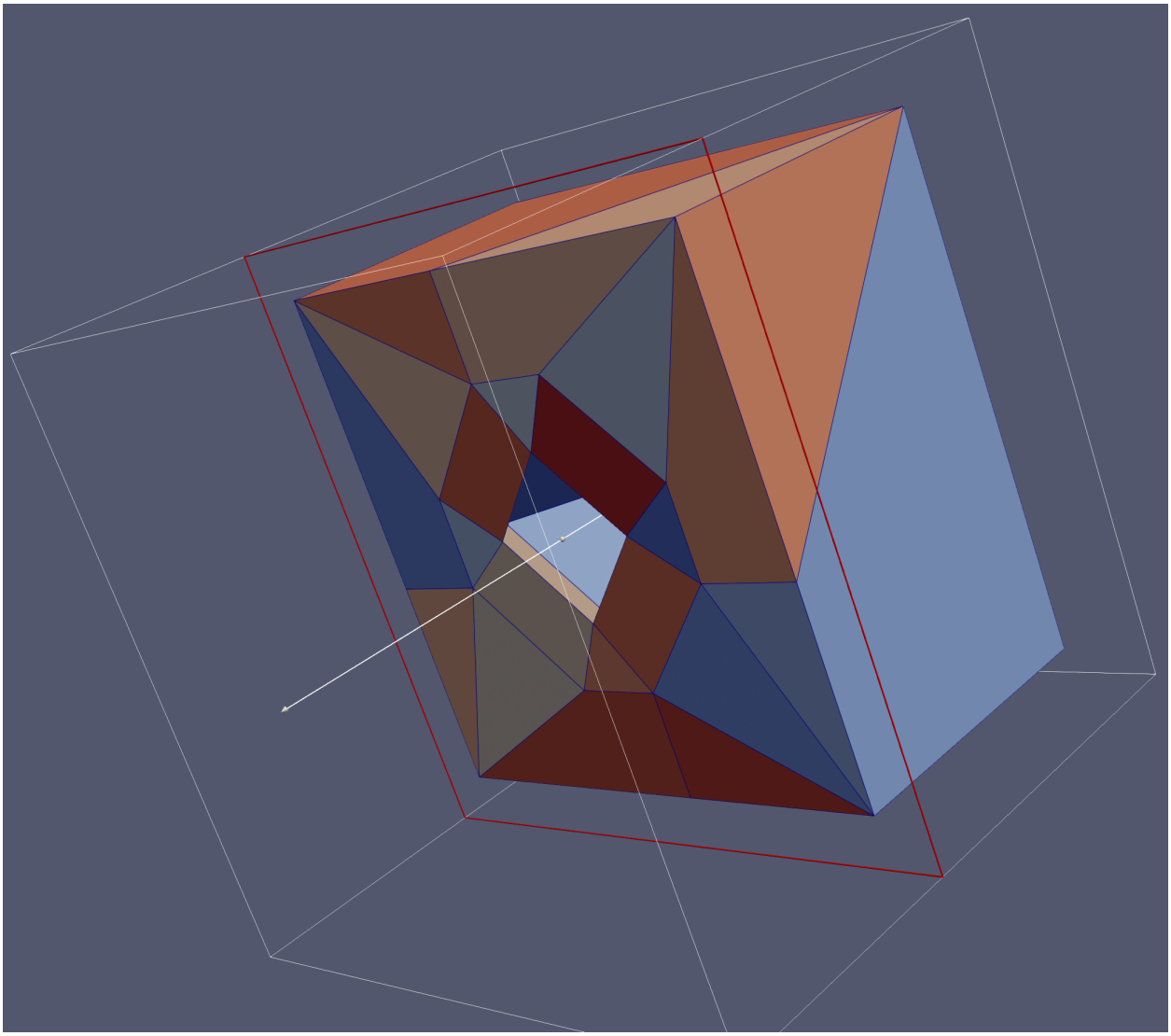


Figure 11: Unit test showing a cutaway of a coarse tetrahedralization.

