ORNL/TM-2024/3596

MOOSE–Workbench Integration and MOOSE Meshing Capability Enhancements to Facilitate Inputs and Outputs for Multiphysics Modeling

Robert Lefebvre
Brandon Langley
Marco Delchini
Furkan Oz
Emily Shemon
Yinbin Miao
Shikhar Kumar
Yeon Sang Jung
Aaron Oaks
Soon Lee
Kalin Kiesling
Kun Mo
Cody Permann
Logan Harbour
Daniel Schwen
Guillaume Giudicelli
Roy Stogner

September 2024

OAK RIDGE National Laboratory

ORNL IS MANAGED BY UT-BATTELLE LLC FOR THE US DEPARTMENT OF ENERGY

ORNL/TM-2024/3596

Nuclear Energy and Fuel Cycle Division

MOOSE–WORKBENCH INTEGRATION AND MOOSE MESHING CAPABILITY ENHANCEMENTS TO FACILITATE INPUTS AND OUTPUTS FOR MULTIPHYSICS MODELING

Robert Lefebvre
Brandon Langley
Marco Delchini
Furkan Oz
Emily Shemon
Yinbin Miao
Shikhar Kumar
Yeon Sang Jung
Aaron Oaks
Soon Lee
Kalin Kiesling
Kun Mo
Cody Permann
Logan Harbour
Daniel Schwen
Guillaume Giudicelli
Roy Stogner

September 27, 2024

Prepared by
OAK RIDGE NATIONAL LABORATORY
Oak Ridge, TN 37831
managed by
UT-BATTELLE LLC
for the
US DEPARTMENT OF ENERGY
under contract DE-AC05-00OR22725

CONTENTS

LIST OF FIGURES

LIST OF TABLES

ABBREVIATIONS

ANL	Argonne National Laboratory
CSG	constructive solid geometry
GUI	graphical user interface
HIT	Hierarchical Input Text
HPC	high-performance computing
INL	Idaho National Laboratory
MC	Monte Carlo
MG	mesh generator
MOOSE	Multiphysics Object-Oriented Simulation Environment
NCRC	Nuclear Computational Resource Center
NEAMS	Nuclear Energy Advanced Modeling and Simulation
ORNL	Oak Ridge National Laboratory
RGMB	Reactor Geometry Mesh Builder
VTB	Virtual Test Bed

ACKNOWLEDGMENTS

This work was funded by the Department of Energy Nuclear Energy Advanced Modeling and Simulation (DOE-NEAMS) Program under the Multiphysics Application Technical Area. This work is a collaborative effort between Argonne National Laboratory (ANL), Idaho National Laboratory (INL), and Oak Ridge National Laboratory (ORNL). ANL work was supported by the U.S. Department of Energy, Office of Nuclear Energy, under contract DE-AC02-06CH11357. The INL portion of the manuscript has been authored by Battelle Energy Alliance, LLC under Contract No. DE-AC07-05ID14517 with the U.S. Department of Energy. The ORNL portion of the manuscript was prepared by Oak Ridge National Laboratory, Oak Ridge, TN 37831 managed by UT-Battelle LLC for the U.S. Department of Energy under contract DE-AC05-00OR22725.

EXECUTIVE SUMMARY

The Multiphysics Object-Oriented Simulation Environment (MOOSE) is an open-source framework that supports many of the US Department of Energy's (DOE's) Nuclear Energy Advanced Modeling and Simulation (NEAMS) technical areas (TA). These TAs develop and use NEAMS physics and coupling modules in multiple ways to enable the research and development of complex physics models. In addition to the MOOSE framework, the NEAMS Workbench user interface provides a common analysis environment with user-interaction accelerators that streamline the tasks of model creation, review, execution, and output inspection. In FY 2024, objectives were realized in the MOOSE framework application development support and user-oriented improvements. Application development improvements support both developers and users with an expanded Reactor Module and Mesh System, stateful material property support for mortar contact, and customizable convergence criteria. Additionally, new user-oriented features were implemented in the MOOSE framework language server, including autocompletion snippets, definition from source and find reference navigations, and syntax overrides. Lastly, improvements were made to the input interpreter necessary to support the MOOSE language server and the NEAMS Workbench so that they can interact with syntactically incomplete user inputs. These improvements and more were intended to address stakeholder feedback and improve developer and user ability to conduct advanced nuclear energy modeling and simulation in support of DOE and industry needs.

1. INTRODUCTION

The MOOSE (Multiphysics Object-Oriented Simulation Environment) framework (G. Giudicelli et al. 2024) is the critical basis for modeling and simulation tools developed by the US Department of Energy (DOE) Nuclear Energy Advanced Modeling and Simulation (NEAMS) campaign. The framework provides a platform for methods research and development, as well as single and multiphysics applications, in support of the the U.S. Nuclear Regulatory Commission's objective of safety analysis and licensing readiness for advanced reactors (Commission et al. 2020) and the Department of Energy's plans for advanced reactor deployment. This report documents the continued efforts to improve foundational features and usability in the MOOSE framework and the NEAMS Workbench analysis environment in FY 2024, following accomplishments in FY 2023 (Shemon, Miao, Kumar, Mo, Jung, Oaks, Lee, et al. 2023). The NEAMS Multiphysics Applications Technical Area (TA) has the responsibility of maintaining and improving the MOOSE framework to support the numerous MOOSE-based physics applications developed across other NEAMS campaigns' TAs and users: Thermal Fluids, Structural Materials and Chemistry, Fuel Performance, and Reactor Physics.

Stakeholder feedback identified priority improvements in usability centered around enhancing the MOOSE language server and associated user interactions in the NEAMS Workbench, as well as the finite element mesh (FEM) generation capabilities. The MOOSE language server updates focused on input autocompletion improvements, new language server features to accelerate understanding, navigation, and editing user input, along with other enhancements. Improvements were focused on (1) streamlining input for repeated mesh objects; (2) enabling support for quadratic elements (preserving the volume of circular surfaces like fuel pins while also reducing mesh density requirements for physics applications); (3) implementing 3D meshing capabilities such as revolving mesh construction (useful for pebble-bed reactor (PBR) conical geometries or molten salt reactor (MSR) tanks); (4) adding more features, such as control drum construction and the ability to stitch dissimilar assemblies, to the Reactor Geometry Mesh Builder (RGMB); (5) assessing the optimal path to adding Monte Carlo constructive solid geometry (CSG) support within MOOSE; and (6) continuing to support users with mesh generation and understanding their evolving needs.

2. MOOSE USER INTERFACE INTEGRATION UPDATES

2.1 INPUT PROCESSING ENHANCEMENTS

2.1.1 Full Support for `pyhit`

In FY 2023, the Hierarchical Input Text (HIT) parser in MOOSE was replaced with the WASP-HIT interpreter for all downstream C++ usage. This involved the implementation of an adapter pattern so that all access to input data from C++ would use the underlying WASP-HIT NodeView class transparently through wrapper interfaces. However, within the MOOSE framework, there is an entirely separate use of HIT through a Python interface called `pyhit`. This is a collection of Python bindings that is generated for the C++ HIT nodes and is used for applications such as the TestHarness and MooseDocs systems.

The remaining HIT logic used through `pyhit` needed to be adapted in FY 2024. Support for the complete set of `pyhit` functionality using WASP was added. The legacy HIT implementation was subsequently removed, and WASP-HIT was made the exclusive parser. This eliminated duplicate maintenance efforts and potential for inconsistencies between the two.

Some areas of HIT were only accessed downstream by Python through `pyhit` rather than being accessed directly by C++. These areas had not been covered by the previous development of the adapter class. Multiple updates were required to fully support all of these `pyhit` scenarios:

- Updating WASP-HIT lexical patterns with all the TestHarness and MooseDocs scenarios.
- Modifying MooseDocs Python logic to address changes in the extraction of input blocks.
- Capturing inline comments in the HIT tree separately to properly render with round trips.
- Using state information in WASP-HIT to process multiple nested brace expression levels.
- Adding support of blank node types in the parse tree to compress consecutive blank lines.
- Replacing explode functionality used by HIT utilities with parsed render expansion logic.

Updating `pyhit` to use the WASP-HIT wrappers revealed needed performance improvements. For example, the retrieval of values from the HIT node tree with C++ is templated based on the expected value type, e.g.,

```
1 std::vector<int> ints = root_node->param<std::vector<int>>(path);
2 std::string      text = root_node->param<std::string>(path);
```

However, `pyhit` heavily uses the `Field::kind()` method to get parameter values from the parse tree, e.g.,

```
k = node.kind()
if k == FieldKind.Int:
    return int(f.intVal())
elif k == FieldKind.Float:
    return float(f.floatVal())
elif k == FieldKind.Bool:
    return bool(f.boolVal())
return f.strVal().decode('utf-8')
```

When `pyhit` was updated to use the adapter interfaces, a significant slowdown was observed in the startup time of the MOOSE TestHarness system. To fix this, the HIT `Field::kind()` logic was updated to directly use the wrapped WASP token type rather than deduce type from the string value through a series of expensive `try{}` / `catch{}` blocks. This improved the MOOSE TestHarness startup time by ~875% because Python relies on the `Field::kind()` method so extensively. It decreased the TestHarness startup from ~70 seconds to ~8 seconds for `moose/test` and `moose/modules` combined.

2.1.2 Include External Input

A feature that has been requested by MOOSE developers and users is the ability to define portions of input in external files and then include those files from within another input file. This capability was added to the WASP input processing prior to FY 2024, but it had not previously been utilized in the HIT wrapper interfaces described in Section 2.1.1.

In FY 2024, minor updates were added to WASP to catch error conditions like circular input file inclusions. After the legacy HIT implementation was removed to make WASP-based HIT the exclusive input processor in MOOSE, the ability to include external files was unlocked in the HIT adapter. The `!include filename` syntax can be used at any arbitrarily nested context of input. The `filename` can be any file in the current directory, relative path, or absolute path as long as it contains syntactically complete blocks or parameters.

This allowed users to define pieces of input that were common to many problems once in a single input file. That file could then be included by multiple inputs wanting to reuse the same shared input specification. For example, observe the following set of input files:

`basefile.i`:

```
[Block01]
  param01a = value01a
  !include include_param_from_basefile.i
  param01c = value01c
[]
!include include_block_from_basefile.i
[Block03]
  param03a = value03a
[]
```

include_param_from_basefile.i:

```
  param01b = value01b
```

include_block_from_basefile.i:

```
[Block02]
  param02a = value02a
  !include include_param_from_included.i
  param02c = value02c
[]
```

include_param_from_included.i:

```
  param02b = value02b
```

This example demonstrates the `!include` construct being used to import various types of input components from multiple input contexts. The result is that using `basefile.i` as the base input file will recursively comprise all contents included from downstream files. This effectively results in the following equivalent specification:

```
[Block01]
  param01a = value01a
  param01b = value01b
  param01c = value01c
[]
[Block02]
  param02a = value02a
  param02b = value02b
  param02c = value02c
[]
[Block03]
  param03a = value03a
[]
```

2.1.3 Merge of Input Blocks

After gaining the ability to include external input files (as described in Section 2.1.2), MOOSE users then wanted to allow block parameters to be split across input files. For example, suppose the following two files are used to define a problem with the idea that the `tst_pp` postprocessor subblock would be the merged definition specified over both inputs:

basefile.i:

```
!include included.i
...
[Postprocessors]
  [tst_pp]
    type = ConstantPostprocessor
  []
[]
```

included.i:

```
[Postprocessors]
  [tst_pp]
    value = 5.0
  []
[]
```

However, this was not possible and failed with the following error from the same block being specified twice:

```
A UserObject with the name "tst_pp" already exists.
  You may not add a Postprocessor by the same name.
```

The input processing was updated to combine parameters from blocks that have the same name. The example above, which previously failed because of a duplicate block name error, will now run as desired. The `tst_pp` postprocessor subblock will be made up of the `type = ConstantPostprocessor` parameter from `basefile.i` and the `value = 5.0` parameter from `included.i` merged together to become:

```
[Postprocessors]
  [tst_pp]
    type = ConstantPostprocessor
    value = 5.0
  []
[]
```

2.1.4 Override Value Syntax

In addition to the capabilities described in sections 2.1.2 and 2.1.3, users wanted to put groups of default settings common to multiple problems in a shared input file and be able to include those default settings while overriding certain values as needed. For example, suppose the following two files are used to define a problem with the idea that `defaults.i` contains default settings that are used unless overridden by the `specifics.i` input:

specifics.i:

```
!include defaults.i
...
[Functions]
  [f]
    value = '${fparse new_expression}'
  []
[]
```

defaults.i:

```
[Functions]
  [f]
    type = ParsedFunction
    value = '0'
  []
[]
```

However, this was not possible and failed with the following errors from the same parameter being specified twice:

```
defaults.i:4.5: parameter 'Functions/f/value' supplied multiple times
specific.i:4.5: parameter 'Functions/f/value' supplied multiple times
```

The ability to override the values of parameters when using included files was added to the input processing. This enabled choosing the parameter setting with precedence for conflicting specifications in included files. It also improved flexibility by allowing common settings to be included and overridden by problem specific values as necessary. Either the concise `param := value` syntax or verbose `param :override= value` syntax may be used, and both are equivalent, e.g.,

file_a.i:

```
!include file_b.i
[Block]
  param_01 :=          value_01_from_file_a
  param_02 :override= value_02_from_file_a
  param_03 =           value_03_from_file_a
  param_04 =           value_04_from_file_a
[]
```

file_b.i:

```
[Block]
  param_01 =           value_01_from_file_b
  param_02 =           value_02_from_file_b
  param_03 :=          value_03_from_file_b
  param_04 :override= value_04_from_file_b
[]
```

If `file_a.i` is used as the base input file, it would effectively result in the following specification:

```
[Block]
  param_01 = value_01_from_file_a
  param_02 = value_02_from_file_a
  param_03 = value_03_from_file_b
  param_04 = value_04_from_file_b
[]
```

So the previous example, which failed because of duplicate parameter errors, can now be specified as:

specifics.i:

```
!include defaults.i
...
[Functions]
  [f]
    value := '${fparse new_expression}'
  []
[]
```

defaults.i:

```
[Functions]
  [f]
    type = ParsedFunction
    value = '0'
  []
[]
```

The `'${fparse new_expression}'` value will override the `'0'` value for the `expression` parameter.

The WASP NodeView wrapped by each HIT node can be accessed using the `hit::Node::getNodeView()` method. The first step to accomplishing this was updating the WASP processor so it would accept an op-

tional override specifier. Then, a new `is_override()` interface that indicates whether the override syntax was used when specifying a parameter was added to the WASP NodeView.

Previously, the WASP NodeView to HIT node tree building logic did not check if a parameter had already been defined. Parameters defined multiple times in the same context were simply re-added to the HIT node tree for each occurrence. The MOOSE Parser later walked all `hit::Field` nodes in the tree and threw an error for any duplicate parameters. The final step was updating the WASP NodeView to HIT node tree build logic to use the `NodeView::is_override()` and do the following:

If the current WASP NodeView (`wasp_child`) is a field of type `KEYED_VALUE` or `ARRAY`, then:

- Look for a `hit::Field` node of the same name and context in the tree, and if not found:
  - Create new `hit::Field` node from `wasp_child` and add to the tree as is standard.
- But, if a `hit::Field` node with the same name and context is found (`found_node`), then:
  - Get override settings of previously added `found_node` and new `wasp_child` with:
    - `override_old_node = found_node.getNodeView().is_override()`
    - `override_new_node = wasp_child.is_override()`
  - Then, use the following conditions and decide how to handle those nodes in the tree:
    - `if (!override_old_node && !override_new_node)`:
      - Leave previously added found `hit::Field` node `found_node` in the tree.
      - Create new `hit::Field` node from `wasp_child` and add to the tree also.
      - The MOOSE Parser will throw an error for the duplicate parameters later.
    - `if (!override_old_node && override_new_node)`:
      - Remove previously added `hit::Field` node `found_node` from the tree.
      - Create new `hit::Field` node from `wasp_child` and add to the tree.
    - `if (override_old_node && !override_new_node)`:
      - Leave previously added found `hit::Field` node `found_node` in the tree.
      - Do not create new `hit::Field` node from `wasp_child` for the tree.
    - `if (override_old_node && override_new_node)`:
      - Throw an error as override syntax was specified twice for same parameter.

2.2 AUTOCOMPLETION ENHANCEMENTS

2.2.1 Partial Input Scenarios

In FY 2023, autocompletion assistance was added to the MOOSE language server. This covered the completion of all blocks and subblocks for any context, all parameters (global, action, and object) for any block context, and all values for any parameter context. This context-aware autocompletion was the most valuable and widely used feature added for assisting users with input creation. However, real-world use of the autocomplete functionality revealed a broad user scenario that was not yet supported.

Prior to the FY 2024 development, the MOOSE language server required that the current state of the input document be at least structurally sound according to HIT syntax in order for the autocompletion logic to return any results. But users wanted the ability to autocomplete within syntactically incomplete documents, and this was not available. This would allow a much more natural way to interact with input. Enabling it was a three step process.

First, the WASP input processing needed to support error recovery. This involved making lexer and parser updates to handle numerous erroneous input scenarios. These included parameters without equal signs or values, blocks without names or terminators, and block names without closing braces. These changes properly captured failure messages while allowing parse tree construction to continue after the error conditions were encountered.

Second, the MOOSE language server autocompletion was modified to only emit values for parameters that have defaults. Previously, an insertion value was decided for every parameter using information like enumerated choices and types. This was necessary when WASP did not support error recovery to make the input syntactically complete so that it could be captured fully in the parse tree. But this completion logic was changed so it no longer tries to determine an insertion value. Instead, `param =` is simply emitted with no value when any parameter without a default is chosen. This is now acceptable since the WASP processor has been updated to recover from this state and build the remainder of the parse tree.

Finally, new autocompletion logic was added to the MOOSE language server for each of the partial input cases that support error recovery. These scenarios are listed below.

2.2.1.1 Parameter Missing Value After Equal Sign

In FY 2023, one of the features implemented with the MOOSE language server was the autocompletion of parameter values. This subset of the completion capabilities was predicated on the request context coming from an existing value node that had a parameter or array parent. A series of checks were implemented using the parent context of that value node to determine what category of options should be presented to the user, e.g.,

- **active / inactive parameter value** - all defined child subblock names
- **boolean type parameter value** - `"true"` and `"false"` string literals
- **enum type parameter value** - enum choices with description strings
- **assoc syntax parameter value** - associated syntax path input queries
- **type named parameter value** - object action names verified by tasks

When one of the response choices was selected, it would replace the parameter value that already existed in the input and was the origin location of the completion request. After the WASP error recovery development described above, however, it was no longer guaranteed that this class of completion requests would originate from existing value nodes. The objective became to enable autocompletion of parameter values when a value is not already present in the input.

Due to the structure of partial input being completed in this scenario, the cursor may not be directly attached to a piece of text representing a terminal node in the parse tree. Instead, the cursor could follow after an arbitrary amount of whitespace to the right side of a `param =` statement that has no value. In this case, the logic that finds the node associated with the request location would return the parent block context because the range of positions representing the parameter spans from the name declarator to the equal sign.

Whenever the request context was reported to be from within a block context, extra measures were added to move backward character by character and continually check each position until the context changed. If the first change of context points to an equal sign, then the request context is updated to be that node rather than the parent block node.

The language server autocompletion conditions that previously triggered the actions listed above only when the context of the completion request was from a value node were then updated to also trigger when

the request context was from an equal sign node. Figure 1 shows an example of this in practice with the NEAMS Workbench autocompletion of block type values when there is not a value already specified in the input for that parameter.

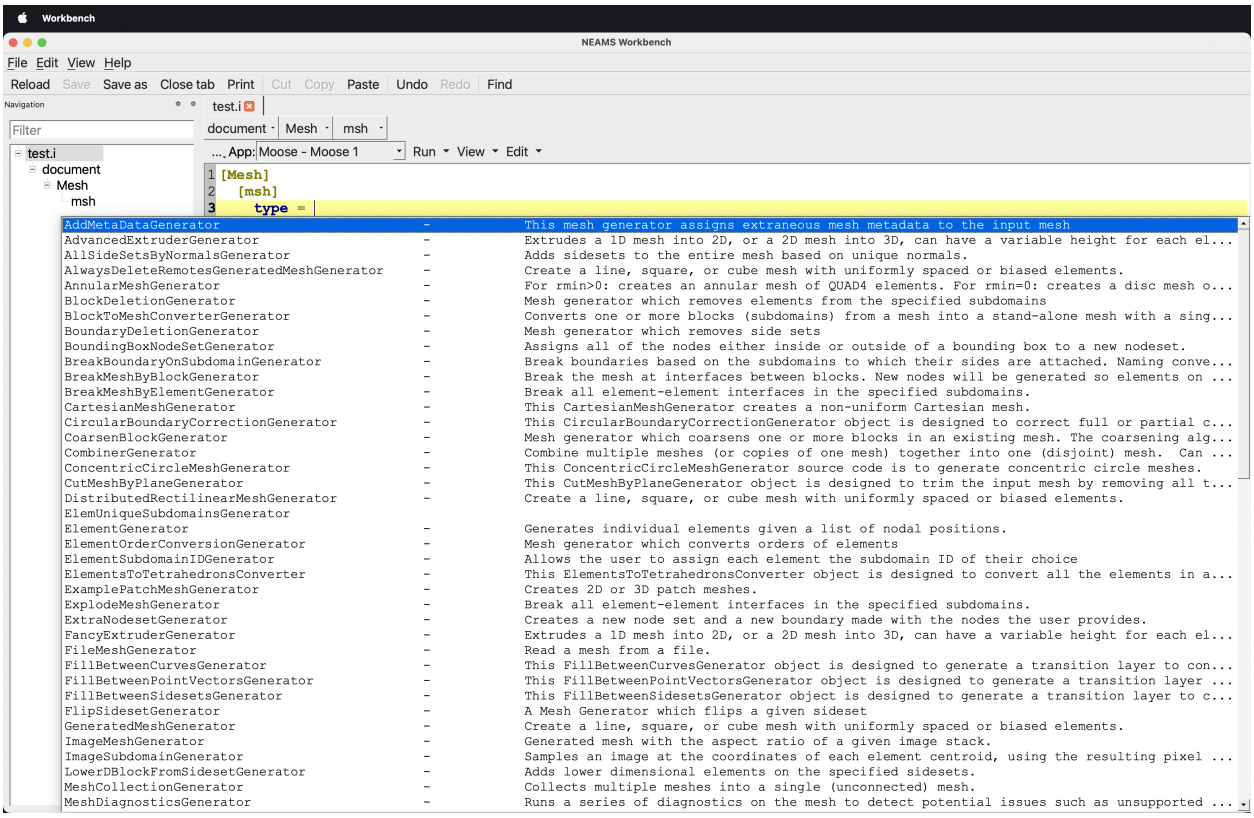


Figure 1. Partial input autocompletion when no value is specified for the parameter.

2.2.1.2 Partial Parameter Name Filtered by Prefix

The ability to autocomplete within any block and be provided all parameters that are valid to exist in that block context was another feature added to the MOOSE language server in FY 2023. Similar to the implementation previously described for values, this parameter completion was also strictly tied to the type of node from which the request originated. In this case, parameter completion was only available when the request was from directly inside a block context.

This meant that the cursor location had to be inside a block but not within the bounds of any child parameters of that block. A user could not begin typing the name of a parameter and then request autocompletion options while their cursor was still attached to the partial name because this request would not originate from a direct block context.

One of the updates during WASP error recovery development was to store standalone string primitives in the parse tree as childless declarator type nodes. In this scenario, the partial name attached to the cursor when autocomplete is requested is a parameter declarator type node. This means that the request originates from a parameter declarator context.

The language server conditions that previously triggered parameter completion only when the context of the completion request was from a block node were updated to also trigger parameter completion when the context of the request was from a parameter declarator node. Additionally, the data specified by the user for the partial parameter name were captured to use as prefixes for filtering the items provided to the user.

Parameters are only added to the list of available options if they begin with these prefixes.

One other change necessary to support this completion scenario was that the line and column range of the partial parameter name data specified by the user was captured. These positions were added to each item in the filtered list of options to be used as replacement ranges so that the partial name already specified in the input would be replaced by the full parameter name upon completion. Figure 2 shows an example of the NEAMS Workbench using this feature to autocomplete parameters from the context of a partial parameter name that is used as a prefix to filter available options.

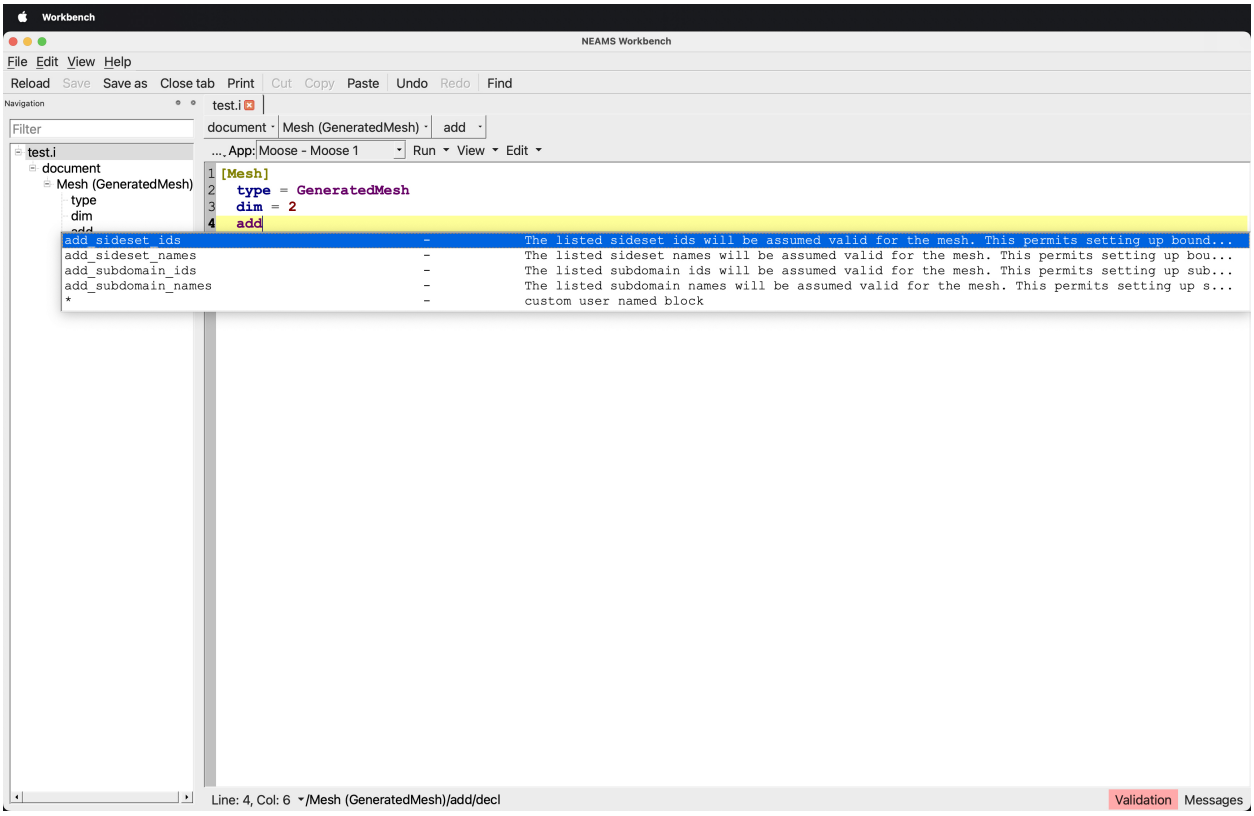


Figure 2. Partial input autocompletion from an incomplete parameter name context.

One unplanned, yet fortunate, side effect of this partial parameter name filtering capability is the way it is handled by the VS Code client. VS Code sends autocomplete requests for each keystroke rather than requiring the user to always use a special key combination. This means that the list of available options is automatically displayed and filtered as the user types the parameter prefix.

2.2.1.3 Partial Name of Block with Prefix Filtering

In addition to the autocompletion of parameters and values, another feature implemented for the MOOSE language server in FY 2023 was the ability for autocomplete and add blocks that are allowed to exist in a given context. All valid blocks were gathered and provided as options only when the request originated from the root level of the document or from directly inside another block context.

This meant that users were not able to begin typing a block name after an opening brace and then request autocompletion options. Their cursor would be attached to the partial block name, so this request would not originate from either the document root context or direct block context that was required for this type of completion. The WASP error recovery development updated the input processing to capture the partial name attached to the cursor as a block declarator type node in the parse tree so this request would originate

from a block declarator context.

Similar to the changes described above to support partial parameter name autocompletion, the conditions that trigger the gathering of blocks to be provided as autocomplete options were updated to include the block declarator context. The partial block name data specified by the user were also captured and used to filter blocks from the list that do not begin with the prefix.

The line and column range of the partial block was again captured to use as a replacement range so that the full block name would overwrite the partial name upon completion. The opening block brace was also removed from the insertion text because it would already exist in the input with this scenario. Figure 3 shows an example of this feature being used from the context of a partial block name in the NEAMS Workbench to autocomplete blocks for which the specified prefix is used to filter the provided options.

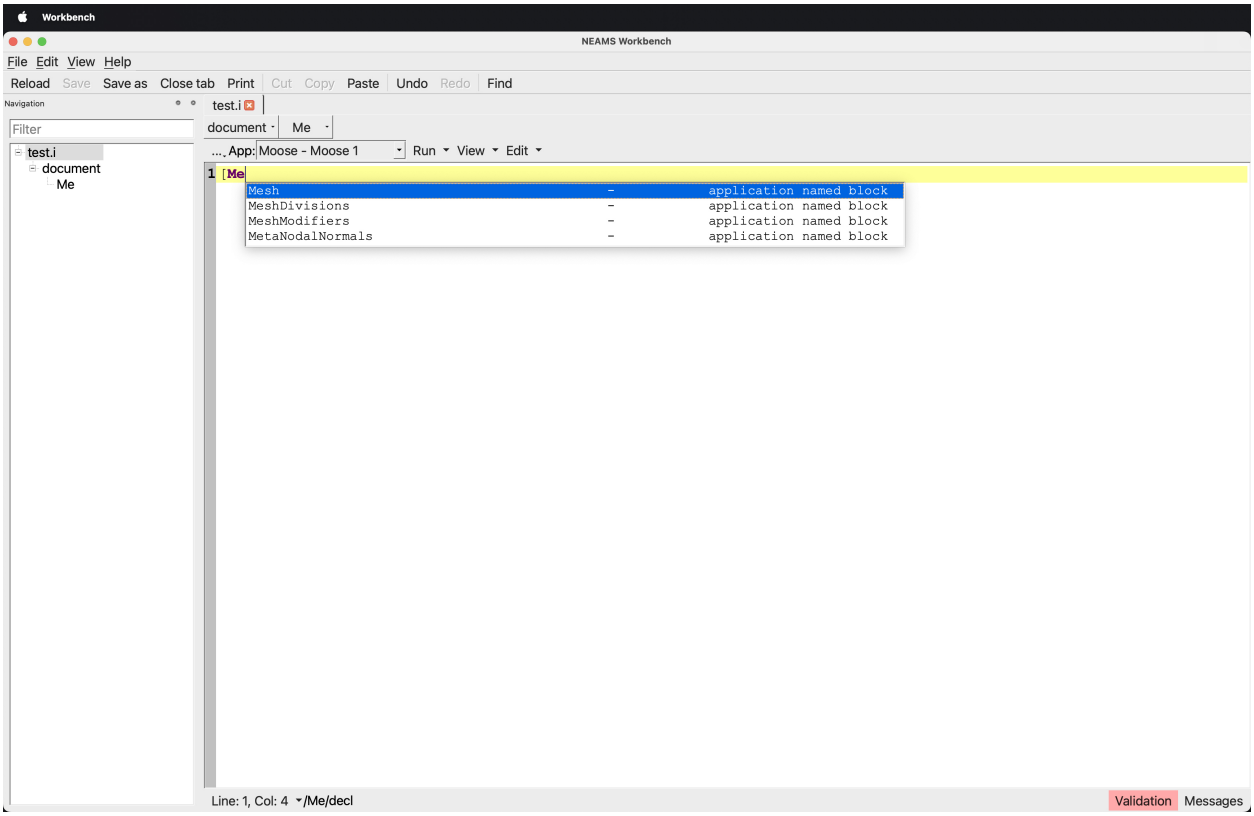


Figure 3. Partial input autocompletion from the context of an incomplete block name.

Just like with partial parameter name autocompletion, VS Code sends autocomplete requests for every keystroke while the user is typing the partial block name prefix. As the user types, all of the available blocks are displayed and automatically filtered in a completion list. In the future, integrating this feature into the NEAMS Workbench will be useful because this feature has been effective in assisting input creation.

2.2.1.4 Unclosed Start Block Brace Without Name

The user may also wish to request autocompletion immediately after typing an opening block brace and be provided all blocks available in a given context without needing to recall any block name prefix information. This was also not possible after the FY 2023 development efforts because the request would originate from the context of the opening block brace attached to the cursor.

During the WASP error recovery development, a special case was added to capture this scenario of an opening block brace with no name specified by manufacturing an empty name token and adding an empty block declarator leaf node to the parse tree. Because an empty block declarator has been added to the tree at the same byte offset location as the opening block brace, the context for the autocompletion request would once again come from a block declarator context in this case.

The updates to the autocompletion logic previously described to support partial block names also supported this scenario where no partial name is specified after the opening block brace. The node associated with this request would not have any data, so the filter prefix would be empty in this case. Therefore no block names are filtered from the list of provided options.

The line and column information manufactured for the empty block declarator node is identical to the line and column of the preceding opening brace node. This meant that upon selection of a block from the list, the autocompletion would actually remove the opening block brace from the input in this special scenario because it matches the column of the empty node. An update was needed to avoid this by bumping the columns in the replacement range out by one character. Figure 4 shows an example of this being used for autocompletion in the NEAMS Workbench from the context of an unclosed opening block brace attached to an empty block name declarator.

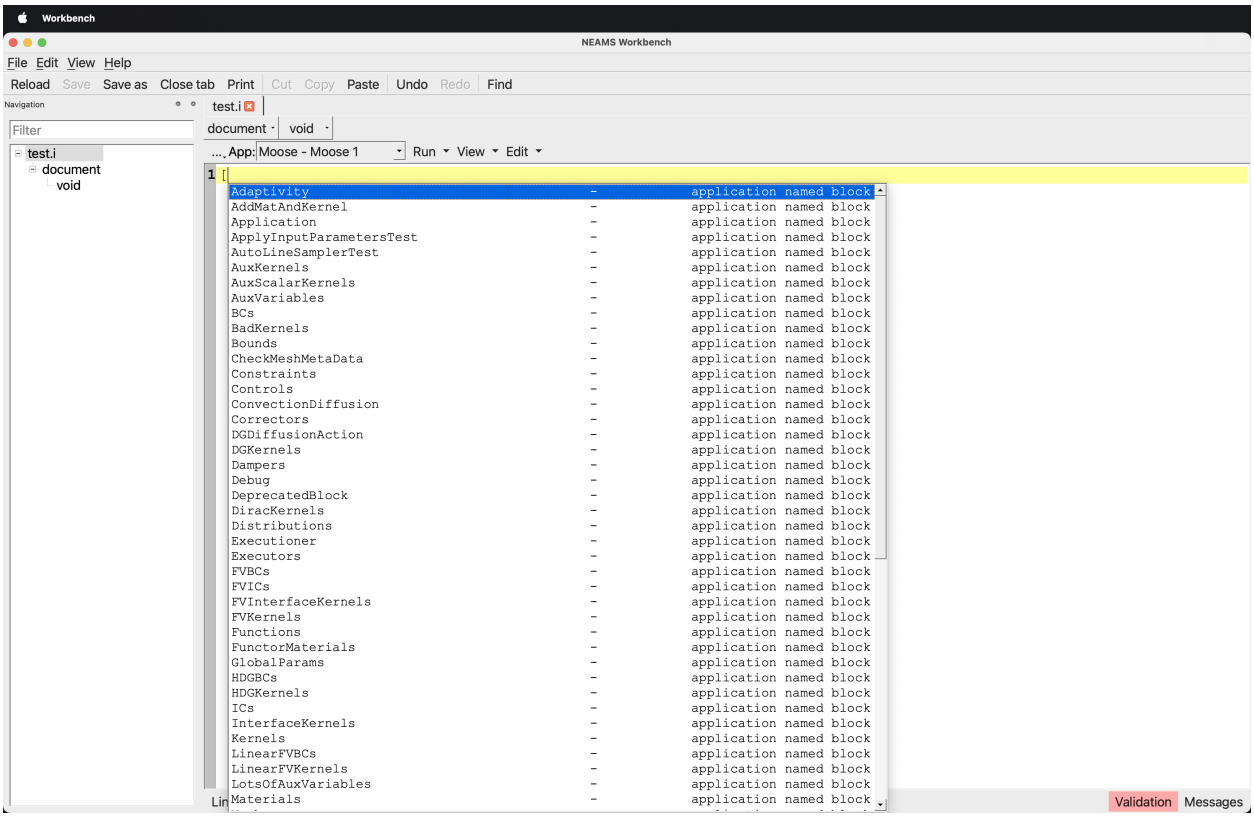


Figure 4. Partial input autocompletion at the opening brace of a block with no name.

2.2.1.5 Missing Terminator for Block or Subblock

A common process used for creating input with a MOOSE language server enabled client involves first writing a block declarator (e.g., `[Executioner]`) followed by a new line and then requesting autocomplete to choose parameters that should be added for the new block context. It is natural to want to create input in this manner without first adding a terminator (e.g., `[]`) to close the block context. However, this

was not possible after the FY 2023 development because it temporarily left the input in an erroneous state without terminating the block.

Prior to the WASP error recovery, nodes belonging to unterminated blocks were not stored in the parse tree. This meant that there was no way for the autocompletion request context in this scenario to be from the block without a terminator because no components of that block existed in the parse tree. The error recovery development captured all specified pieces of input, even when errors such as unterminated blocks occurred.

After these WASP updates properly added the block context to the tree, another issue presented itself with this completion scenario. The logic that searches the tree to find which node is associated with the request location would crash when the line and column from the request were past the end of the document and the final block did not have a terminator. It could not handle this edge case. This was updated to return the parent node when the given position was past all content defined in the input.

The autocompletion logic was also updated to first check if the line and column from the request are greater than the last position defined in the parse tree. If this is the case, then the last line and column from the input are used to capture the final node in the document.

If this final node is not a block terminator, then it is a sibling of the requested context, so the context for completion becomes the parent of this node. If this final node is a block terminator, then the parent block of this terminator is a sibling of the requested context, so the context for completion becomes the grandparent of this node. In either case, the completion request context will be the unterminated block as intended.

With these changes, users can now create input using the natural workflow described above. They are no longer required to first add a terminator for each block and then move the cursor above this terminator to autocomplete the parameters in that block context. Figure 5 shows an example of autocomplete using these updates to provide parameters to the NEAMS Workbench when the context of the request is from a block that does not have a terminator.

2.2.2 Adding Required Input

As the MOOSE language server gained traction and became more widely used, new features were requested by the user and developer communities to assist with their workflows for input creation. One of these requests was that when autocompletion is used to create a new input context, all parameters that are required to exist in that context are automatically added to the input. This request is applicable to the following two scenarios with regard to autocompletion. Both of these cases either add a new context to the input entirely or change an existing context within the input to become something else.

1. When autocomplete is used to add a new block for a given context, e.g.,
   - adding a whole `[Mesh]` block from the root level of the document
   - adding a whole `[Partitioner]` block to a `[Mesh]` block context
2. When autocomplete is used to update the value of a `type` parameter inside an existing block, e.g.,
   - changing `type = Steady` to `type = NonlinearEigen` within an `[Executioner]` block
   - autocompleting `type =` to be `type = MTICMult` within an `[InitialCondition]` block

It was noted in the motivation of this feature request that certain categories of input blocks, such as components related to actions or physics, often have a long list of parameters that are mandatory to exist in the input. It was also made apparent how valuable to users it would be if the parameters that must always be specified were included automatically by the autocompletion logic.

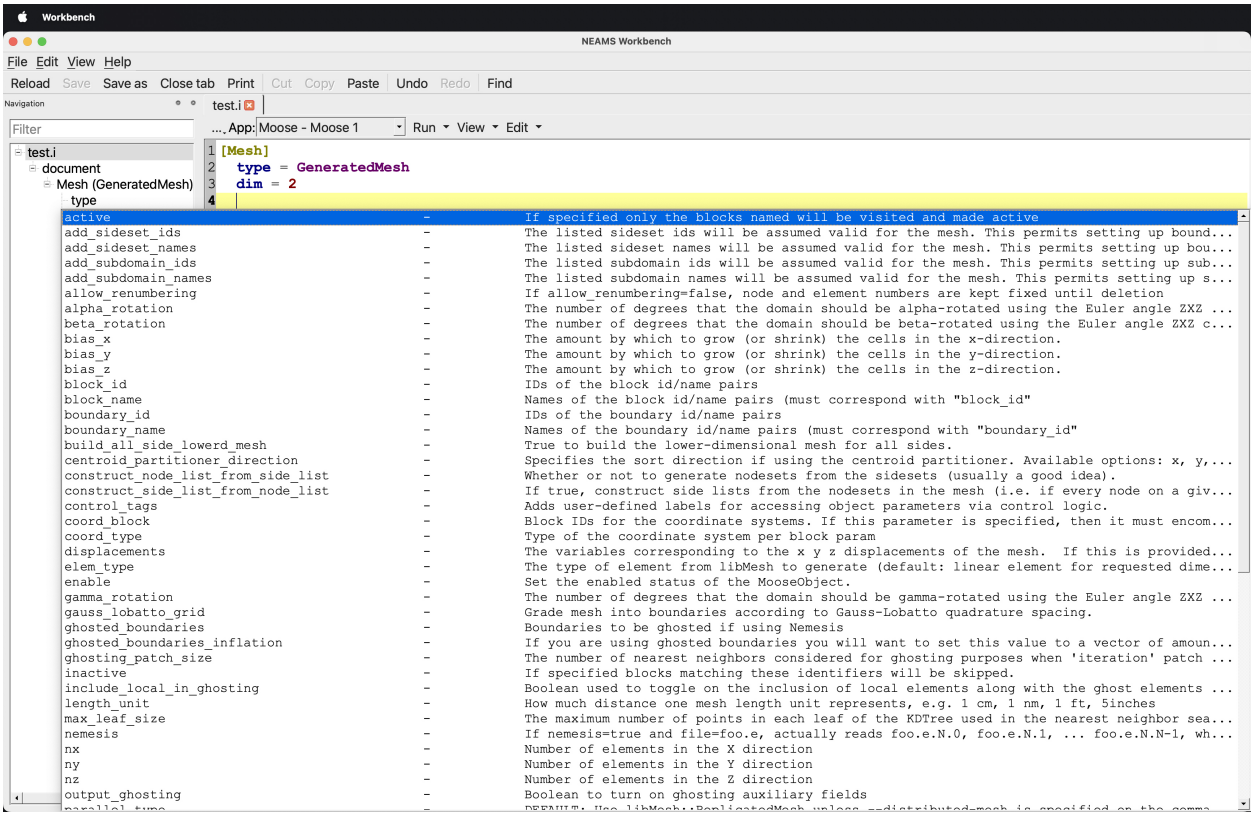


Figure 5. Partial input autocompletion from the context of a block with no terminator.

This would prevent the user from having to use autocomplete over and over again to fetch every required parameter one by one after already adding the block. Instead, autocomplete could collect all of the parameters that are not optional for the new input context and add each one with an empty value field inside the block. The user would then only need to fill in the values for the provided parameters rather than having to first search for every one that is required in the list and add them manually.

For the first scenario, in which autocomplete is used to insert entirely new blocks to the input, the full syntax path of each block option from the list that could possibly be selected is used to gather all of the action parameters that are valid to exist in that new block context. For the other case of using autocomplete to add or change the value of a type parameter in an existing block, the full syntax path of the parent block is used along with each type value option that could possibly be selected by the user to gather all of the action parameters and object parameters that are valid to exist in that combination of block and type context.

Then, for each new input context that would be created if the associated option is the one chosen from the list, the collection of parameters valid to exist in that context is traversed. Any parameter that is said to be required and is not set elsewhere in the framework gets formatted for completion with an equal sign and the proper indentation level. The parameter is then appended to a text list that will be inserted into the document within the block if that option is selected by the user.

The names of all parameters that already exist in the input as siblings of the autocomplete request location are also gathered. Parameters that exist in this set do not get added to the insertion text list for that option because they have already been specified by the user. This prevents parameters, which are allowed to be specified only once, from being added to the input a second time when the user just changes a type value

using autocomplete.

The required parameters are only inserted when the user actually selects the block name or type value in the autocomplete list by clicking or pressing return. It does not perform any live updates to the input by adding and removing parameters while the user is typing partial block names and type values or scrolling through the list of available options. Although a separate set of parameters may be gathered and constructed for every possible selection in the list, only the single set associated with the option that the user chooses will be inserted into the input at the time of selection. The lists for all of the other options are built so that they are ready to be used in case they get selected, but these lists are simply discarded otherwise.

This capability could not have been added without the WASP error recovery development and the updates to support autocompletion of absent parameter values described in section 2.2.1 happening first. Prior to those efforts, it was not possible to insert these required parameters into the input without values (e.g., `param =`) and then use autocomplete to add values when options are available. Those features were necessary prerequisites for this feature. Figure 6 shows examples of the required parameters that are collected and automatically added for various contexts in the NEAMS Workbench when autocompleting new blocks and block type values.