6. Executor System Extension

The `Executor` system was originally developed to be a complete replacement for the `Executioner` system in MOOSE. It behaves similarly to the `MeshGenerator` system in terms of user interaction, with the execution ordering being defined in input which builds an underlying dependency graph. This flexibility is the ideal path forward to a general execution system for

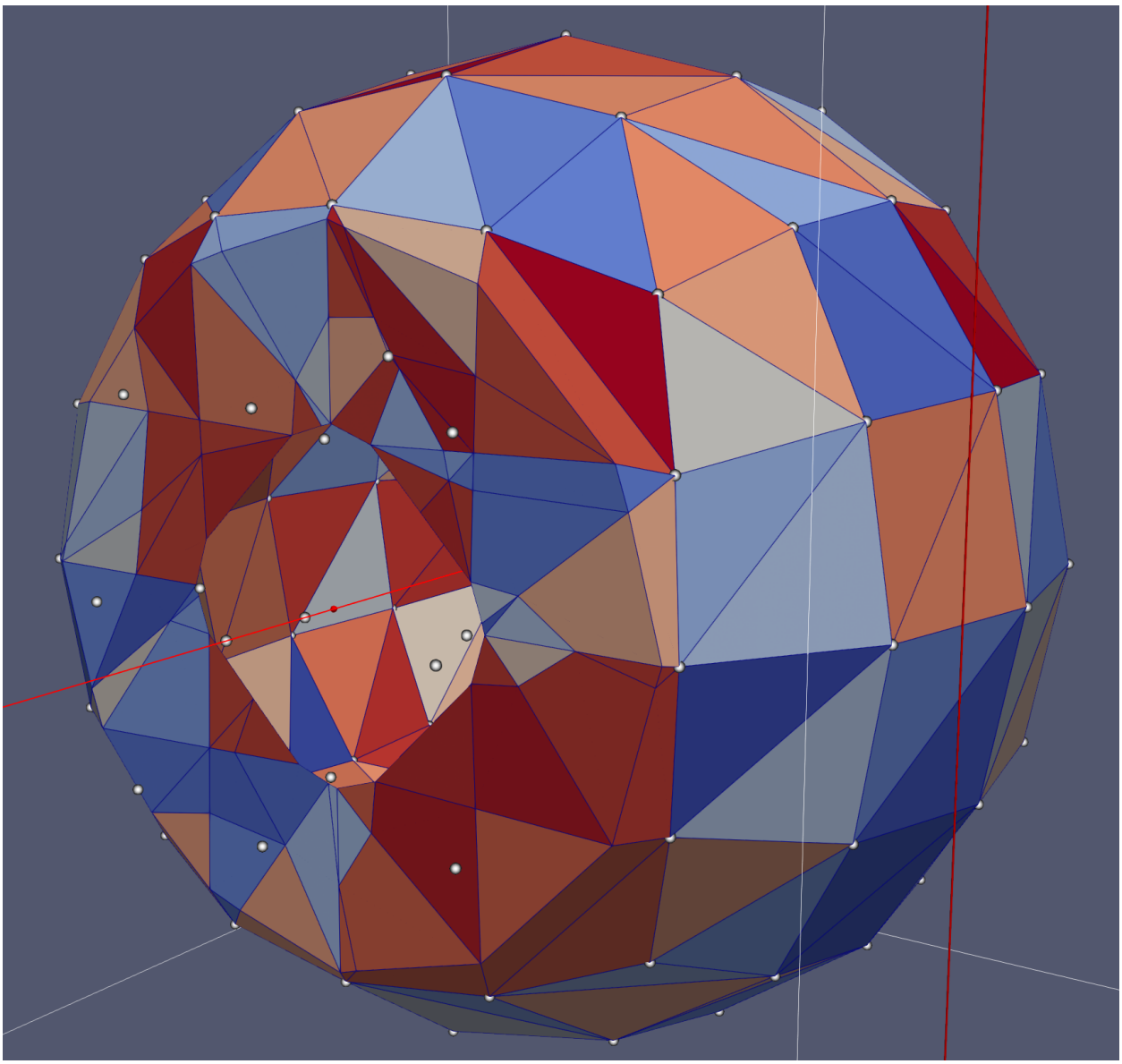


Figure 12: Unit test showing a cutaway featuring an unnecessary extra interior vertex in the upper right.

MOOSE that is extendable and will remove the need for littering the framework with custom execution steps. Additionally, it enables input-based options for custom execution schemes within MOOSE. While the `Executor` system is very flexible, it cannot currently be used to execute a solve.

The `Executioner` system is used to control the primary execution of the system, notably the solve parameters and optionally the time state of the problem (i.e., a transient or steady state). The system was eventually extended to support nested solves, including fixed point solves used for tightly coupled multi-physics simulations with multiapps. Despite these improvements, the `Executioner` system is rigid and lacking in extensiblity. The nesting of solves exists as a single nested solve object in code only, not easily produced by a separate object in input. Because of this, the propagation of parameters within these nested solve objects is incredibly unclear as they all must be defined in the parameters for the single `Executioner`.