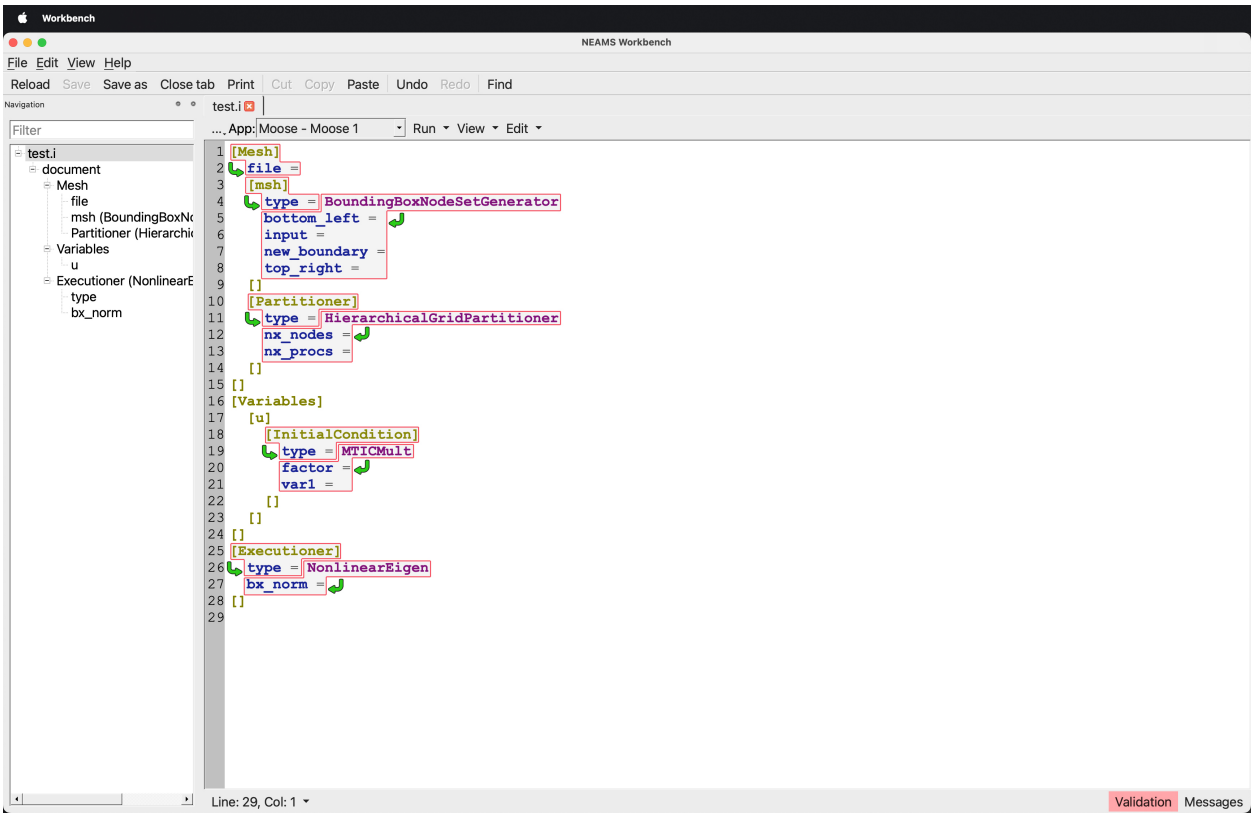


Figure 6. Required parameters added when autocompleting blocks and type values.

2.2.3 Completion Kind Icons

Client editors like VS Code have the ability to choose an icon to display for every item in an autocomplete list based on what value the language server provides for the `completionItemKind` field of that option. However, after the initial autocompletion capabilities were implemented for the MOOSE language server in FY 2023, it was discovered that VS Code was not presenting different icons for the various categories of options available. The editor was, in fact, always displaying the same icon for all autocomplete options in

every list, regardless of the type or context.

This was happening because the autocompletion logic on the server side was not performing the required examination of item properties to properly communicate their types to the client. Therefore, type introspection was added to the MOOSE language server using constructs like the registered syntax, factories, and data of each component. This logic was used to choose a `completionItemKind` value for every autocompletion option that would be provided to users by the client and then to apply the chosen values to the associated items in the response list.

Providing clients like VS Code with this information enabled icons to be used as a way to quickly differentiate separate groups or categories of available options. This update automatically led to more feedback for an improved experience. This increased the potential of recognition through visual cues, which can allow faster autocompletion speeds from a user perspective. Figure 7 shows an example of VS Code presenting an autocomplete list of parameters in which the icons being displayed for all of the items are based on the provided `completionItemKind` values.

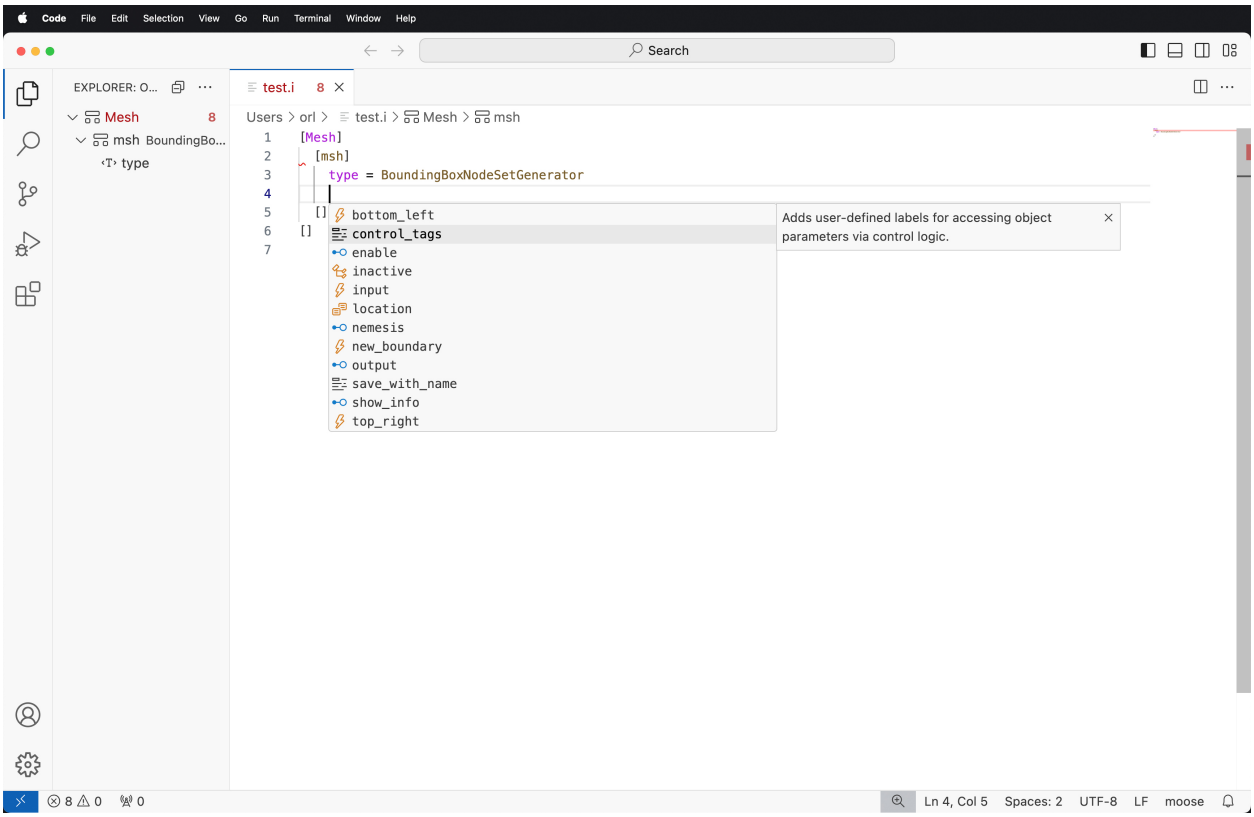


Figure 7. Autocomplete icons used in VS Code based on the provided introspection.

The NEAMS Workbench does not yet support using the `completionItemKind` fields from completion responses to display icons. However, the MOOSE language server uses a specific value for this field to indicate any parameter that is mandatory to exist in the input. The NEAMS Workbench is able to use this setting and prepend a `(REQUIRED)` tag to the description label of any parameters in the list that must be specified.

2.2.4 Snippet Syntax Support

After the development in FY 2023, whenever autocomplete was used to insert a new block into the input, the cursor was left in the location immediately following the block. After autocomplete is used to add an

input block, the next action for a user will most commonly be to add an element inside that block. But with the cursor being positioned after the terminator of newly added blocks, it left for a somewhat clumsy interaction involving the user consistently needing to either click inside the block or use their arrow keys to move the cursor back to the intended context to add this element.

Similarly, when autocomplete was used to add a new parameter that had a default value, the cursor was left in the location immediately following that default value. In this case, a user usually intends to either update that default value or at least view the available options as their following action. However, the cursor being positioned at the end of the default parameter values created another awkward editing experience. The user was required to either select the entire value to replace it or arrow backward to trigger autocomplete again directly from the beginning of the value context.

Both of these scenarios involving the position of the cursor as well as the selection of text after autocompletion are actually already supported by the language server protocol. However, this construct had not been previously been added to the autocomplete logic. It was requested that this capability be utilized by updating the MOOSE language server to provide autocompletion text formatted as snippets.

However, snippet formatting should not be added to the autocomplete insertion text across all cases. Certain client editors may not support the ability to properly apply snippet syntax to autocompletion. If the MOOSE language server always formatted its completion responses as snippets, then clients that do not support the capability would insert mangled text with invalid decorative characters directly into the document.

These editors would not use those special syntax constructs for their intended purpose to control actions like movement, selection, and placement of the cursor upon completion. Special care needed to be taken so that the snippet syntax would only be applied to autocomplete responses when the client editors support the snippet capability for autocompletion. Fortunately, one of the requirements in the protocol is that a client must notify the server in the initialization request if this is one of their capabilities.

Similar to `completionItemKind` discussed in section 2.2.3, another optional field available for each autocomplete response item is `insertTextFormat`. This field is used to indicate the format of text to be inserted when a completion item is selected and to define how it should be interpreted by the client. This property is not required to be set for any of the items in the completion response. If the field is omitted, then the default interpretation is for the insertion text to be treated as plain text. If a language server intends for a client to treat any completion item as a snippet, then the server must set this field for that item.

The only two options allowed for this property are plain text, which is inserted without any formatting applied, and the snippet format. If the `insertTextFormat` field is specified to indicate that the snippet format is being used for insertion text, then various formatting constructs can be taken advantage of to assist with input creation. Client editors can use these to decide how to handle the inserted text and provide a better user experience. For example, tab stops can be used to control cursor movement during autocompletion, and placeholders can be used to select values so they can easily be updated. This allows users to easily navigate and complete multiple areas of the input quickly.

The MOOSE language server is derived from an abstract base language server class in WASP that implements all of the general protocol logic that is not language specific. Any language server implementation that is derived from this will inherit all of the common behaviors and attributes that are shared among all servers without having to worry about implementing pieces not specific to their language. Prior to FY 2024, the `insertTextFormat` field did not exist in the autocomplete packets defined at this base server level. Because the protocol defines this as an optional property, it was never required to exist for the completion items and was not added during the initial development of the base class.

If `insertTextFormat` could be explicitly set to indicate the snippet format, then the completion item text could be updated with snippet formatting to improve user interaction. For example, placeholders could be used for parameter completion (e.g., `param = ${1:value}`) so that values are automatically selected in the client, and tab stops could be used for block completion (e.g., `[Block]\n $0\n[]`) to automatically move the cursor inside any added blocks.

However, because this field was missing entirely, it was impossible for any derived servers to set the property. This meant that the default plain text behavior of text insertion with no formatting applied was the only option available for any client. This required an update to add this field to the base autocompletion item packets and then propagate its availability throughout all of the necessary interfaces in the package.

The MOOSE language server was then updated to check if the connected client editor claimed to support the snippet autocompletion capability in its initialization request. The VS Code editor, for example, added snippet completion years ago, whereas the NEAMS Workbench was updated in FY 2024 to support a limited snippet syntax. If a client does have this capability, then the `insertTextFormat` field is set to indicate that snippet formatting is being used, and the insertion text items are formatted with snippet syntax. But if the client does not claim to support snippet completion, then the autocomplete logic leaves its responses in the plain text format. The MOOSE language server currently uses snippets for the following two cases:

- When a parameter with a default is completed, the value is selected so a user may easily change it.
  An example is using autocomplete to add a parameter named `enable` with `true` for a default value.
  - If the client does not support snippets, this is sent, and the cursor gets placed after the value:

    ```
    enable = true
    ```

  - If the client does support snippets, this is sent to indicate that the value `true` will be selected:

    ```
    enable = ${1:true}
    ```

- When a block is completed, the cursor is put on a blank line in the block and indented two spaces.
  An example is using autocomplete to insert a new block named `Postprocessors` into the input.
  - If the client does not support snippets, this is sent, and the cursor gets placed after the block:

    ```
    [Postprocessors]

    []
    ```

  - If the client does support snippets, this is sent to indicate the cursor will be at the `$0` location:

    ```
    [Postprocessors]
      $0
    []
    ```

2.3 GENERAL SERVER ENHANCEMENTS

2.3.1 Show Hover Description

The MOOSE language server was updated to add hover support for all available input documentation types. This enabled the ability for a user to hover their cursor over a piece of input in a client editor and be presented with a popup tooltip showing the description of the component. This new hover capability is predicated on the request context coming from either the key or value of a valid parameter. If the request context is from any other type of node in the input, then an empty string is always returned in the response to the client.

Otherwise, the full syntax path of the parent block context for the parameter, as well as the value of the type parameter in that block (if one exists), is captured. These block path and type values are used to gather the set of all global parameters, action parameters, and object parameters that are valid to exist at the context of the request location. This full collection of parameters is then used to query the documentation string defined for the following three scenarios:

- If the hover request context is the key of a parameter, then respond with its registered description.
- If the hover request context is an enumerated type value, then respond with its item description.
- If the hover request context is the type value for a block, then respond with its class description.

If the request is from a parameter key that is not valid to exist in its context, then an empty string is returned. Likewise, if the request is from an enumerated type value that has not had a documentation string defined, then it will return an empty response string. Furthermore, if the request is from a type parameter value that not a registered object, then it will also return an empty string in the response. Figure 8 shows an example of this capability being used to hover over the key of a parameter in the NEAMS Workbench and being shown a popup tooltip displaying its documentation string.

2.3.2 Tree Symbol Kind Icons

Client editors like VS Code have the ability to choose an icon to display for every item in the navigation tree outline view based on what value the language server provides for the `symbolKind` field of that symbol. But after implementing the initial hierarchical document symbols capability in FY 2023 for the MOOSE language server, it was found that VS Code did not display different icons for the numerous types of symbols that should be available. Instead, the client always displayed the same icon for all document symbols in the outline tree regardless of what kind of input context each symbol was referencing.

This was due to a lack of introspection necessary in the server to determine a classification for each symbol when building the document symbol tree. The logic used to build the hierarchical symbols directly traverses the NodeView tree created by WASP processing, so the type stored in each NodeView was examined in conjunction with the input data captured in each leaf to choose a suitable `symbolKind` value for each symbol inserted into the tree. These values were then added to every symbol in the hierarchy before providing the tree to the client.

Editors like VS Code were then able to use this information to associate an appropriate icon with each symbol and apply it when constructing the outline view. This created a more feature-rich experience by increasing the amount of feedback that was available to be provided to users. It resulted in a better way to visually discern between different components in the tree to allow for potentially quicker navigation to the intended pieces of input.

A separate reason for implementing the symbol analysis was of likely greater value than the aesthetic improvements the icons contributed to the editor. The document symbol tree was being created at too fine a level of granularity, and it displayed far too much detail in the VS Code outline view. Although this issue

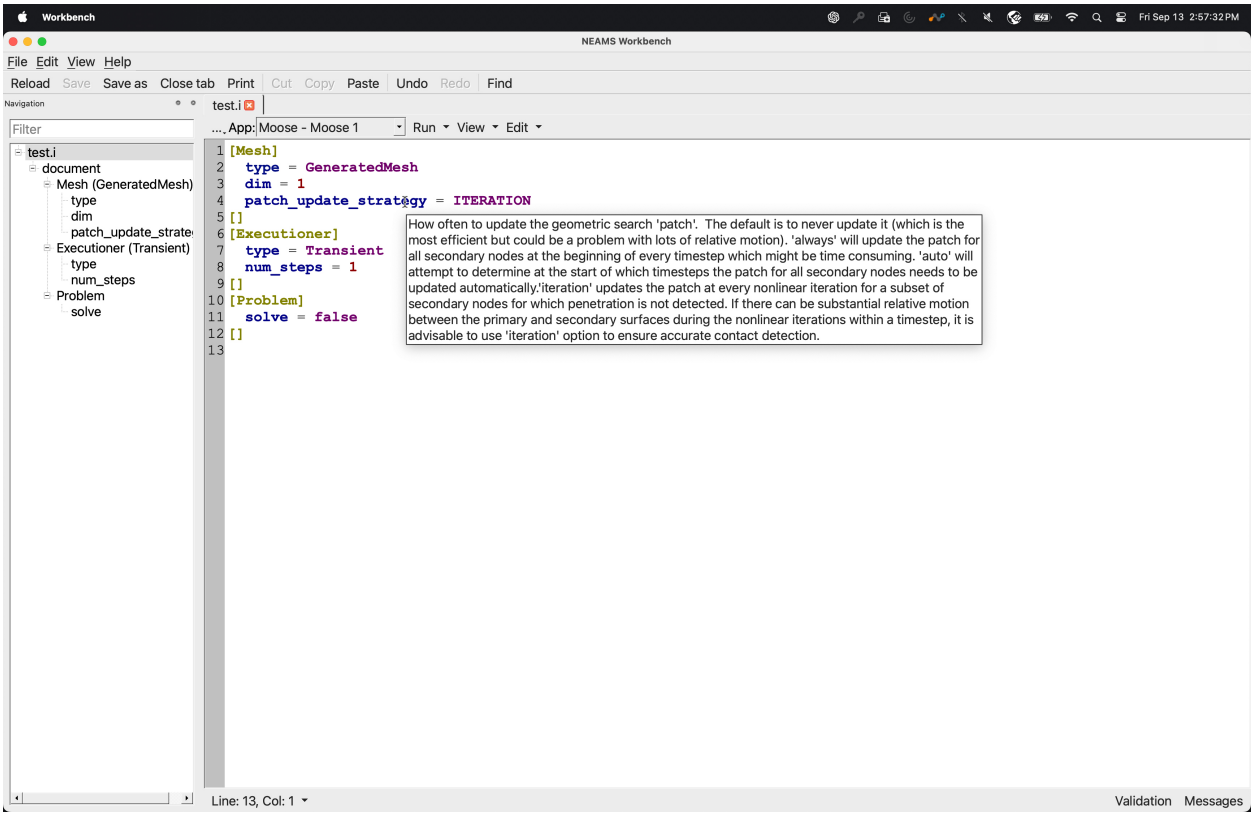


Figure 8. Hovering the cursor over a parameter key to show its documentation string.

was somewhat overwhelming to VS Code users, the NEAMS Workbench needed all of these symbols to perform other tasks such as reconstructing positions used by its document structure, breadcrumb trail, and cursor context. The NEAMS Workbench does not yet use `symbolKind` values for any purpose, but it does already filter down its own navigation tree level of detail by removing all decorative and terminal value symbols to present a much more coarse and manageable hierarchy.

Other, more general client editors like VS Code that use the MOOSE language server also needed a way to filter down their document outlines and remove elements that only clutter up the view without adding any value. This was the second big advantage of breaking down the document symbols into categories and applying relevant `symbolKind` values. The `symbolKind` value was set to `SymbolKind::Property` for all decorative syntax (e.g., `[`, `]`, and `=`) in the tree, and the field was set to `SymbolKind::String` for all terminal value nodes.

The `SymbolKind::Property` value was then able to be filtered in the VS Code extension logic to hide all decorative nodes from the outline view. Likewise, the `SymbolKind::String` value was filtered to hide all terminal values. By using these two special classifications in the server when building the hierarchical symbols and then simply updating the extension to ignore the categories when constructing the outline, it filtered the view to significantly clean up what is presented to users. Figure 9 shows an example of a too granular VS Code outline before this capability (on the left) compared with after the `symbolKind` values were added to filter the level of detail (on the right).

2.3.3 Find References in Input

The MOOSE language server was updated to add support for a new capability to find all input references. When provided with location where an input component is defined, the ability to traverse the entire input

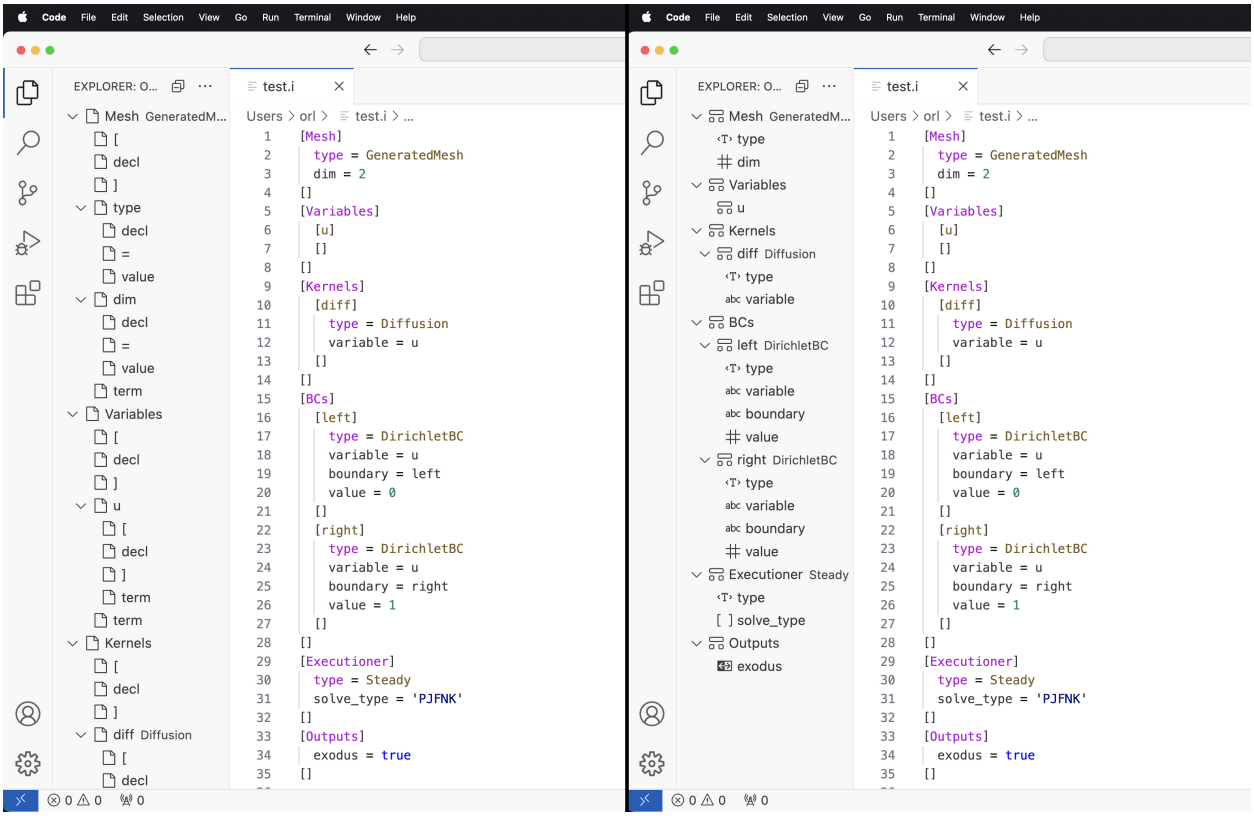


Figure 9. VS Code outline before symbol analysis (left) and after (right).

and use metadata to find references to all locations where that definition is used as the value of a parameter throughout the input is now available. Users can now take advantage of this feature to request a list of references from a client and quickly navigate to each location.

For example, if a user requests references to a variable named `[u]` defined in a `[Variables]` block, a list of all locations where that variable is used will be presented in the editor. Every target reference has a file, line, and column position associated. This information is used to provide a method for navigation. The list will even contain the location of any references that exist in externally included input files.

This references capability is only applicable when the context of the request comes from a block name declarator because block names are where the source definitions originate. If references are requested from any other type of input component, then an empty response list is always returned. The first step is that a map from all registered syntax paths to their associated parameter types is constructed. This map is currently as follows:

Subblock Syntax Path	Registered Type(s)
`Mesh/*`	MeshGeneratorName
`Functions/*`	FunctionName
`MeshDivisions/*`	MeshDivisionName
`Distributions/*`	DistributionName
`Samplers/*`	SamplerName
`Variables/*`	VariableName + NonlinearVariableName
`AuxVariables/*`	VariableName + AuxVariableName
`Materials/*`	MaterialName
`FunctorMaterials/*`	MaterialName
`Postprocessors/*`	PostprocessorName + UserObjectName
`VectorPostprocessors/*`	VectorPostprocessorName
`Reporters/*`	ReporterName
`Positions/*`	PositionsName
`Times/*`	TimesName
`Outputs/*`	OutputName
`Executors/*`	ExecutorName
`UserObjects/*`	UserObjectName
`Adaptivity/Indicators/*`	IndicatorName
`Adaptivity/Markers/*`	MarkerName
`MultiApps/*`	MultiAppName

Next, the syntax path registered for the location of the request context is obtained. This registered syntax path is used to query the previously built map to gather all parameter types associated with the request context. The entire input tree is then recursively walked to collect all parameter nodes that match any associated parameter type and also have a value matching the name specified in the input for the block at the request context location. These location nodes, which match both type and value, are used to build response objects that are inserted into a list of references and returned to the client editor. Figure 10 shows an example of the NEAMS Workbench using this capability to find all references to a definition and navigating to the line and column positions associated with each response location.

2.3.4 Navigation to Definition

Users are now able to quickly navigate to the definition of a component. In addition to navigating within an input, users can conduct definition navigation where a definition resides in the source code of the application. Specifically, if a user has access to the source code, then a definition request from the value of a block type parameter will navigate to the source code file and highlight the line where that object type was registered. This provides a significant benefit to the application user who is also a framework developer.

Additionally, users can now conduct definition navigation involving multiple input files. Because the new file include feature allows users to split inputs across files, it is possible for the definition of a component to reside in an included file. The ability to navigate to definitions that exist in external files is also available.

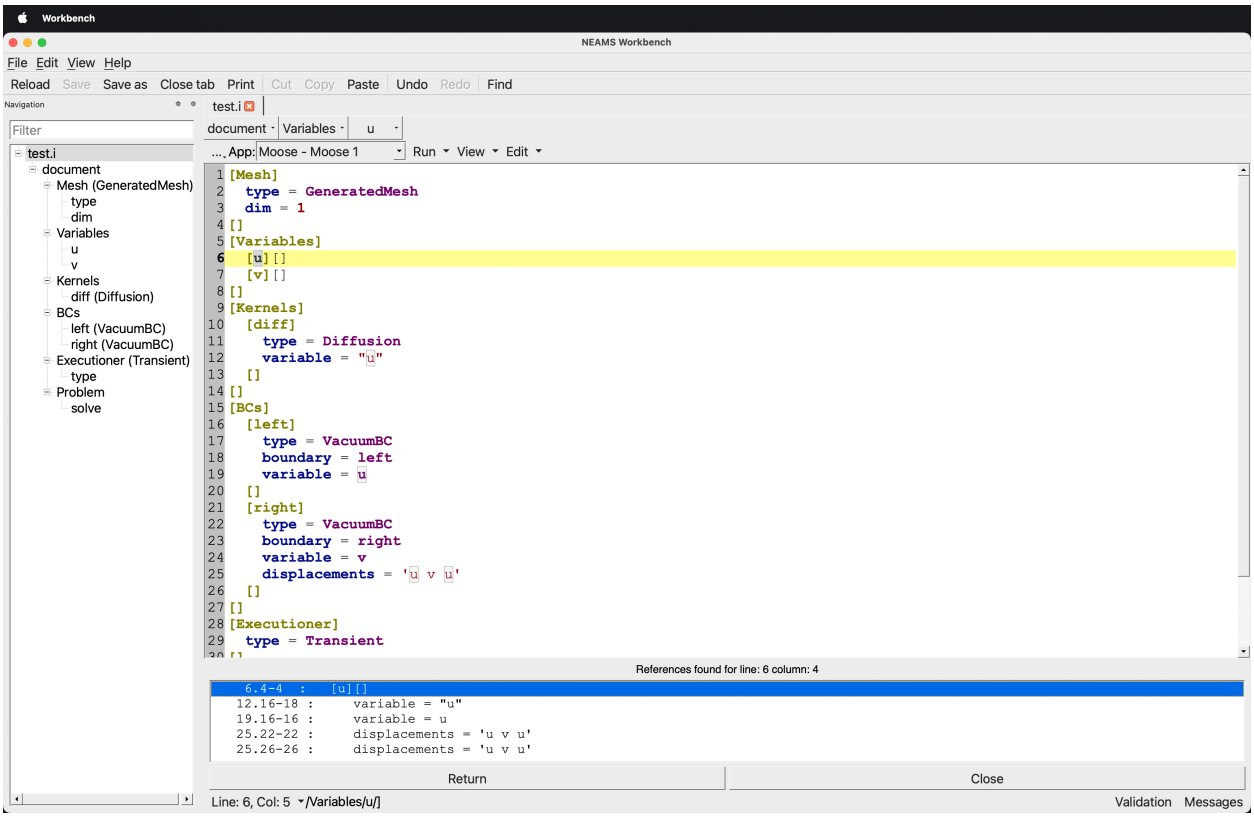


Figure 10. Find references request response with locations to be used for navigation.

2.4 ERROR CONTEXT ENHANCEMENTS

With the extensive development of the language server capability, it has become increasingly important to associate as many errors as reasonably possible to a location within the input file. Some association has already been achieved through the `paramError()` method, in which a developer can tie an error to specific parameter. However, the `mooseError()` method, which is used for a general error, was context-less.

For example, a context-less error looked something like this:

```
*** ERROR ***
The following error occurred in the Kernel "diff", of type "Diffusion".

You did something bad.
```

This is not particularly useful for presentation to the user within an input file when using the language server. Although it would have been possible to parse the output of the error header (the Kernel `diff` component) and find the corresponding syntax in input, this is a tedious task and is not general. A significant overhaul to the inner propagation of input file context was added to the `mooseError()` system. This overhaul enables the overwhelming majority of errors to contain at the very least the context of the object that they were produced within (if not more). The error above would then be presented as follows:

```
*** ERROR ***
/path/to/input.i:5:1:
The following error occurred in the Kernel "diff", of type "Diffusion".

You did something bad.
```

This additional context enables the majority of errors to be tied to a specific location in input. This allows the errors within input to be presented by the language server.

Internally, this was achieved by detailed bookkeeping of the parse node associated with every object within the error context. The `paramError()` context was also improved by associating parameters in objects that are set via action parameters to be associated with the corresponding action parameters.

2.5 DOCUMENTED ERRORS

Situations exist in which an error scenario or an unsupported capability is well known and is described in an application repository issue. Issues are at the forefront of feature development and community support, which commonly include detailed discussions about issues and oftentimes include textual descriptions of resolutions to issues. To make issues easily available to both users and developers, a standard syntax for producing an error that is associated with an issue was implemented. This error is produced by the `mooseDocumentedError` message as follows:

```
object.mooseDocumentedError("moose", 1234, "Something something your error");
```

In this example, the error should be associated with the `moose` repository issue #1234, which will produce the following error:

```
*** ERROR **
/path/to/input.i:5:1:
The following error occurred in the Kernel "diff" of type "Diffusion".

Something something your error

This error is documented at github.com/idaholab/moose/issues/1234.
```

2.6 VSCODE EXTENSION UPDATES

With the deployment of the MOOSE language server, substantial changes to the official VSCode *MOOSE Language Support* extension (Schwen, D and Giudicelli, G 2024) became necessary. Previous versions of the extension implemented a separate language server using the `Microsoft/vscode-languageserver-node` library. This language server would launch a MOOSE executable to dump a static description of the MOOSE syntax in JavaScript Object Notation (JSON) format. No MOOSE server process was kept running and no simulation objects were constructed. Although this approach guaranteed minimal memory usage, the limitation to static syntax by design did not allow the autocompletion of runtime items, such as variables or objects added by actions and material property names.

Beginning with version 1.0.0, the built-in static syntax language server was dropped; instead, a MOOSE executable in language server mode was launched. The Idaho National Laboratory (INL) VSCode extension developers coordinated with the ORNL language server developers to improve compliance with the language server protocol for the best possible user experience in VSCode. The extension currently provides dynamic syntax completion, preselection of inserted default values for quick overwriting, a clean outline view for fast input navigation, icons indicating the type of autocompleted values, and hover support for documentation strings.

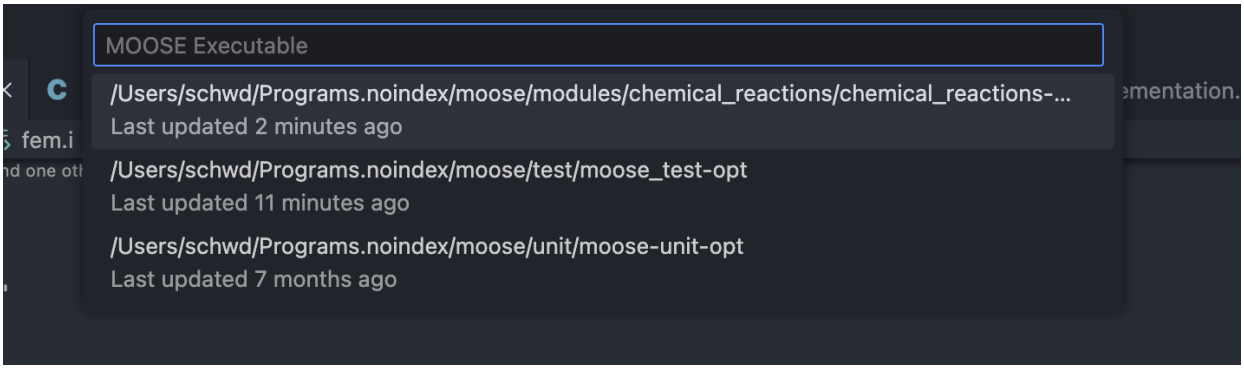


Figure 11. VSCode extension user interface for selecting the language server MOOSE executable.

In the current version of the extension, the user is prompted to select a MOOSE executable from anywhere in the current VSCode workspace. The extension compiles a list of executables and sorts them by modification date. A default executable can be added to this list in the top position by setting the `MOOSE_LANGUAGE_SERVER` environment variable. This enables setting up binary-only environments where the MOOSE executable is not located in the current workspace (software as a service).

The extension has now been deployed since March 2024, and future developments will focus on incorporating user feedback. A planned change is to bring back the automatic executable detection from previous versions to make the extension minimally intrusive.

3. CARDINAL INTEGRATION STATUS

In FY 2024, the Workbench team has continued to integrate Cardinal (Novak et al. 2022) into the NEAMS Workbench to run on Sawtooth. Cardinal is now available as a container on Sawtooth. The NEAMS Workbench now integrates this new feature by loading the Cardinal container as part of the initialization process. After loading the container, the Cardinal executable *cardinal-opt* is available in the path and is used to run jobs on the HPC platform. This updated workflow was tested with a case taken from the tutorial folder that is *test/tests/cht/sfr_pincell*. Screenshots of the different steps are shown below. First the Cardinal container is activated with the localhost feature (Figure 12). Once the activation process is finalized,

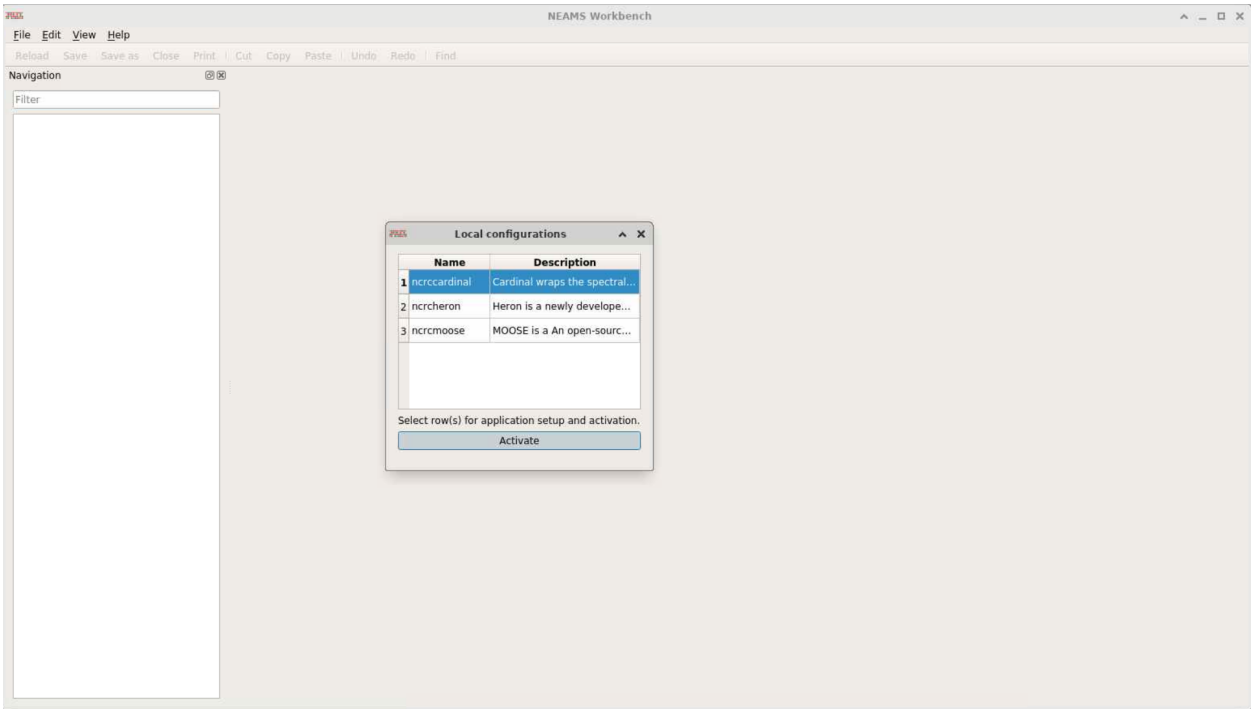


Figure 12. Activate Cardinal container through the localhost feature.

the master input file *nek_master.i* is opened with the NEAMS Workbench and submitted to the queue (Figure13). The submission process consists of clicking on the Run button, which opens a widget with PBS options to edit, and then clicking on *Ok* to submit the job to the queue (Figure 14). Once the job starts, the solver output is redirected to the *Output*. After completion of the job, the numerical solution can be visualized with the built-in ParaView GUI (Figure 15). This test showcases basis capabilities of the NEAMS Workbench to run Cardinal on Sawtooth. The following points of improvement were identified for future work:

- Some Cardinal test cases (G. L. Giudicelli et al. 2023) require users to pass two input files to the executable. This capability is not currently available with the NEAMS Workbench but will be made available soon.
- Cardinal relies on the third-party package NekRS, which requires additional preprocessing and post-processing steps to convert the mesh to a Nek-compatible format and to visualize the numerical solution with ParaView or VisIt. As part of an effort to integrate Nek5000 to the NEAMS Workbench,

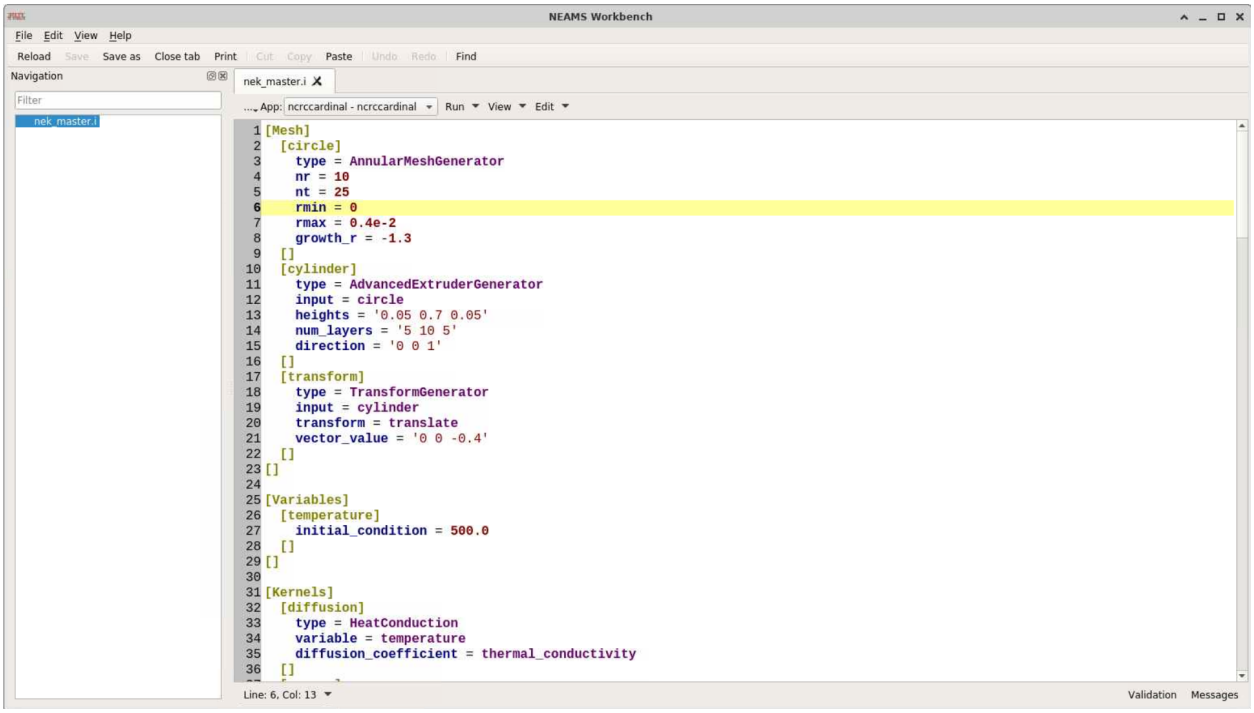


Figure 13. Open master input file in the NEAMS Workbench.

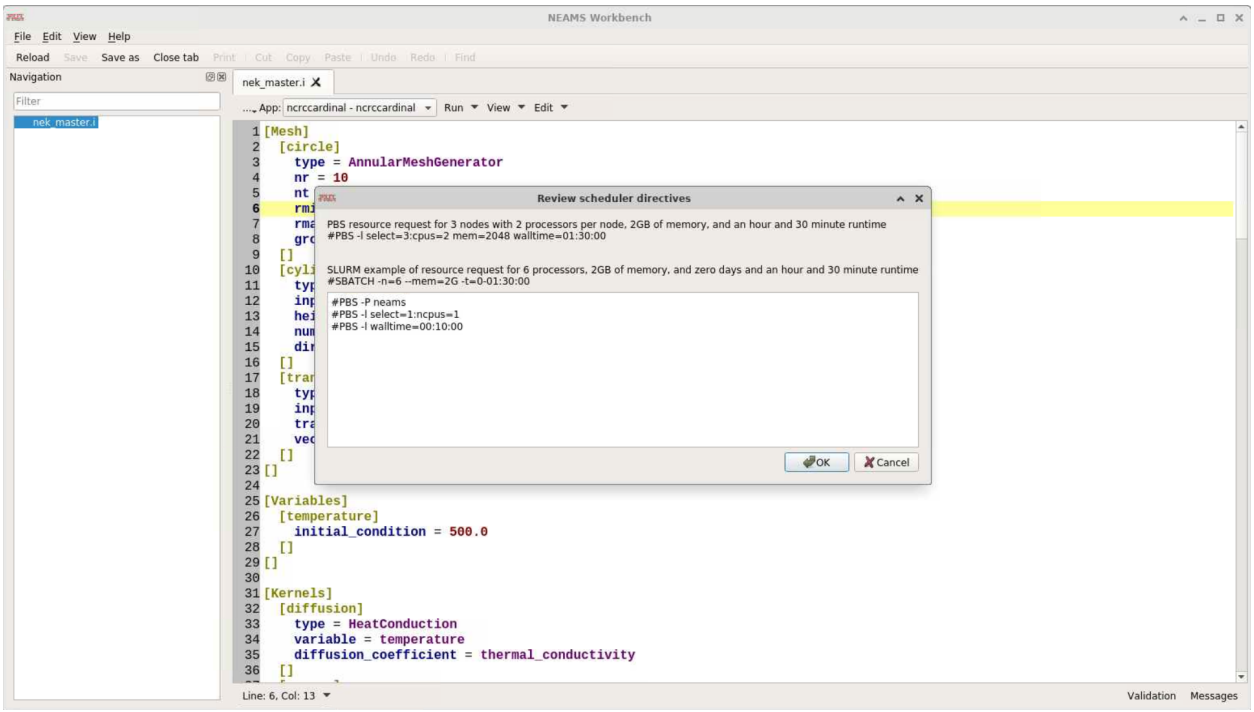


Figure 14. Submit jobs to the Sawtooth queue.

the same preprocessing and post-processing steps were supported. Further integrating Cardinal to the NEAMS Workbench will require leveraging this work to streamline the workflow.

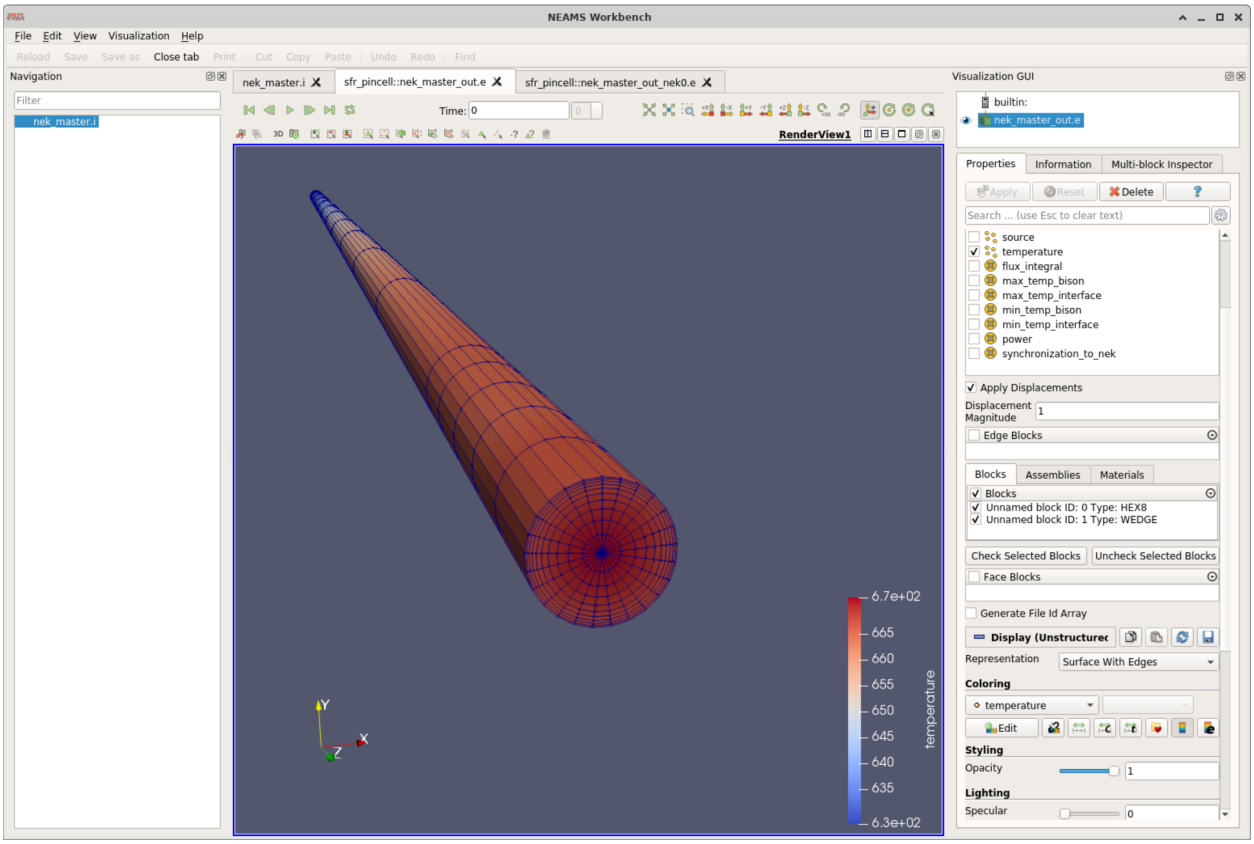


Figure 15. Visualize numerical solution with ParaView.