The introduction of multi-system support, which enables the solve of independent systems within a single solve, increased the need for a more flexible execution system. Such independent systems often require different solve techniques. Additionally, iteration between these independent systems is desired and the definition of such an iteration scheme is often physics specific. For example, the current `Executioner` for the optimized linear system assembly within the `navier_stokes` module is incredibly non-general due to the lack of pluggability in the `Executioner` system.

With these deficiencies came the expansion of the `Executor` system as the underlying execution system within MOOSE. The `Executioner` still serves as the top level execution object. The inner solves that occur within the `Executioner` are then done via the `Executor` system. This design was chosen in order to preserve support for the current input syntax.

For example, take the current execution syntax for a basic transient solve:

```
[Executioner]
  type = Transient
  solve_type = PJFNK
  num_steps = 2
[]
```

Unseen to the user, the above will setup a single `SolveExecutor` object that does the underlying solve on each timestep. The `Executioner` objects now contain a `executors` parameter, which are a list of `Executor` objects to be solved in the given order. That is, the following input will be equivalent to the input above:

```
[Executors/solve]
  type = SolveExecutor
  solve_type = PJFNK
[]

[Executioner]
  type = Transient
  executors = 'solve'
[]
```

This input is indeed more verbose, but enables input-driven custom solves while still supporting the previous input syntax.

A more interesting example would be an example with multisystem capability. Take the following example, which is only a snippet, but includes a system for the variable u and a system for the variable v, solved in that order.

```
[Variables]
  [u]
    solver_sys = u
    block = 0
  []
  [v]
    solver_sys = v
    block = 1
  []
[]

[Executors]
  [solve_u]
```

```
    type = SolveExecutor
    solve_type = PFJNK
    sys = u
  []
  [solve_v]
    type = SolveExecutor
    solve_type = NEWTON
    sys = v
  []
[]

[Executioner]
  type = Steady
  executors = 'solve_u solve_v'
[]
```

Before the expanded `Executor` capability, the above solve would have required a custom `Executioner` object. Additionally, it would have been very difficult as a developer to maintain different solve options for each system due to the way that parameters are added to the `Executioner` objects.

7. CONCLUSION

This report highlighted five foundational capabilities added in support of NEAMS applications: multiple time integrators in the same input file, the initial design of framework components classes, an input file `Application` block, extension of NetGen to 3D geometries in MOOSE, and deployment of executors in the multi-system paradigm. These additions are expected to be heavily leveraged by MOOSE-derived NEAMS applications so as to meet the ever-growing need for higher-fidelity multiphysics modeling and simulation capabilities.

REFERENCES

[1] G. Giudicelli, A. Lindsay, L. Harbour, C. Icenhour, M. Li, J. E. Hansel, P. German, P. Behne, O. Marin, R. H. Stogner, J. M. Miller, D. Schwen, Y. Wang, L. Munday, S. Schunert, B. W. Spencer, D. Yushu, A. Recuero, Z. M. Prince, M. Nezdyur, T. Hu, Y. Miao, Y. S. Jung, C. Matthews, A. Novak, B. Langley, T. Truster, N. Nobre, B. Alger, D. Andrs, F. Kong, R. Carlsen, A. E. Slaughter, J. W. Peterson, D. Gaston, and C. Permann, "3.0 - moose: Enabling massively parallel multiphysics simulations," *SoftwareX*, vol. 26, p. 101690, 2024.

[2] R. L. Williamson, J. D. Hales, S. R. Novascone, G. Pastore, K. A. Gamble, B. W. Spencer, W. Jiang, S. A. Pitts, A. Casagranda, 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 repository." `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.

[7] G. L. Giudicelli, A. Abou-Jaoude, A. J. Novak, A. Abdelhameed, P. Balestra, L. Charlot, J. Fang, B. Feng, T. Folk, R. Freile, T. Freyman, D. Gaston, L. Harbour, T. Hua, W. Jiang, N. Martin, Y. Miao, J. Miller, I. Naupa, D. O'Grady, D. Reger, E. Shemon, N. Stauff, M. Tano, S. Terlizzi,

S. Walker, and C. Permann, "The virtual test bed (vtb) repository: A library of reference reactor models using neams tools," *Nuclear Science and Engineering*, vol. 0, no. 0, pp. 1–17, 2023.

[8] P.-C. A. Simon, P. W. Humrickhouse, and A. D. Lindsay, "Tritium transport modeling at the pore scale in ceramic breeder materials using tmap8," *IEEE Transactions on Plasma Science*, pp. 1–7, 2022.