- Cases using OpenMC have additional XML input files that are generated by a Python script. This step currently requires users to call the Python script at the command line. Discussions will be held soon with the Cardinal lead and developers to see how some of the above points could be addressed.

4. ADVANCED REACTOR MESHING CAPABILITY ENHANCEMENT IN MOOSE

Generation of finite element meshes for NEAMS applications is a roadblock to users who are accustomed to traditional physics codes with much simpler geometry input. To address this user roadblock, MOOSE's native finite element meshing capabilities have been continuously improved since 2021. The primary objective has been of to provide a free, open-source, easy-to-use meshing tool for advanced reactor geometries. The MOOSE Reactor Module (Shemon, Miao, Kumar, Mo, Jung, Oaks, Richards, et al. 2023) and Mesh System are now the most commonly used mesh generation tools for analyzing reactor core geometries with NEAMS tools, particularly for any multiphysics problem containing reactor physics. User feedback has been overwhelmingly positive, and users have been attempting to use the tools for increasingly complex applications. Through user feedback, capability gaps are continually identified and addressed to further improve the user experience.

The meshing improvements addressed this year were (1) streamlining input for repeated meshing objects; (2) enabling support for quadratic elements (preserving the volume of circular surfaces like fuel pins while also reducing mesh density requirements for physics applications); (3) implementing 3D meshing capabilities such as revolving mesh construction (useful for PBR conical geometries or MSR tanks); (4) adding more features, such as control drum construction and the ability to stitch dissimilar assemblies, to the RGMB; (5) assessing the optimal path to adding Monte Carlo CSG support within MOOSE; and (6) continuing to support users with mesh generation and understanding their evolving needs.

4.1 STREAMLINING REPEATED INPUT BLOCKS

When using patterned meshes (e.g., assemblies, cores), users sometimes reported that they needed to nearly duplicate multiple blocks of input to achieve their desired mesh. These nearly duplicated input blocks contained tiny differences related to ID changes. This resulted in excessively long input files and introduced the likelihood of user errors. Two improvements were made this year to streamline the input files for patterned meshes.

4.1.1 Multiple Reporting ID Assignment in Patterned Mesh Generators

When creating patterned meshes (e.g., repeated assemblies or pins), MOOSE offers the option of automatically assigning reporting IDs to different zones (e.g., pin ID, assembly IDs, plane IDs). Specifically, the patterned mesh generators (`PatternedHexMeshGenerator`, `PatternedCartesianMeshGenerator`) were originally designed to assign a single reporting ID based on the user's selected assignment scheme. For example, the reporting ID at the assembly level was only permitted to be one of the following assignment schemes:

- *Cell*: Assign unique reporting ID values for each cell (pin) in the lattice in sequential order.
- *Pattern*: Assign ID values based on the input cell types.
- *Manual*: Assign ID values based on user-defined mapping.

However, MOOSE did not conveniently support the assignment of multiple reporting IDs at the same level (e.g., pin ID and ring ID both assigned at the pin level), so blocks of input had to be copied and pasted throughout the input file with only slight syntax changes, causing excessive length and potential for errors. For example, in fuel cycle analysis, multiple assembly level reporting IDs are useful for tracking both assembly type and movement of assemblies across cycles. As another example, postprocessing also often requires multiple reporting IDs to easily extract solution quantities. By defining reporting IDs for each pin location and pin type, users can postprocess pin-wise distributions and quantities from specific pin types

without setting up complicated post processors. Therefore, the reporting ID capabilities in the patterned mesh generators were extended to support multiple reporting ID assignments.

Multiple reporting IDs can now be assigned in a single input block by providing an array of reporting ID names in the reporting ID variable *id_name*. Corresponding assignment schemes should be provided as an array of the same length in the *assign_type* parameter. The *id_pattern* parameter must contain manual ID patterns that correspond to each *manual* assignment scheme specified in *assign_type*.

Figure 16 demonstrates the new multiple reporting ID assignment capability for a hexagonal assembly. In this case, four IDs are assigned simultaneously at the pin level: *pin_id*, *pin_type_id*, *man_id_1*, *man_id_2*. The latter two are specified to be manual patterns in *assign_type*, and so two manual patterns are specified in *id_pattern*. The first manual ID *man_id_1* uses the first pattern defined in *id_pattern*, and the second manual ID *man_id_2* uses the second pattern defined in *id_pattern*.

```
[core]
  type = PatternedHexMeshGenerator
  pattern = '1 0 1;
             0 0 0 0;
            1 0 1 0 1;
             0 0 0 0;
              1 0 1'
  id_name ='pin_id
            pin_type_id
            man_id_1
            man_id_2'
  assign_type = 'cell
                 pattern
                 manual
                 manual'
  id_pattern = '2 2 2;
                2 1 1 2;
               2 1 0 1 2;
                2 1 1 2;
                 2 2 2|
                 1 1 1;
                2 2 2 2;
               3 3 3 3 3;
                4 4 4 4;
                 5 5 5'
[]
```

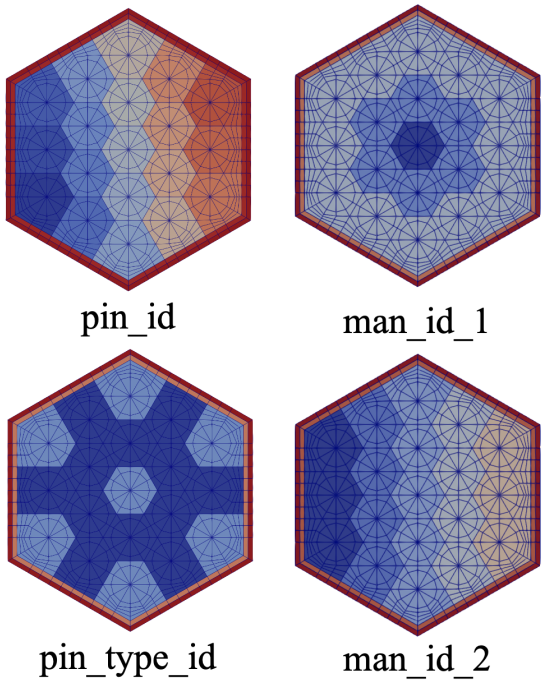


Figure 16. Multiple reporting ID Assignment in *PatternedHexMeshGenerator*.

4.1.2 Shifting Internal Sideset IDs in Patterned Mesh Generators

Motivated by liquid-metal fast reactor assembly load pad contact calculations, which require unique internal sidesets on each hexagonal duct, an option was added to the patterned mesh generators to reassign interface boundary IDs for each lattice cell component (generally a pin or assembly). This feature enables users to effortlessly create a lattice mesh with unique interface IDs on each pattern, thereby eliminating the need to duplicate input pattern meshes with different interface ID settings.

The interface boundary IDs for each lattice cell can now be shifted by specifying a 2D vector pattern in *interface_boundary_id_shift_pattern* matching the dimensions of *pattern*. The values in *interface_boundary_id_shift_pattern* are used to shift the standard interface ID of each lattice cell during the stitching process. This shifting procedure produces unique interface boundary ID values for each unit within the lattice.

Figure 17 demonstrates the shifting of interface IDs for a 7-assembly configuration in which the standard interface ID of the constituent assembly was 103. The stitched assemblies contain interface IDs of 1103, 2103, etc., rather than 103.

```
[core]
  type = PatternedHexMeshGenerator
  pattern = ' 0 0;
            0 0 0;
             0 0'
  interface_boundary_id_shift_pattern =
           '1000 2000;
          3000 4000 5000;
             6000 7000'
[]
```

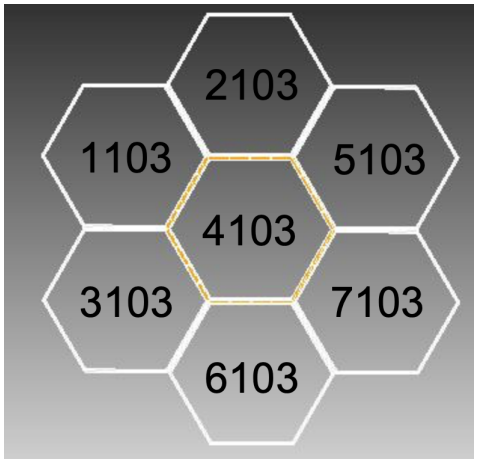


Figure 17. Shifting interface boundary IDs in *PatternedHexMeshGenerator*

4.2 QUADRATIC ELEMENT CAPABILITIES DEVELOPMENT

When the MOOSE Reactor Module tools were originally developed, only linear elements were supported for simplicity. As the functionalities and user base of the Reactor Module continue to expand, interest in enabling quadratic elements has emerged. In FY 2024, major efforts have been made to implement meshing capabilities using quadratic elements for all the MOOSE Reactor Module meshing tools that are commonly used for reactor core and reactor components meshing.

Notes on 2D Quadratic Elements

The quadrilateral quadratic elements include QUAD8 and QUAD9. Compared to its linear counterpart (QUAD4), a QUAD8 element has one midpoint node on each of its four sides (edges) to enable the use of quadratic shape functions. A QUAD9 element has an additional central node inside the element. This node helps capture more complexity and lower errors.

The most common triangular quadratic element is TRI6, which, compared to its linear counterpart (TRI3), also has one midpoint node on each of its three sides (edges). A TRI7 element has an additional central node inside the element. This node introduces third-order terms and makes TRI7 an incomplete third-order element. TRI7 is mostly useful either as a lower-D element in a 3D mesh or in some specialized contexts, but TRI6 is the element that is generally used for its own sake.

All the aforementioned elements have EDGE3 elements as their edges; that is, they can be stitched together to form meshes without creating hanging nodes.

4.2.1 PCCMG Based Generators

PolygonConcentricCircleMeshGenerator and its derived mesh generators, including *AdvancedConcentricCircleGenerator*, *CartesianConcentricCircleAdaptiveBoundaryMeshGenerator*, and *HexagonConcentricCircleAdaptiveBoundaryMeshGenerator*, have been upgraded to support quadratic elements. The specification of linear or quadratic elements is controlled by two input parameters: "tri_element_type" (options: TRI3, TRI6, and TRI7) and "quad_element_type" (options: QUAD4, QUAD8, and QUAD9). TRI3 and QUAD4 specify linear elements and the others specify quadratic elements. An example mesh of quadratic elements generated by *PolygonConcentricCircleMeshGenerator* can be found in Figure 18.

```
[gen]
  type = ↩
      PolygonConcentricCircleMeshGenerator
  num_sides = 5
  num_sectors_per_side = '4 4 4 4 4'
  background_intervals = 2
  polygon_size = 5.0

  background_block_ids="30 35"
  background_block_names="back_in back_out↩
      "
  tri_element_type=TRI7
  quad_element_type=QUAD9

  preserve_volumes = on
  external_boundary_id = 9999
  external_boundary_name = 'polygon_out'
[]
```

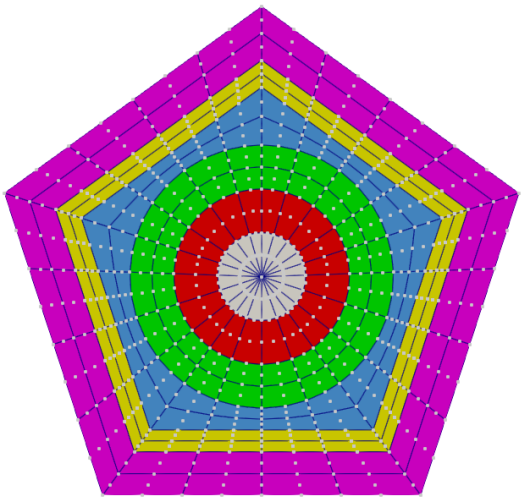


Figure 18. Quadratic element meshing in *PolygonConcentricCircleMeshGenerator*

The use of *AdvancedConcentricCircleGenerator* is similar to *PolygonConcentricCircleMeshGenerator* because they do not require other mesh generator inputs to function. However, both *CartesianConcentricCircleAdaptiveBoundaryMeshGenerator* and *HexagonConcentricCircleAdaptiveBoundaryMeshGenerator* depend on an input mesh that is then used for adaptive meshing. Therefore, in addition to the two input parameters used to control the element order, the orders of the input meshes defined by "meshes_to_adapt_to" must be consistent with the designated element types (Figure 19 shows an example).

4.2.1.1 Circular Radius Correction for Volume Preservation

A noteworthy subtopic for further discussion is the radius correction for quadratic elements to preserve circular areas (and volumes after extrusion). The radius correction is achieved by applying a correction factor (f_{corr}) on the radius of each circular region to ensure that the meshed circular region with corrected radius ($r_{corr} = f_{corr} \cdot r_{original}$) has the same area as the perfect circle with the original radius ($r_{original}$), as shown in the Eq. 1:

$$S_{mesh}\left(f_{corr} \cdot r_{original}\right) = S_{circle}\left(r_{original}\right) = \pi r_{original}^2, \tag{1}$$

where $S_{mesh}(r)$ is the actual area of the mesh as a function of the radius. $S_{mesh}(r)$ is the summation of the areas of all the linear and quadratic elements that construct the mesh. Because the area function is proportional to square of radius ($S_{mesh}(r) \propto r^2$), the correction factor f_{corr} can be deduced as follows:

$$f_{corr} = \sqrt{\frac{S_{circle}\left(r_{original}\right)}{S_{mesh}\left(r_{original}\right)}}. \tag{2}$$

For linear elements, correcting the polygonization effect is straightforward because the area of a polygonized area is simply the summation of a series of isosceles triangles. However, 2D quadratic elements have sides (each defined by two vertices and one midpoint) with quadratic shapes rather than polygonized shapes. Consequently, the error in an uncorrected circular mesh consisting of quadratic elements is much

```
[gen]
  type = ↩
      HexagonConcentricCircleAdaptiveBoundaryMeshGenerator
  num_sectors_per_side = '4 4 4 4 4 4'
  background_intervals = 2
  hexagon_size = 5.0
  sides_to_adapt = 0
  meshes_to_adapt_to = 'fmg'
  tri_element_type = TRI7
  quad_element_type = QUAD9
[]
```

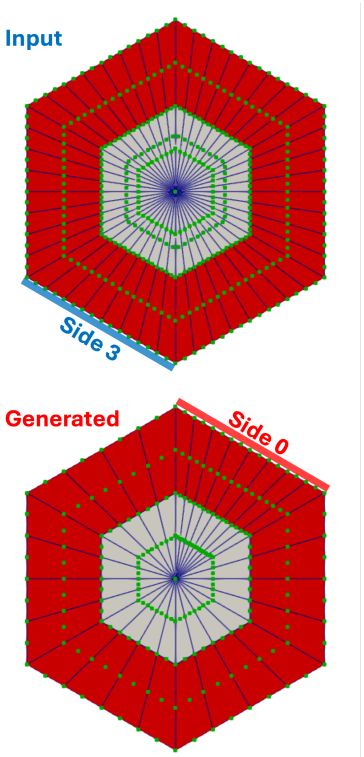


Figure 19. Use of *HexagonConcentricCircleAdaptiveBoundaryMeshGenerator* to generate a mesh with its Side 0 adapting to Side 3 of an input quadratic mesh.

smaller than that of a similar mesh with linear elements. Nevertheless, the quadratic side still deviates from a true circular boundary, necessitating correction.

In this context, a full or partial circle consisting of TRI6 elements, with the center of the circle as one of their vertices, is used to calculate the area of the quadratic element mesh. The volume attribute of the TRI6 element in libMesh is directly utilized for area calculation, allowing for the production of a correction factor. This approach is also applied to other mesh generators that involve volume preservation (also discussed in this section).

4.2.1.2 Azimuthal Block Splitting

The *AzimuthalBlockSplitGenerator* is used to modify a mesh generated by *PolygonConcentricCircleMeshGenerator* and its derived generators. The mesh generator has been upgraded to support quadratic elements. The generator automatically detects the element order of the input mesh and applies the corresponding splitting algorithm so that no additional input is needed (Figure 20 shows an example).

```
[Mesh]
  [cd]
    type = ↩
        PolygonConcentricCircleMeshGenerator
    num_sides = 6
    num_sectors_per_side = '4 4 4 8 4 4'
    background_intervals = 1
    ring_radii = '4.2 4.8'
    ring_intervals = '2 1'
    ring_block_ids = '10 15 20'
    ring_block_names = 'center_tri center ↩
        cd_ring'
    background_block_ids = 30
    background_block_names = background
    polygon_size = 5.0
    preserve_volumes = true
    tri_element_type = TRI6
    quad_element_type = QUAD9
  []
  [cd_azi_define]
    type = AzimuthalBlockSplitGenerator
    input = cd
    start_angle = 280
    angle_range = 100
    old_blocks = '10 15 20'
    new_block_ids = '100 150 200'
    new_block_names = 'center_tri_new ↩
        center_new cd_ring_new'
    preserve_volumes = true
  []
[]
```

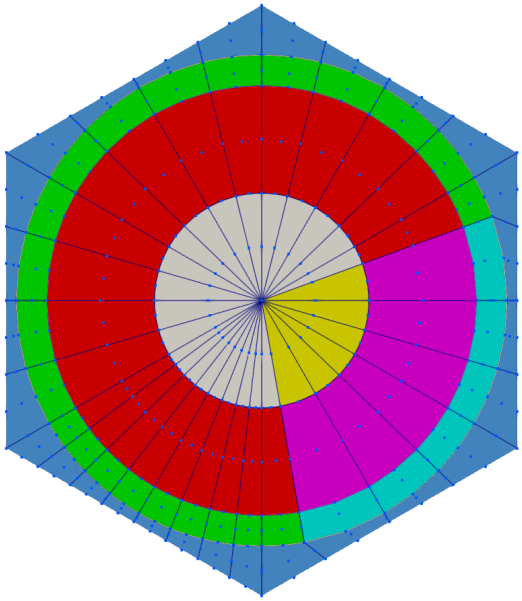


Figure 20. Block splitting performed by *AzimuthalBlockSplitGenerator*.

4.2.2 Patterned Hex/Cartesian Generators

PatternedHexMeshGenerator and *PatternedCartesianMeshGenerator* have been upgraded to support quadratic elements. Whether linear or quadratic elements are generated is controlled by a single input parameter, "boundary_region_element_type" (options: QUAD4, QUAD8, and QUAD9). An example quadratic element mesh generated by *PatternedHexMeshGenerator* can be found in Figure 21. An example mesh of quadratic elements generated by *PatternedCartesianMeshGenerator* can be found in Figure 22.

```
[hex_1]
  type = ↩
      PolygonConcentricCircleMeshGenerator
  tri_element_type=TRI6
  quad_element_type=QUAD8
  ...
[]
[pattern]
  type = PatternedHexMeshGenerator
  inputs = 'hex_1'
  background_intervals=2
  hexagon_size=18
  duct_sizes=17
  duct_intervals=1
  uniform_mesh_on_sides=true
  boundary_region_element_type=QUAD8
  pattern = '0 0;
            0 0 0;
             0 0'
[]
```

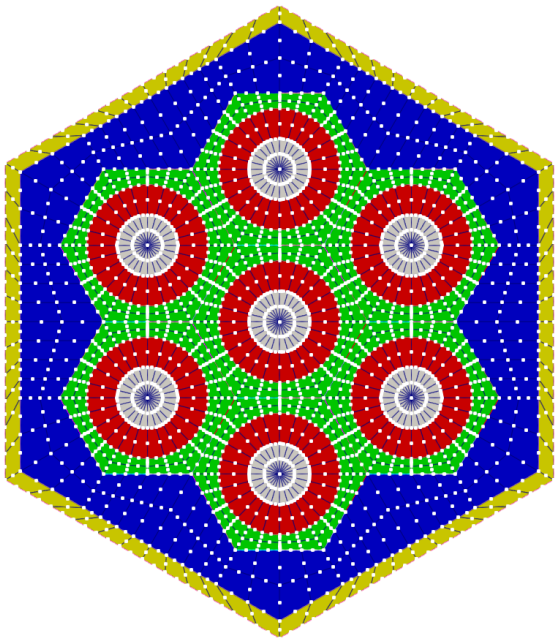


Figure 21. Quadratic elements meshing in *PatternedHexMeshGenerator*.

```
[square_1]
  type = ↩
      PolygonConcentricCircleMeshGenerator
  num_sides = 4
  tri_element_type=TRI6
  quad_element_type=QUAD8
  ...
[]
[square_2]
  type = ↩
      PolygonConcentricCircleMeshGenerator
  num_sides = 4
  tri_element_type=TRI6
  quad_element_type=QUAD8
  ...
[]
[pattern]
type = PatternedCartesianMeshGenerator
inputs = 'square_1 square_2'
square_size=44
duct_sizes=21
duct_intervals=2
uniform_mesh_on_sides=true
boundary_region_element_type=QUAD8
pattern = '0 0 0 0;
           0 1 0 0;
           0 0 1 0;
           0 0 0 0'
[]
```

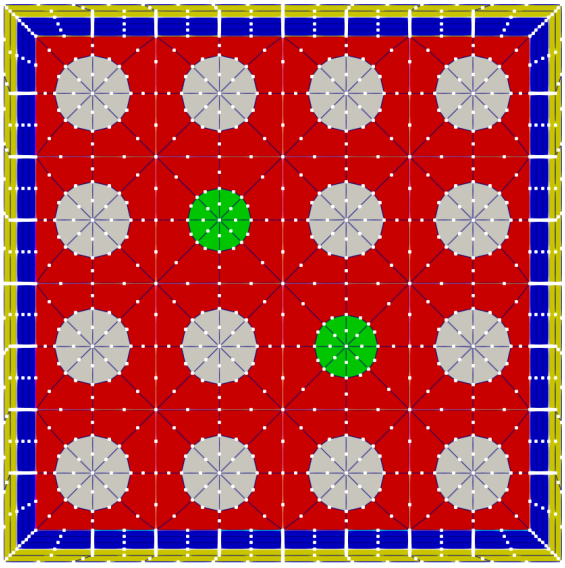


Figure 22. Quadratic elements meshing in *PatternedCartesianMeshGenerator*.

4.2.3 Patterned Hex/Cartesian Periphery Modifiers

PatternedHexPeripheralModifier and *PatternedCartesianPeripheralModifier* utilize *FillBetweenPointVectorsTools* to replace the outmost layer of the quad elements of the 2D hexagonal/cartesian assembly mesh with a transition layer consisting of triangular elements so that the assembly mesh can have nodes on designated positions on the external boundary. This boundary modification facilitates the stitching of assemblies that have different node numbers on their outer periphery as a result of differing numbers of interior pins and/or different azimuthal discretization.

The quadratic element option has been enabled for these two mesh generators. The element order is automatically determined by the element order of the input mesh. Because *FillBetweenPointVectorsTools* only supports linear meshes, a linear transition layer mesh is first generated before being converted into a second-order mesh by adding midpoints on the corresponding geometric centers. Then, the midpoints on the boundary subject to stitching are further adjusted to match the input mesh to facilitate stitching. An example of using *PatternedHexPeripheralModifier* to modify a quadratic mesh is provided in Figure 23.

```
[Mesh]
  [hex]
    type = ↩
        PolygonConcentricCircleMeshGenerator
    num_sides = 6
    num_sectors_per_side = '2 2 2 2 2 2'
    background_intervals = 1
    ring_radii = 4.0
    ring_intervals = 2
    ring_block_ids = '10 15'
    background_block_ids = 20
    polygon_size = 5.0
    preserve_volumes = on
    quad_element_type = QUAD8
    tri_element_type = TRI6
  []
  [pattern]
    type = PatternedHexMeshGenerator
    inputs = 'hex'
    pattern = '0 0;
              0 0 0;
               0 0'
    background_intervals = 2
    hexagon_size = 15
    boundary_region_element_type = QUAD8
  []
  [pmg]
    type = PatternedHexPeripheralModifier
    input = pattern
    input_mesh_external_boundary = 10000
    new_num_sector = 10
    num_layers = 2
  []
[]
```

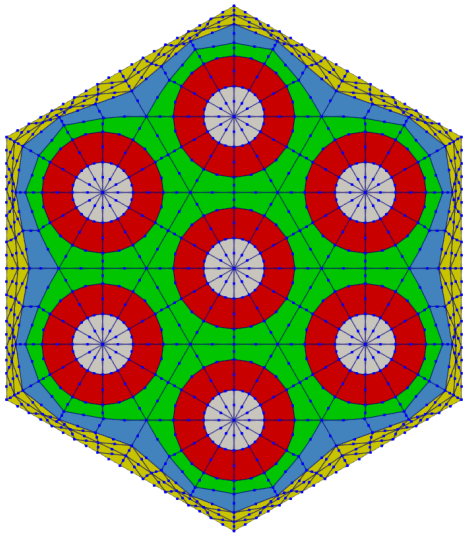


Figure 23. A quadratic mesh with peripheral region modified by *PatternedHexPeripheralModifier*.

4.2.4 Peripheral Rings

PeripheralRingMeshGenerator has been upgraded to support quadratic element meshing. The element order is automatically determined by the order of the input mesh elements on its external boundary. If the

external boundary consists of pure EDGE2 elements, QUAD4 (linear) elements are used to construct the peripheral ring mesh. On the other hand, if the input mesh has EDGE3 elements on its external boundary, QUAD9 (quadratic) elements are generated. Note that mixed element order is NOT supported. An example of using *PeripheralRingMeshGenerator* to generated a peripheral ring mesh using QUAD9 elements for an quadratic input mesh is provided in Figure 24.

```
[Mesh]
  [pccmg]
      type = PolygonConcentricCircleMeshGenerator
      num_sides = 8
      num_sectors_per_side = '4 4 4 4 4 4 4 4'
      background_intervals = 2
      polygon_size = 5.0

      preserve_volumes = on

      tri_element_type = TRI6
      quad_element_type = QUAD8
  []
  [pr]
      type = PeripheralRingMeshGenerator
      input = pccmg
      peripheral_layer_num = 4
      peripheral_ring_radius = 15
      input_mesh_external_boundary = 10000
      peripheral_ring_block_id = 250
      peripheral_ring_block_name = reactor_ring

      peripheral_inner_boundary_layer_bias = 2.0
      peripheral_inner_boundary_layer_intervals = 3
      peripheral_inner_boundary_layer_width = 1

      peripheral_outer_boundary_layer_bias = 0.5
      peripheral_outer_boundary_layer_intervals = 3
      peripheral_outer_boundary_layer_width = 1

      peripheral_radial_bias = 1.5
  []
[]
```

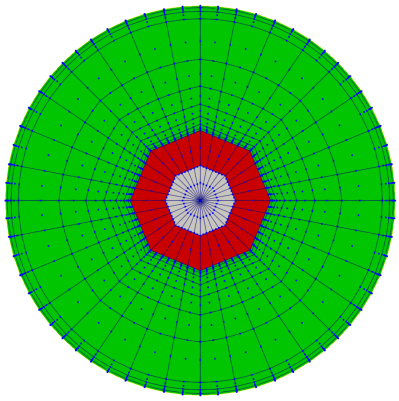


Figure 24. Quadratic element mesh generated by *PeripheralRingMeshGenerator*.

4.2.5 XYDelaunay Generator and the Derived Generators

In previous years, NEAMS developed a set of triangular mesh generation tools that have since become an indispensable part of many workflows. The libMesh `Poly2TriTriangulator` class uses the third-party triangulator Poly2Tri, along with original triangulation adaptive refinement code, to allow for the automatic generation of a triangular mesh within a user-defined connected 2D domain. The MOOSE `XYDelaunayMeshGenerator` class integrates this triangulation capability with the MOOSE mesh generator system, with boundaries and holes defined by inputs from previous mesh generators, mesh generation and adaptivity option control via MOOSE input parameters, and stitching of hole meshes into the holes they define. Lastly, the MOOSE mesh "subgenerator" API allows for complex mesh generators, such as the MOOSE Reactors module, to combine complex subsystems into a single and much simpler task-specific user interface.

One flaw in this system, however, grew out of a typical shortcoming of mesh triangulation code: none of the above capabilities supported anything other than linear triangles (using first-order inputs that might also involve linear edges or at most bilinear quadrilaterals). Meshes with second-order (quadratic, bi-quadratic, etc.) elements have always been supported in libMesh and MOOSE, and getting higher-order domain approximation in this way can be critical for calculating accurate results on even the simplest of curved geometries. However, this was unachievable for geometries that required unstructured triangle meshing. Why mesh a circular subdomain cross section with a curved boundary if doing so would prevent the gaps between those subdomains from being meshed afterward?

The following sections discuss the upgrade of both libMesh and MOOSE to enable quadratic element meshing capabilities in the general purpose triangular mesh generation tools.

4.2.5.1 libMesh Updates to Enable Quadratic Element Capabilities for XYDelaunay

Because MOOSE supports the composition of outputs from many different mesh generator objects, the first part of the solution to this problem was to ensure that different mesh generator codes (many of which had also been written to only support first-order geometric elements) could be upgraded to support second-order output, one by one, without creating incompatibilities between them. Support for mixed-order meshes at the data structure level was added in libMesh PR #3752, which added mixed-order capability to the code responsible for maintaining mesh topology by reconstructing neighbor links between elements, as well as to the code responsible for stitching multiple meshes together into one.

To produce a higher-order mesh from a lower-order one, the libMesh methods `all_second_order()` and `all_complete_order()` are used. These methods were designed to begin with input of consistent order, but after libMesh PR #3707, they support inputs of mixed element order. These methods upgrade elements selectively as necessary to produce the requested consistent output order in the mesh afterward while still respecting any existing higher-order geometry in the input.

The third low-level change, libMesh PR #3816, added support for mixed-order inputs to our triangulation code, allowing triangulations with quadratic boundary triangles to be generated to match curved boundary inputs.

Lastly, MOOSE PR #27546 added MOOSE interfaces, regression tests, and documentation for all of these new capabilities. A new `ElementOrderConversionGenerator` class gives users the ability to manually upgrade (or downgrade, for completeness) their geometric element type. The `XYDelaunayGenerator` now gives users control over element order there, and this control extends upward to Reactor Module mesh generators that use the unstructured triangulation capability internally.

4.2.5.2 Quadratic Element Capabilities in XYDelaunayGenerator and Derived Mesh Generators

The aforementioned updates on the libMesh side were adopted by the `XYDelaunayGenerator` in MOOSE. The order of the elements to be generated is controlled by the input parameter "tri_element_type" (options: TRI3, TRI6, TRI7, DEFAULT). Currently, the DEFAULT is the same as the TRI3 option. In the future, automation in mesh order determination might be added. By setting "tri_element_type", `XYDelaunayGenerator` can generate triangulation mesh using the specified elements.

When generating a mesh of TRI3 elements for first-order boundary and hole meshes, or a mesh of TRI6/TRI7 elements for second-order boundary and hole meshes, the process is straightforward. However, if the boundary and hole meshes are first-order (or mixed order) and “tri_element_type” is set to TRI6 or TRI7, the second-order mesh is created by placing midpoints at the centers of the element sides. In this case, if a first-order hole mesh needs to be stitched to the second-order triangulation mesh, the first-order hole mesh is converted to a second-order mesh before stitching. Conversely, if the boundary and hole meshes are second-order (or mixed order) and “tri_element_type” is set to TRI3, the first-order mesh is generated by ignoring the midpoints in the boundary and hole meshes. Attempting to stitch a second-order hole mesh to

the first-order triangulation mesh results in an error message. In Figure 25, a quadratic triangulation mesh (TRI7) is generated based on a linear boundary mesh and a quadratic hole mesh that is eventually stitched.

```
[Mesh]
  [fmg]
    # a second-order hole
    type = FileMeshGenerator
    file = accg_one_layer_quadratic.e
  []
  [ext]
    # a first-order boundary
    type = PolyLineMeshGenerator
    points = '-4.0 0.0 0.0
              0.0 -4.0 0.0
              4.0 0.0 0.0
              0.0 4.0 0.0'
    loop = true
  []
  [xyd]
    type = XYDelaunayGenerator
    boundary = 'ext'
    holes = 'fmg'
    stitch_holes = 'true'
    refine_holes = 'false'
    verify_holes = 'false'
    add_nodes_per_boundary_segment = 2
    refine_boundary = true
    desired_area = 1.0
    tri_element_type = TRI7
  []
[]
```

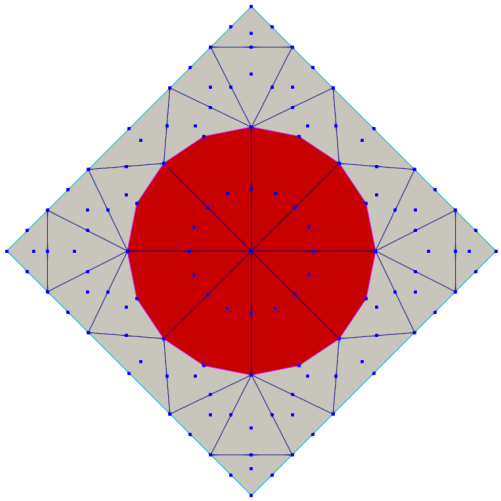


Figure 25. Quadratic mesh generated by `XYDelaunayGenerator`.

The same input parameter "tri_element_type" and the element order rules are inherited by the derived mesh generators of `XYDelaunayGenerator`, including `PeripheralTriangleMeshGenerator` and `FlexiblePatternGenerator`.

4.2.6 Summary of Quadratic Element Meshing Capabilities

After the systematic upgrades of the Reactor Module tools to support meshing with quadratic elements in FY 2024, MOOSE's intrinsic meshing tools are capable of performing general meshing tasks using quadratic elements for reactor designs that do not involve complex and nontrivial geometry features. For instance, the full-core heat-pipe microreactor (HP-MR) mesh available on the Virtual Test Bed (G. L. Giudicelli et al. 2023; Stauff et al. 2024) can be generated using quadratic elements, as shown in Figure 26.

4.3 MESH REVOLUTION CAPABILITIES

The *RevolveGenerator* was developed to enable revolution of a 1D mesh into 2D, or a 2D mesh into 3D, by revolving the input mesh around a user-defined axis. This new mesh generator provides an alternative tool for increasing the dimensionality of a lower-dimension mesh (1D or 2D) in addition to *MeshExtruderGenerator*/*AdvancedExtruderGenerator*. Each element is converted to one or more copies of its corresponding higher-dimensional element along an open or closed specific circular curve. Inspired by *AdvancedExtruderGenerator*, *RevolveGenerator* provides similar styles of enriched features such as subdomain/boundary ID swap and extra element integers swap.

Revolving is a useful functionality for 3D meshing of components with axisymmetry, such as a reactor

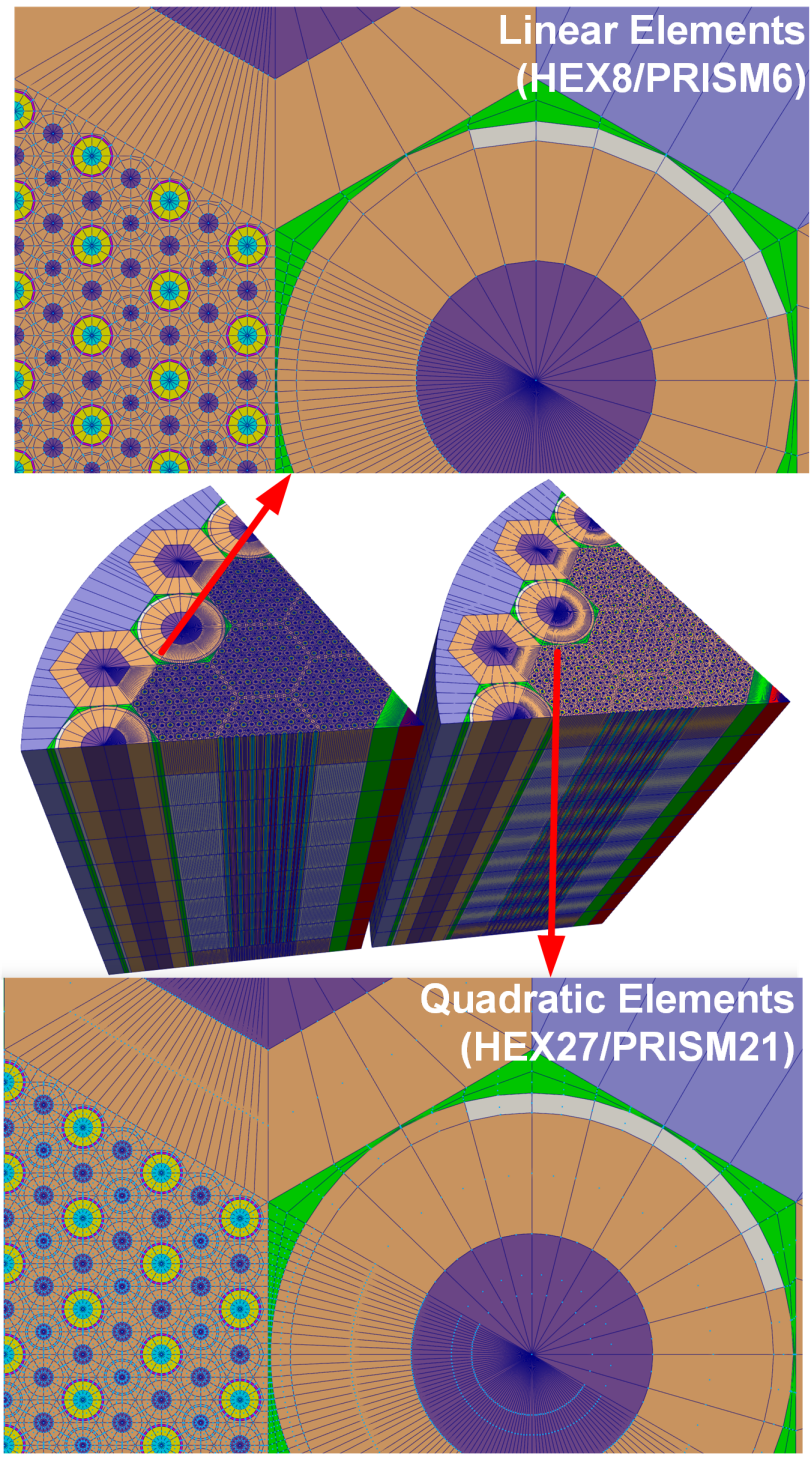


Figure 26. Linear and quadratic HP-MR meshes generated by MOOSE Reactor Module.

dome, a fuel rod, and a tokamak magnet. Revolving is a powerful tool for generating a 3D mesh based on an existing 2D-RZ mesh.

```
[Mesh]
  [shg]
    type = SimpleHexagonGenerator
    hexagon_size = 1.0
    block_id = '100 105'
    block_name = 'hex_tri hex'
    element_type = HYBRID
    radial_intervals = 2
  []
  [rg]
    type = RevolveGenerator
    input = shg
    axis_point = '5.0 0.0 0.0'
    axis_direction = '0.0 1.0 0.0'
    nums_azimuthal_intervals = 32
  []
[]
```

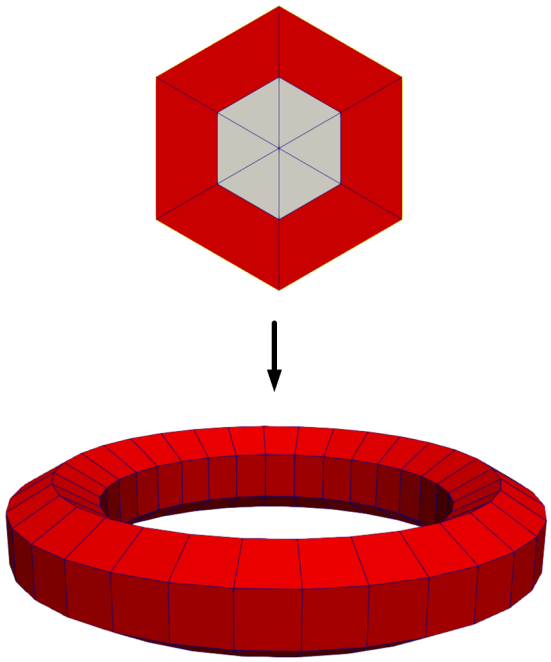


Figure 27. Simple full-circle mesh revolution produced by *RevolveGenerator*.

4.3.1 Basic Revolving Functionality

RevolveGenerator revolves a lower-dimension mesh (1D or 2D) given by "input" into a higher-dimension mesh (2D or 3D) along an revolving axis defined by "axis_point" and "axis_direction". *RevolveGenerator* supports both linear and quadratic elements. Because revolution results in circular geometry features, "volume_preserve" is an option for both linear and quadratic element order options.

By default, mesh revolution is performed along a full closed circular curve (i.e., 360 degrees) with a single azimuthal section with uniform azimuthal discretization (as shown in Figure 27). Optionally, mesh revolution can be performed along a partial circular curve (as shown in Figure 28), and multiple azimuthal sections can be defined each with unique discretization (as shown in Figure 29). These options can be selected by specifying "revolving_angles." As long as the summation of the angles listed in "revolving_angles" is 360 degrees, a full closed circular revolution is performed. Otherwise, a partial revolution is performed. Both partial and full revolutions with multiple azimuthal sections can be performed in either the clockwise or counterclockwise directions, as controlled by "clockwise."

Each azimuthal section can have separately defined subdomains, extra element integers, and boundaries. The number of azimuthal elements in the different sections can be provided through "nums_azimuthal_intervals."

RevolveGenerator is also capable of handling the special condition in which some original mesh nodes are located on the revolving axis, as indicated in Figure 30.

4.3.2 ID Remapping/Swap

In *RevolveGenerator*, three types of IDs from the input mesh can be remapped/swapped during revolving: subdomain IDs, extra element integer IDs, and boundary IDs.

By default, the revolved higher-dimension elements retain the same subdomain IDs as their original lower-

```
[Mesh]
  [shg]
    type = SimpleHexagonGenerator
    hexagon_size = 1.0
    block_id = '100 105'
    block_name = 'hex_tri hex'
    element_type = HYBRID
    radial_intervals = 2
  []
  [rg]
    type = RevolveGenerator
    input = shg
    axis_point = '5.0 0.0 0.0'
    axis_direction = '0.0 1.0 0.0'
    nums_azimuthal_intervals = 32
    revolving_angles = 175
  []
[]
```

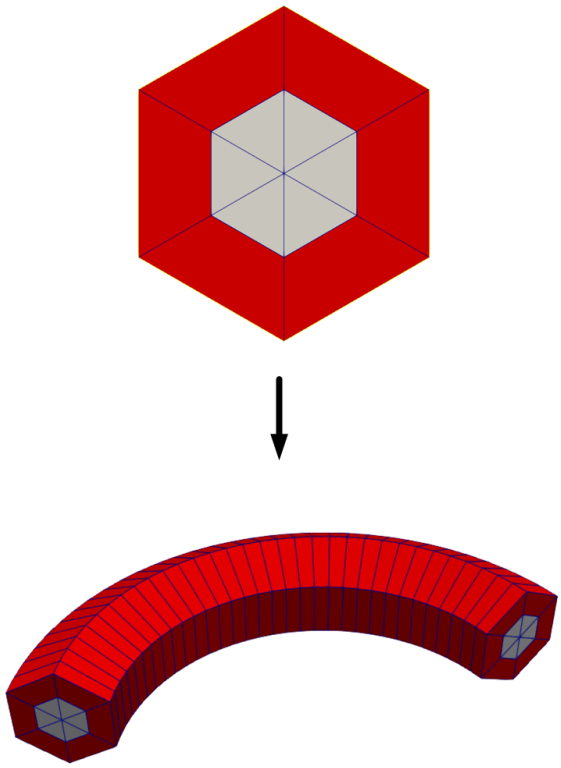


Figure 28. Simple partial-circle revolving in *RevolveGenerator*.

```
[Mesh]
  [shg]
    type = SimpleHexagonGenerator
    hexagon_size = 1.0
    block_id = '100 105'
    block_name = 'hex_tri hex'
    element_type = HYBRID
    radial_intervals = 2
  []
  [rg]
    type = RevolveGenerator
    input = shg
    axis_point = '5.0 0.0 0.0'
    axis_direction = '0.0 1.0 0.0'
    nums_azimuthal_intervals = '4 8 16 32'
    revolving_angles = '20 40 60 80'
    subdomain_swaps = ' ;
                      100 200 105 205;
                      100 300 105 305;
                      100 110'
  []
[]
```

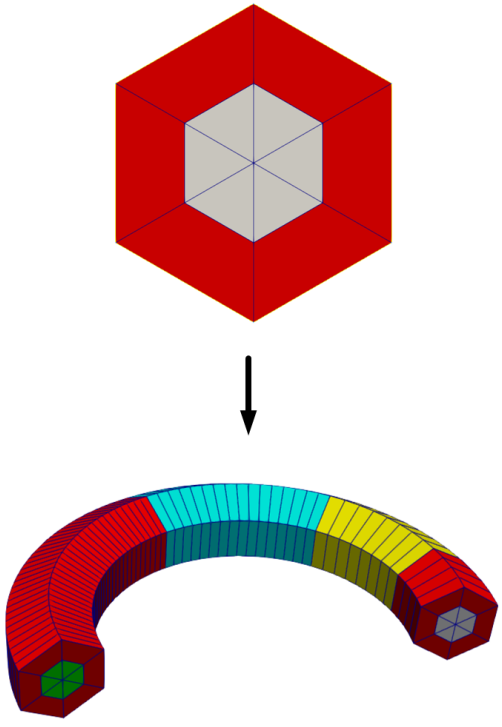


Figure 29. Multilayered partial-circle revolving with subdomain ids swap in *RevolveGenerator*.

```
[Mesh]
  [gmg]
    type = GeneratedMeshGenerator
    dim = 2
    nx = 5
    ny = 6
    x_max = 2.0
    y_max = 5.0
  []
  [rg]
    type = RevolveGenerator
    input = gmg
    axis_point = '0.0 0.0 0.0'
    axis_direction = '0.0 1.0 0.0'
    nums_azimuthal_intervals = 32
  []
[]
```

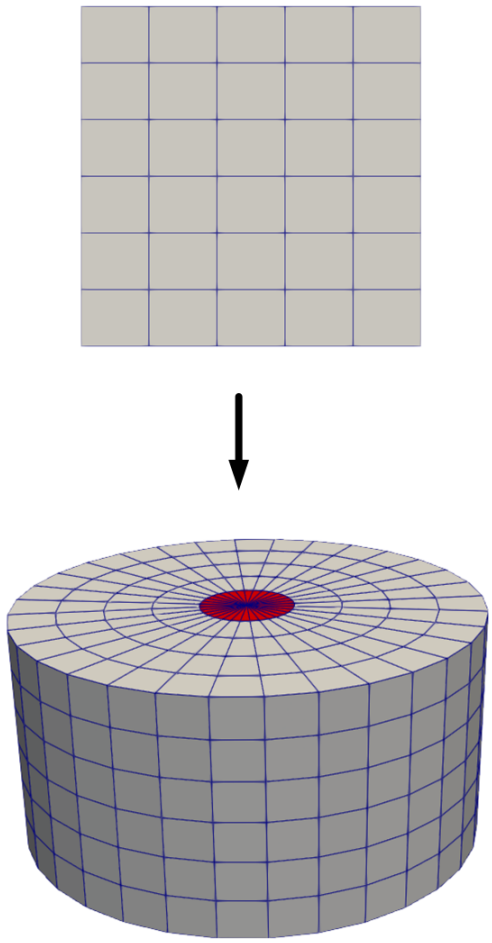


Figure 30. Revolving with on-axis nodes in *RevolveGenerator*.

dimension elements. *RevolveGenerator* provides an option to remap subdomain IDs for each azimuthal section through "subdomain_swaps," which is a double indexed array input parameter. Each elemental vector of "subdomain_swaps" contains subdomain remapping information for a particular elevation, where the first elemental vector represents the first revolved azimuthal section. The elemental vector contain pairs of subdomain IDs: the first subdomain ID is the input mesh subdomain ID that needs to be remapped, and the second subdomain ID the new subdomain ID to be assigned.

Extra element integer ID remapping works in a similar manner as subdomain ID remapping. The extra element integers to be remapped must already exist in the input mesh and need to be specified in "elem_integer_names_to_swap." The remapping information of multiple extra element integers is provided as a triple-indexed array input parameter ("elem_integers_swaps"). For each extra element integer, the syntax is similar to "subdomain_swaps." The following input example shows the remapping of two extra element integers.

```
[Mesh]
  [rg]
    type = RevolveGenerator
    input = fmg
    axis_point = '5.0 0.0 0.0'
    axis_direction = '0.0 1.0 0.0'
    nums_azimuthal_intervals = '2 4 2'
    revolving_angles = '30 60 30'
    elem_integer_names_to_swap = 'element_extra_integer_1 element_extra_integer_2'
    elem_integers_swaps = '1 4 2 8;
                           2 8;
                           2 7 |
                           1 8 2 4;
                           2 4;
                           2 5'
  []
[]
```

Boundary ID remapping also works similarly to subdomain ID remapping. During revolution, the lower-dimension boundaries are converted into higher-dimension boundaries. A double indexed array input parameter, "boundary_swaps," can be used to remap the boundary IDs. Here, the boundary IDs to be remapped must exist in the input mesh; otherwise, dedicated boundary defining mesh generators, such as *SideSetsBetweenSubdomainsGenerator* and *SideSetsAroundSubdomainGenerator*, need to be used to define new boundary IDs along different azimuthal sections.

4.4 3D MESH CUTTING

The *CutMeshByPlaneGenerator* is basically the 3D version of *XYMeshLineCutter*, which was developed in FY2023. *CutMeshByPlaneGenerator* is used to slice a 3D input mesh along a given plane and discard the portion of the mesh on one side of the plane. The input mesh, given by "input", must be 3D and contain only first-order (i.e., linear) elements. The cutting plane is specified by "plane_normal" and "plane_point," which are the two points that represent the normal vector of the cutting plane and a point located on the cutting plane, respectively. This mesh generator removes the part of the mesh located on the side of the plane in the direction of the normal vector. The mesh is then smoothed to ensure a straight cut (instead of a "zigzag" cut) along element boundaries as generated by *PlaneDeletionGenerator*, facilitating the follow-up meshing steps and/or simulation.

CutMeshByPlaneGenerator can be used to cut a mesh to take advantage of symmetry when the original mesh does not have element faces aligned on the cutting boundary.

```
[convert]
  type = ElementsToTetrahedronsConverter
  input = extrude
[]
```

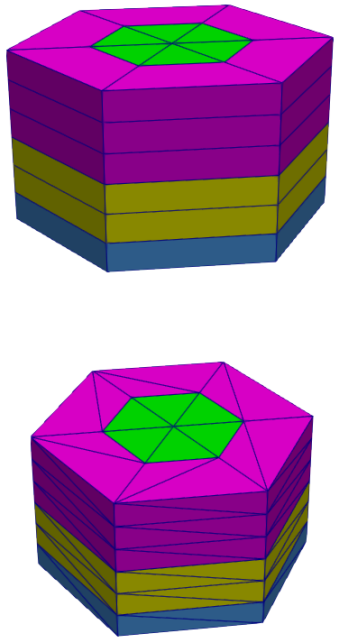


Figure 31. Tetrahedralization done by *ElementsToTetrahedronsConverter*.

4.4.1 Tetrahedron Conversion

Cutting an element involves complex algorithms. For simplicity, in *CutMeshByPlaneGenerator*, the input mesh is first converted into a mesh that only consists of TET4 elements so that only the cutting algorithm for TET4 elements needs to be implemented. Tetrahedralization is achieved by splitting non-tetrahedron linear elements—including HEX8, PRISM6, and PYRAMID5—into TET4 elements. Specifically, each HEX8 element is split into six TET4 elements, each PRISM6 element is split into three TET4 elements, and each PYRAMID5 element is split into two TET4 elements. Note that the same tetrahedralization algorithm has also been used to build a standalone mesh generator named *ElementsToTetrahedronsConverter* (as shown in Figure 31).

4.4.1.1 Splitting of HEX8 Elements

There are multiple ways to split one HEX8 element into multiple TET4 elements, resulting in either five or six TET4 elements (or even more if additional nodes can be added). A splitting method that does not require adding nodes needs to split each of the six quadrilateral faces of a HEX8 element into two triangles, which can be done in two different ways for each face. Because these quadrilateral faces could be shared with neighboring HEX8 or other types of elements, the splitting of the quadrilateral faces on neighboring elements must be performed consistently. To achieve a consistent splitting approach (a topic discussed later in this documentation page) a HEX8 element needs to be split into six TET4 elements. An example of this splitting is illustrated in Figure 32. Note that the splitting approach shown in Figure 32 is not unique and will be discussed later.

4.4.1.2 Splitting of PRISM6 elements

A PRISM6 element can be split into three TET4 elements. An example of this splitting is illustrated in Figure 33. Note that the splitting approach shown in Figure 33 is not unique and will be discussed later. Namely, the three quadrilateral faces of a PRISM6 element need to be split consistently with the neighboring elements.

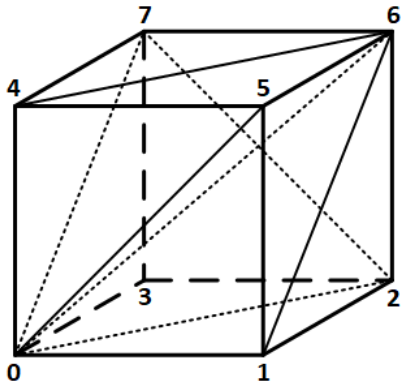


Figure 32. An example of splitting of a HEX8 element into six TET4 elements.

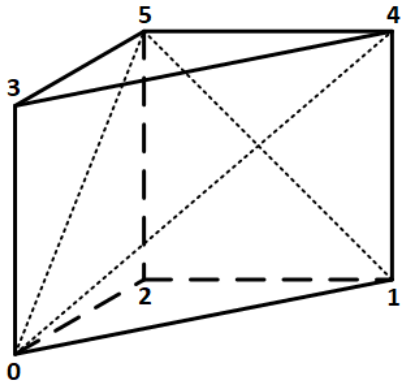


Figure 33. An example of splitting of a PRISM6 element into three TET4 elements.

4.4.1.3 Splitting of PYRAMID5 elements

A PYRAMID5 element can be split into two TET4 elements. An example of this splitting is illustrated in Figure 34. Note that the splitting approach shown in Figure 34 is not unique and will be discussed later. Namely, the one quadrilateral face of a PYRAMID5 element needs to be split consistently with the neighboring elements.

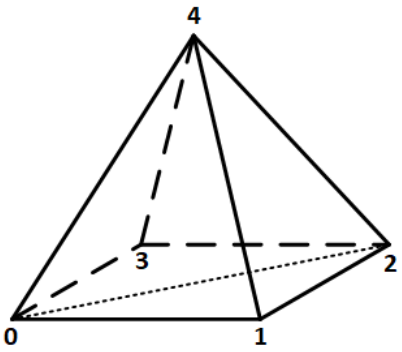


Figure 34. An example of splitting of a PYRAMID5 element into two TET4 elements.

4.4.1.4 Consistent Splitting for Neighboring Elements

As discussed above, although splitting of non-TET elements into TET4 elements is not unique, it is crucial to ensure that the splitting of the neighboring elements involves consistent splitting of the quadrilateral faces. To achieve this, the following approach is used. There are two ways to split each quadrilateral face into two triangles, which correspond to the two diagonal lines of the quadrilateral face. Therefore, to ensure that one of the two diagonal lines is selected consistently for all the elements, the diagonal line is selected that involves the node with the lowest global node ID among the four nodes of the quadrilateral face.

4.4.2 Tetrahedron Cutting

Once all the elements of the input mesh have been converted into TET4 elements, the cutting method only needs to be applied to TET4 elements. First, all the elements that are cut by the given cutting plane are identified. For the TET elements involved, their relationship with the cutting plane can be categorized into one of the six cases shown in Figure 35. For each of these six cases, new nodes are created at the intersection points between the cutting plane and the edges of the TET element. Then, the red part of the original TET element is removed, and new TET element(s) are created, as shown in Figure 35. The cross sections created by this cutting procedure are assigned a new boundary ID, as defined by "cut_face_id". Figure 36 provides an example of how *CutMeshByPlaneGenerator* can be utilized to cut an existing 3D mesh.

4.5 SUPPORT FOR MIXED-ORDER AND QUAD8 3D EXTRUSION

Axial extrusion is commonly used to create 3D reactor models from 2D layouts. The subdomains on each axial layer are reassigned to create an extruded geometry with nonuniform axial domains. Because of the development of second-order capabilities that support QUAD8 and QUAD9 elements in many reactor module generators, the `AdvancedExtruderGenerator` had to be upgraded to handle QUAD8 elements. The algorithm for extrusion had to be modified slightly because the mid-plane in a 20-node hexagon (the 3D equivalent of a QUAD8) only contains four nodes and not eight like the initial 2D shape. In the process, mixed-order (linear and quadratic) mesh extrusion was enabled, even though mixed-order 2D meshes are not currently supported within the Reactor Module.

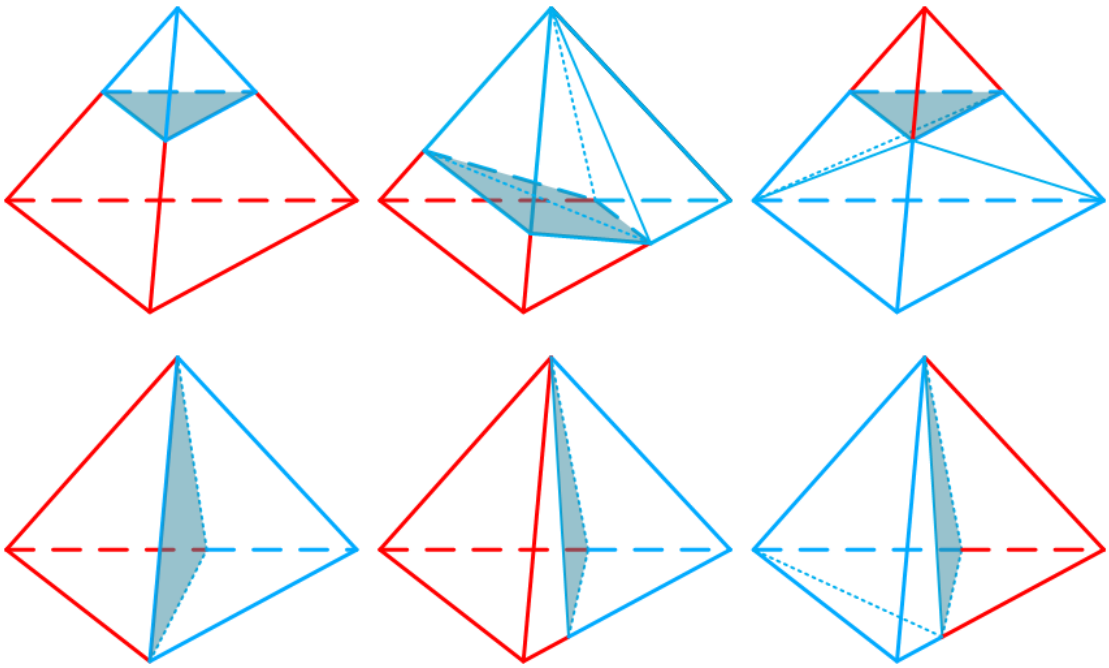


Figure 35. The six possible cases when slicing a TET element. The cutting plane intersection with the element is shown as blue faces. The red part of the original TET element is removed after cutting, while the blue part of the original TET element is kept and split into multiple TET elements if necessary.

```
[cut]
  type = CutMeshByPlaneGenerator
  input = extrude
  plane_point = '0 0 1.2'
  plane_normal = '1.0 1.0 2.0'
  cut_face_id = 12345
  cut_face_name = cut
[]
```

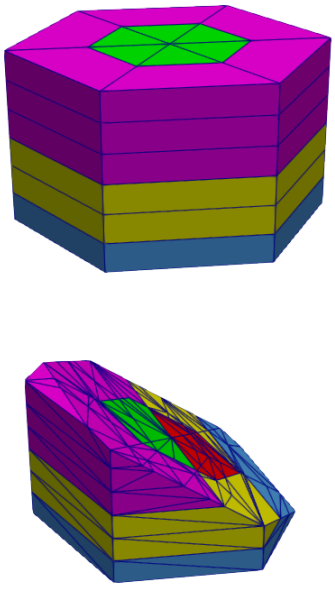


Figure 36. 3D plane cut performed by *CutMeshByPlaneGenerator*.

4.6 DEVELOPMENT STATUS OF 3D MESH TETRAHEDRALIZATION

Previous sections discuss recent improvements to 3D meshing in MOOSE. This section summarizes additional efforts to specifically incorporate the Netgen library into MOOSE for general tetrahedralization.

4.6.1 Netgen Integration Status

To build upon the success of the 2D unstructured triangulation support that has been added to (and now extended in) MOOSE, an attempt was made to integrate 3D unstructured triangulation (tetrahedralization) capabilities in a similar fashion. Like the 2D `XYDelaunayMeshGenerator`, a new 3D `XYZDelaunayMeshGenerator` would be able to take boundary and hole definitions from other MOOSE mesh generators, generate a mesh of simplices (triangles or tets) in the domain defined by those inputs, and stitch together input meshes to the new tetrahedral mesh, both metaphoricalliy and literally filling in the gaps left by other types of mesh generation. A volume mesh generated by `XYDelaunay` must begin with simple extrusions of 2D triangulations, but `XYZDelaunay` would be far more flexible.

The choice of software to integrate in this case was not an easy one. Delaunay tetrahedralization is not as straightforward as Delaunay triangulation, so implementing a from-scratch mesh generator would be unwise. Many tetrahedralization codes already exist, but software compatibility issues and/or licensing issues precluded their inclusion in MOOSE. For instance, a libMesh interface to the software TetGen has existed for a long time; however, this interface exists under a license that permits it to be redistributed and built for academic use but requires disabling the interface for the libMesh builds used by most MOOSE users. Another libMesh interface to QHull is license-compatible but lacks basic features that would be needed by a useful MOOSE `XYZDelaunay` tool.

Netgen, a featureful mesh generation and optimization library under the same lesser general public license as MOOSE, was ultimately chosen for this project. Netgen has a straightforward tetrahedralization API and has more advanced features that the team hoped might become accessible for future projects after the initial software work was completed for current use cases.

The primary concern was that other MOOSE developers and collaborators working with Netgen had experienced difficulty with direct software integration, likely due to the significant differences between Netgen's CMake-based build system and the make and autotools build systems used by MOOSE and libMesh. This was indeed a source of some difficulty, but as of libMesh PR #3793 it was solved. Netgen is now included in the libMesh codebase via a git submodule, and the libMesh configure command executes the Netgen CMake configuration as an optional dependency, enabled unless a system is lacking some Netgen dependency or unless a user specifically disables it. To make enabling the tetrahedralization support as easy as possible, the libMesh configuration of Netgen disables some of the broader Netgen features that would introduce additional third-party dependencies; thus, the only significant new dependency to enable Netgen is the `cmake` software itself.

As was done years ago in 2D, the new 3D mesh generation at the libMesh level followed a standard object-oriented pattern there: an abstract base class to expose the interface, with multiple backends to allow users to choose between different implementations depending on their license compatibility and feature needs. Most of the newly developed software for this project was factored to be independent of that choice of implementation; tools like boundary integrity testing (to validate user input), and converter tools to generate boundary triangulations from volumetric inputs, are done entirely within the base class code. Other base class utilities, such as support for automatic conversion of hexahedral into tetrahedral meshes, were written to support unit testing of the new tetrahedralization features in libMesh, but may be made user-accessible in MOOSE in the future.

A number of later modifications were made to the new libMesh tetrahedralization code, either as bugs were found "downstream" via MOOSE or as this new feature was made compatible with more and more

existing features in libMesh and MOOSE. Netgen is not an MPI-aware code, and libMesh PR #3926 makes its interface with MPI-parallelized execution a bit more efficient and robust. Netgen error messages triggered by invalid inputs are somewhat cryptic by MOOSE standards, and libMesh PR #3930 tries to intercept certain types of both software and user errors to either correct them or at least return more user-friendly error messages. Perhaps the greatest source of delay was a serious bug inside the Netgen code itself. This bug was triggered first by certain quirks of node ordering on distributed-memory parallel test runs. This was reported upstream along with a test case, and both the test code and the bug fixes in Netgen were recently incorporated into libMesh PR #3939.

Perhaps the largest development culture difference was fixed in libMesh PR #3940, which disables a configuration decision in Netgen that, while useful for optimization on an individual developer's machine, was causing serious incompatibilities when activated on the MOOSE build farm and distributed to MOOSE users on older CPU microarchitectures via the MOOSE conda binary packages.

The new tetrahedralization mesh generation in MOOSE is designed with features similar to the previous triangulation mesh generation. Inputs may be provided from other mesh generators in one of two formats: a direct specification of a boundary manifold as a mesh of lower-dimensional elements (edges for a triangulation, or triangles for tetrahedralization), or an indirect specification via a volumetric mesh whose outer boundary is interpreted as the domain boundary to consider. The direct specification is useful for mesh generators that work from some parametric representation of a boundary, whereas the indirect specification is compatible with the widest variety of existing mesh generators and is necessary for dealing with meshes that are used first to define "holes" in the new mesh and then stitched into those holes afterward. The feature set for basic mesh generation cases is now complete in MOOSE PR #28298.

4.6.2 Future Tetrahedral Generation Directions

Unfortunately, Netgen may not be able to perfectly meet the needs of some more advanced users. Obtaining optimal mesh quality in 3D is significantly more difficult than in 2D, and Netgen is designed to use an advancing front method to try to improve the final mesh quality over what might be expected from a method based on pure Delaunay tetrahedralization combined with optional point insertion and mesh refinement. Unfortunately, with the advancing front method, point insertion is **not** optional for some users. This issue is a minor handicap for typical Finite Element Method users, for whom a slightly overrefined mesh might only mean a slight increase in computational cost. For these users, this cost is more than balanced out by the benefits of having smoother meshes elsewhere, but the team also had some initial interest from users of the Pebble-Tracking Transport (PTT) method, which requires precise placement of tetrahedral vertices; for the PTT method, arbitrary point insertion and mesh refinement are thus not allowable.

Because the new tetrahedral mesh generation design followed an object-oriented pattern, however, this mistake is not irreparable. The work to factor Tetgen-based and Netgen-based tetrahedralization capabilities into a common interface could be reused by any future generator backends. Because of the difficulties of 3D mesh generation—such as the fact that even in the all-tetrahedral case no standard algorithm exists for obtaining a unique, globally optimal solution for a fixed set of points—a wide variety of different software packages with this capability have been released over the years.

Although most of the dozens of alternatives that were briefly investigated turned out to be unsuitable for MOOSE integration, three packages with Berkeley Source Distribution (BSD) licensing (or BSD-equivalent licensing), compatible software architectures, and choices of algorithm might be suitable for PTT users. The integration of any of these would likely be much faster than the integration of Netgen because common code would be shared and common build system work would be easily repurposed by the new backend. Providing alternative algorithm options might also be useful to for non-PTT mesh generation because different generation algorithms can have different levels of resulting mesh quality depending on the domain

to be meshed.

Regardless of backend, user uptake of tetrahedral meshing technology would be greatly facilitated by additional work to make the complete 3D workflow as user-friendly as the current 2D workflow. To bound a 2D domain for triangulation, it is now easy in MOOSE to specify an analytically defined boundary via a parsed function in the input file describing a parameterized curve. A single-parameter function cannot describe a 2D manifold to bound a 3D domain in the same way, and an atlas in higher dimensions is hard to specify analytically, so other options need to be explored. Netgen has capabilities for surface triangulation from CSG, which may be useful in this regard; alternatively, surface definitions based on level sets of 3D parsed functions may be considered.

Additionally, the implementation of tetrahedral meshing technology can be enhanced by developing a series of mesh generators that either utilize or support tetrahedral generation capabilities. For example, a 3D transition layer mesh generator could be created to connect specific surfaces of two or more input 3D meshes. Simultaneously, a surface mesh generator could be designed to produce inputs for the tetrahedral mesh generator based on logical operations, such as unions/intersections/subtraction of input 3D meshes, offering an alternative to direct analytic surface definition. As mentioned before, these logical operations may be achieved using Netgen's intrinsic CSG features.

4.7 AUTOMATIC AREA FUNCTION FOR *XYDELAUNAYGENERATOR*

Previously, *XYDelaunayGenerator* provided two ways for the users to control the size of the generated triangular elements:

- *desired_area*: A single constant value can be provided as the uniform requirement for the maximum triangular element area. Elements with areas larger than the value given by "desired_area" are further refined until the criterion is met.
- *desired_area_func*: A parsed space-dependent area function can alternatively be provided to introduce location-dependent area requirements for the triangular elements.

The aforementioned approaches offer users limited capabilities in controlling the element size, especially the localized element size. Therefore, the element quality could be poor if there is large variation in EDGE element size in the input external and holes boundary meshes. To improve the mesh quality in such cases, an additional option has been developed and added to *XYDelaunayGenerator* for better element size control.

The new option features an automatically generated area function calculated based on inverse distance interpolation of the EDGE element size of the external and holes boundaries. There is also an option to include a background value for those regions away from those boundaries. This feature is enabled by "use_auto_area_func," and the related parameters include "auto_area_func_default_size," "auto_area_func_default_size_dist," "auto_area_function_num_points," and "auto_area_function_power". The development of this new feature involved implementation on both the libMesh and MOOSE sides. The demonstration of the new feature, in comparison with the existing element size control options, is illustrated in Figure 37.

4.8 DATA DRIVEN MESH GENERATION

Use cases of the Reactor Module were identified in which the output mesh object at each mesh generation stage was not strictly required to build the final output mesh. Namely, for homogenized neutronics calculations, the base heterogeneous mesh providing the design of the reactor was not needed to build the homogeneous mesh. Using the metadata only up until a certain point in the generation tree was found to be an alternative pathway for mesh generation. By not generating the physical mesh for the entire generation tree, significant time savings are also possible.

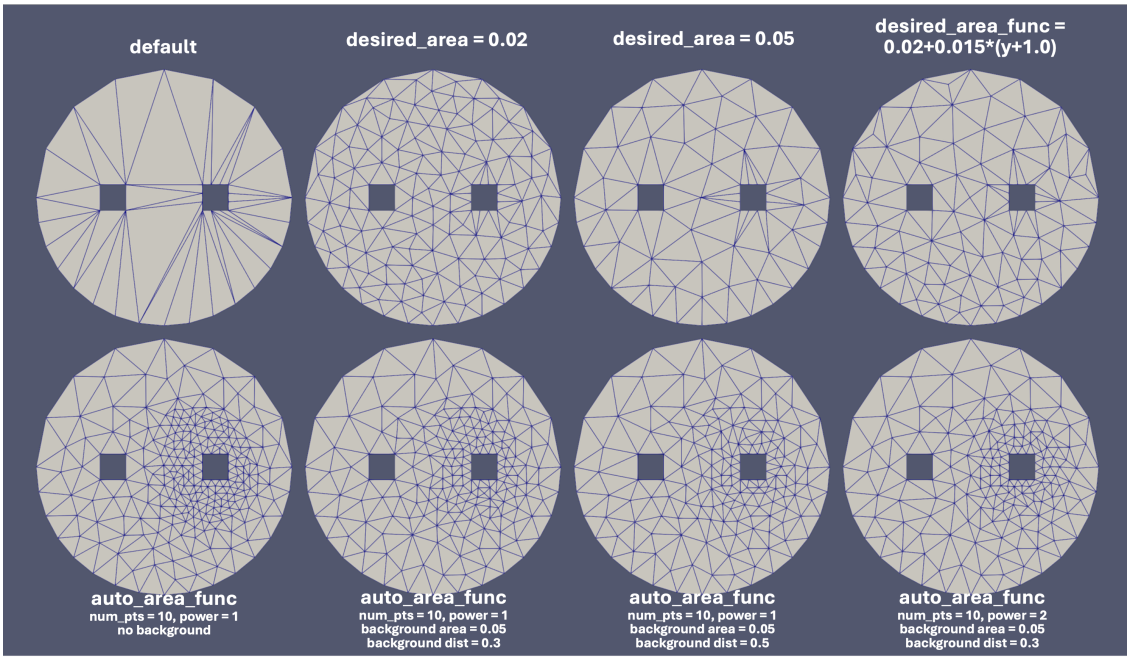


Figure 37. Examples of using different element area limiting options.

This use case led to the development of a data-driven mesh generation capability. A mesh generator that supports data-only generation can override the `generateData()` method and declare itself as a generator that supports this capability. The user can then specify a given mesh generator in the tree to be data-driven via the `Mesh/data_driven_generator` input parameter. All of the generators in the execution tree above said generator are executed in data-only mode (given that they support it), and no physical mesh is generated. All of the generators after the data-driven generator (including the data-driven generator itself) are executed as normal, not in data-only mode.

4.9 UPDATES TO REACTOR GEOMETRY MESH BUILDER (RGMB)

Based on stakeholder requests for expanded capabilities, a number of updates were made to the Reactor Geometry Mesh Builder (RGMB) mesh generation system. These updates are covered in the following subsections.

4.9.1 Setting Mesh Block Names Based on User-provided Region ID Mapping

By default, RGMB mesh generators—consisting of *PinMeshGenerator*, *AssemblyMeshGenerator*, *ControlDrumMeshGenerator* (added in FY 2024 and explained in Section 4.9.4), and *CoreMeshGenerator*—define two block names for the entire output mesh. One name is defined for the triangular elements in the mesh and another name is defined for the quadrilateral elements in the mesh. In Griffin, such a block name convention was not compatible with existing capabilities in the Griffin *Materials* system. Therefore, an additional parameter, *region_id_as_block_name*, was added to the *ReactorMeshParams* mesh generator, which isthe initial mesh generator that is defined before invoking RGMB mesh generators; the *ReactorMeshParams* mesh generator stores global information about the RGMB mesh generation workflow. By setting this parameter to true, the region ID mapping provided by the user at each mesh generation stage (pin, assembly, control drum if defined, and core) are used to set the block name of the output mesh. Specifically, the block name will be defined as *RGMB_<LEVEL_IDENTIFIER>_REG<REGION_ID>*, where *<LEVEL_IDENTIFIER>* is a unique identifier that is based on the level (pin, assembly, drum, or core) of the final mesh generator and the *type_id*

(unique identifying ID among structures that share the same level). Correspondingly, <*REGION_ID*> is the region ID associated with the mesh element. In addition, if the element under consideration is triangular, the suffix *_TRI* is appended to the block name to ensure that quadrilateral and triangular elements do not share the same block name. For example, if the final mesh generator is an assembly structure with a *type_id* of 1, the quadrilateral elements that have a region ID 5 will have block name *RGMB_ASSEMBLY1_REG5*, and triangular elements that share the same region ID will have block name *RGMB_ASSEMBLY1_REG5_TRI*. This naming convention ensures that every region ID defined on the output mesh will have one or two (in case of both triangular and quadrilateral elements) unique block names associated with it.

Figure 38 shows the default block naming convention as well as the block naming convention when *region_id_as_block_name* is set to true for a hexagonal pin with 2 ring regions, a single background region, and 2 duct regions. Notice how the innermost ring region is composed of quad and tri elements, so two block names (distinguished by different colors) are defined in the righthand image in order to avoid different element types from sharing the same block name.

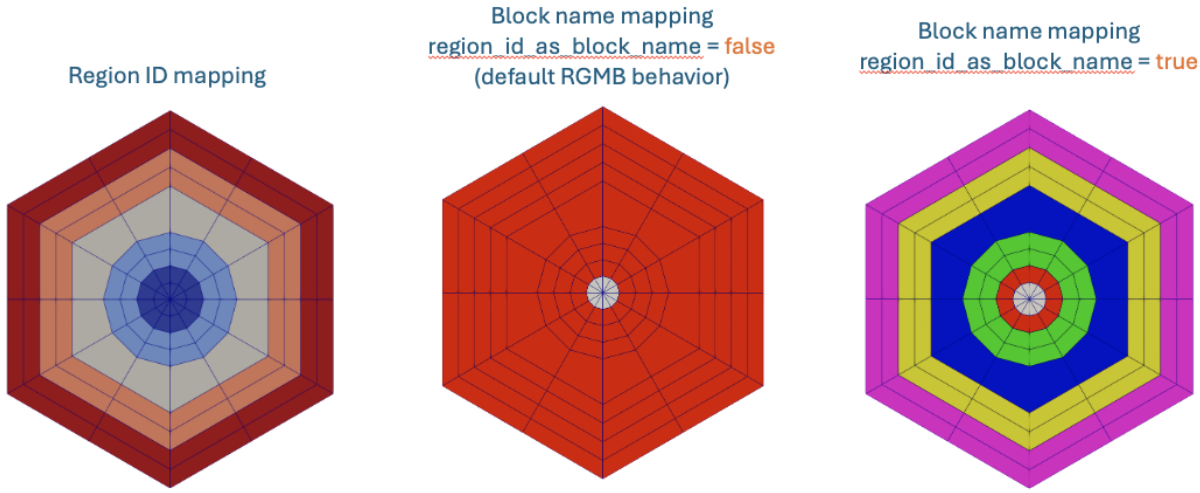


Figure 38. Default block naming conventions. Region ID mapping (left), default block name mapping (middle), and block name mapping with *region_id_as_block_name = True* (right) for a hexagonal pin structure with 2 ring regions, 1 background region, and 2 duct regions.

4.9.2 Enabling Data-Driven Generation in RGMB Mesh Generators and Griffin SFR workflows

In FY 2023, RGMB mesh generators were chosen as the candidate workflow for converting from heterogeneous mesh representations to equivalent homogeneous mesh representations for use by Griffin in downstream physics calculations (Shemon, Miao, Kumar, Mo, Jung, Oaks, Lee, et al. 2023). Because RGMB mesh specifications and region ID mappings are stored entirely as mesh metadata, this conversion process from heterogeneous to homogeneous mesh could occur entirely by inspecting just the mesh metadata defined on the heterogeneous mesh. Thus, an optimization that was pursued in FY 2024 was to bypass generation of an intermediate heterogeneous mesh and define the output heterogeneous mesh purely from the metadata defined on the heterogeneous mesh generator. Depending on the size of the heterogeneous core mesh and the mesh discretization that is employed within this mesh, the time savings from bypassing mesh generation of the heterogeneous mesh could also be quite significant.

Figure 39 shows the input file where a conversion from an input heterogeneous mesh defined by RGMB to an output homogeneous mesh created by Griffin's *EquivalentCoreMeshGenerator* takes place. The pa-

```
[Mesh]
  [rmp]
    type = ReactorMeshParams
    ... # Remaining parameters ommitted for brevity
  []
  [pin1]
    type = PinMeshGenerator
    reactor_params = rmp
    ... # Remaining parameters ommitted for brevity
  []
  [assembly1]
    type = AssemblyMeshGenerator
    inputs = pin1
    ... # Remaining parameters ommitted for brevity
  []
  [pin2]
    type = PinMeshGenerator
    reactor_params = rmp
    ... # Remaining parameters ommitted for brevity
  []
  [assembly2]
    type = AssemblyMeshGenerator
    inputs = pin2
    ... # Remaining parameters ommitted for brevity
  []
  [het_core]
    type = CoreMeshGenerator
    inputs = 'assembly1 assembly2'
    ... # Remaining parameters ommitted for brevity
  []
  [hom_core]
    type = EquivalentCoreMeshGenerator
    input = het_core
    target_geometry = full_hom # Convert het. mesh to homogeneous mesh
  []
  [translate_mesh]
    type = TransformGenerator
    input = hom_core
    transform = TRANSFORM
    vector_value = '1 0 0'
  []
  data_driven_generator = hom_core
[]
```

Figure 39. Input file that leverages data-driven generation for defining an output homogeneous mesh (*het_core*) from an input homogeneous mesh (*hom_core*). All mesh generators that come before *data_driven_generator* (in red) do not generate an output mesh and only define metadata on the mesh, whereas the *data_driven_generator* (in blue) reads in metadata from the input mesh to create an output mesh. All mesh generators that follow *data_driven_generator* (in green) behave as normal and will modify the input mesh and generate an output mesh.

rameter *Mesh/data_driven_generator*, explained in Section 4.8, points to the homogeneous mesh created by Griffin. This tells the MOOSE (G. Giudicelli et al. 2024) mesh generator system that all mesh generators that precede *hom_core* should not generate an output mesh and should only be responsible for defining metadata on the mesh. The data-driven generator (*hom_core*) is then expected to generate an output mesh by reading the metadata defined on the input mesh generator (*het_core*); all mesh generators that follow will behave as typical mesh generators by modifying the input mesh and generating an output mesh.

This new mesh generation workflow was tested for generating the assembly homogenized mesh for the full-core 3D Advanced Burner Test Reactor (ABTR) problem. The total mesh generation time on a single processor was 27.3 seconds when data-driven generation was not used and was 1.7 seconds when data-driven generation was used. These time savings are largely due to the fact that construction of the 3-D, full-core heterogeneous mesh is a costly meshing operation. Because metadata can instead be used to generate the target output homogeneous mesh, this heterogeneous mesh generation step can be entirely bypassed.

4.9.3 Support for Flexible Assembly Stitching in RGMB Mesh Generators

In the example discussed in Section 4.9.2, one aspect of the mesh generation workflow that led to some confusion among users was related to the generation of the heterogeneous mesh. For the ABTR example, specifically, hanging nodes were present at interfaces where heterogeneous and homogeneous assemblies were stitched together. This is because assembly structures created by RGMB's *AssemblyMeshGenerator* are stitched together without ensuring that the number and placement of nodes on both sides of the assembly interface agree with each other, as shown in Figure 40. For the purposes of converting the heterogeneous mesh to the homogeneous one, the presence of hanging nodes in the heterogeneous mesh does not affect the generation of the homogeneous mesh, which is generated entirely through mesh metadata and not from inspecting a physical mesh. However, the ability to generate and visualize a heterogeneous input mesh without hanging nodes was identified as a feature for streamlining RGMB mesh generation, especially for users who need to run physics calculations on a fully heterogeneous mesh.

Upon further testing of the RGMB mesh generation system, three scenarios were identified in which hanging nodes could occur during the assembly stitching phase into a core lattice:

1. Stitching of two assemblies that have a different number of pins in the pin lattice.
2. Stitching of two assemblies that have constituent pins that have different number of azimuthal sectors per side.
3. Stitching of a heterogeneous assembly (one with a defined pin lattice) with a homogeneous one.

The lefthand column of images shown in Figure 40 illustrates how these hanging nodes can occur for each of the three cases explained above for hexagonal assembly structures. To address the issue of hanging nodes, a new parameter in *ReactorMeshParams*, *flexible_assembly_stitching*, is defined to facilitate flexible stitching of assembly structures. When this parameter is set to true, the mesh subgenerator calls are modified to ensure that the assembly interface has a fixed number of sectors at the outermost assembly boundary interface for all constituent assemblies. The effect of setting this parameter to true is shown in the righthand column of images in Figure 40. For each of these images, each side of the assembly boundary interface has six uniformly spaced sectors, thus mitigating the issue of hanging nodes. The number of sectors defined at the outer assembly interface is controlled by the optional *ReactorMeshParams* parameter *num_sectors_at_flexible_boundary*.

To explain what is happening under the hood to facilitate flexible assembly stitching, the following steps are carried out through mesh subgenerator calls to generate the assembly mesh structure:

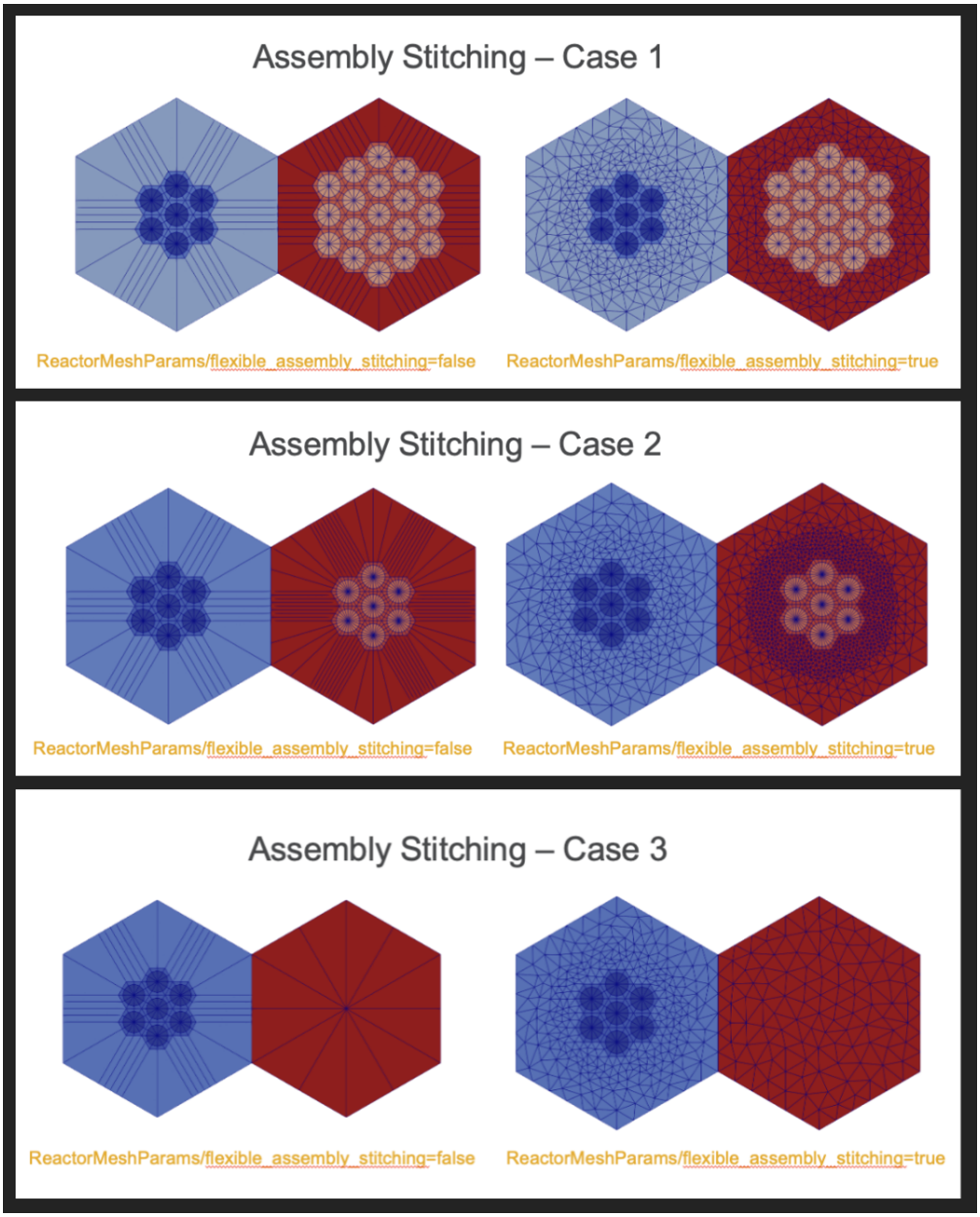


Figure 40. Assembly stitching. Three scenarios in which hanging nodes could occur when assemblies are stitched together with *CoreMeshGenerator* (left column of images), along with how flexible patterning fixes this issue by setting *ReactorMeshParams/flexible_assembly_stitching* to true (right column of images).

1. Generate assembly mesh structure using *PolygonConcentricCircleMeshGenerator* and *PatternedHexMeshGenerator* / *PatternedCartesianMeshGenerator* as normal.
2. Delete the outermost layer of the assembly mesh using *BlockDeletionGenerator*. For heterogeneous assemblies with duct regions, the outermost layer is the outermost duct layer. If duct regions are not present in the heterogeneous assembly, the outermost layer is the background layer. For homogeneous assemblies, the outermost layer is the entire assembly mesh.
3. Re-mesh the deleted outer layer using *FlexiblePatternGenerator*. This mesh generator, covered in the milestone report from FY 2023 (Shemon, Miao, Kumar, Mo, Jung, Oaks, Lee, et al. 2023), uses the auto area function (described in Section 4.7) to triangulate the layer that was deleted instead. The use of *FlexiblePatternGenerator* also ensures a fixed number of sectors at the assembly boundary.

The use of *flexible_assembly_stitching* preserves the behavior of extra element ID generation and block naming conventions for the mesh layer that was deleted and then triangulated, and this parameter can be used in conjunction with data-driven generation. This single parameter also significantly simplifies the mesh generation process for the user so that the user does not need to be concerned with the specific meshing operations required to prevent hanging nodes from occurring. Additionally, the RGMB system will throw a warning at the core-level if it suspects that flexible assembly stitching is needed based on how the assembly structures are defined and *flexible_assembly_stitching* is not set to true.

As an example of how flexible assembly stitching can be used on a full-scale mesh, the righthand image of Figure 41 shows a quarter-core slice of the heterogeneous ABTR mesh. The top right image in Figure 41 shows the mesh element discretization using a naive stitching algorithm in RGMB, and it can be clearly seen that this leads to hanging nodes at the interface between heterogeneous and homogeneous assemblies. However, this issue can resolved by the flexible assembly stitching option in RGMB, as shown in the bottom right image of Figure 41. This option ensures that the placement of nodes is identical on both sides of this assembly boundary interface.

4.9.4 Defining Control Drum Structures with RGMB Mesh Generators

With a general way to stitch assemblies together using the RGMB workflow, as described in Section 4.9.3, RGMB mesh generators can now also be extended to support the definition of control drum structures. The control drum is a series of two concentric rings surrounded by a background region that can optionally be discretized azimuthally into a drum pad region to signify the presence of a strong absorber region that can rotate around the centerpoint of the mesh structure. A diagram of the various regions within a hexagonal control drum is shown in Figure 42 for the case when a drum pad is and is not explicitly defined.

In general, stitching such a mesh structure into the assembly lattice of a reactor core can be a challenging task to ensure that hanging nodes do not occur. However, as explained in Section 4.9.3, a similar procedure can be performed in which the background region is triangulated to ensure that a fixed number of sectors is defined at the outer boundary of the control drum mesh. Thus, a new mesh generator, *ControlDrumMeshGenerator*, was defined to be used exclusively with RGMB mesh generators to facilitate stitching of control drum mesh structures into the assembly lattice of a reactor core. In terms of mesh subgenerator calls, *ControlDrumMeshGenerator* performs the following steps:

1. Define two concentric circles using *AdvancedConcentricCircleGenerator* with user-defined inner and outer radii and apply a uniform azimuthal discretization also based on user-defined input.
2. Triangulate the background region with *FlexiblePatternGenerator* using the auto area function and define a fixed number of sectors at the outer boundary of each side of the control drum mesh structure.

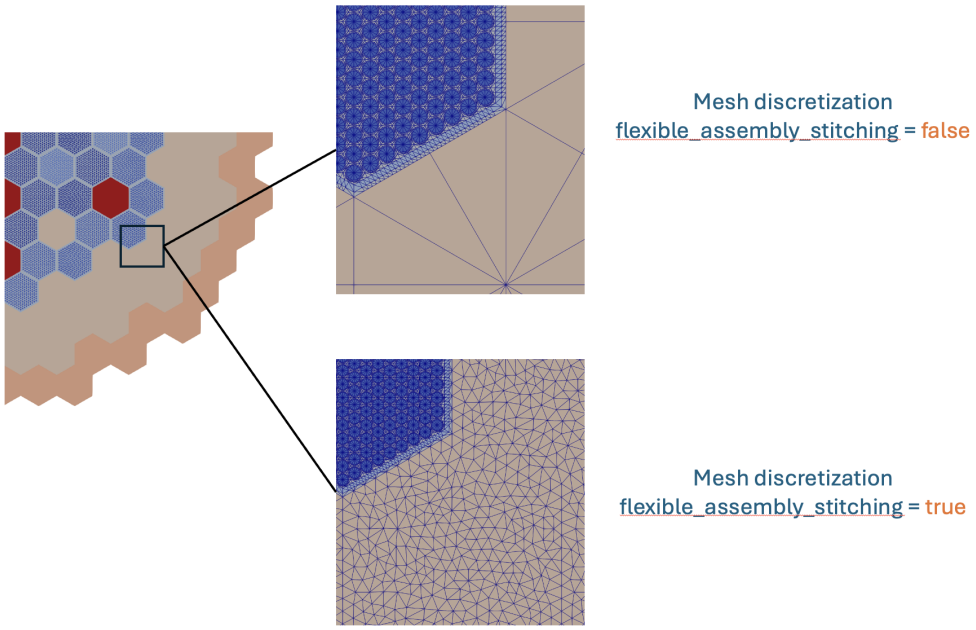


Figure 41. Heterogeneous ABTR mesh (left) with a zoom in of what the mesh elements look like at the interface of heterogeneous and homogeneous assemblies when flexible assembly stitching is not used (top right) and when flexible assembly stitching is used (bottom right).

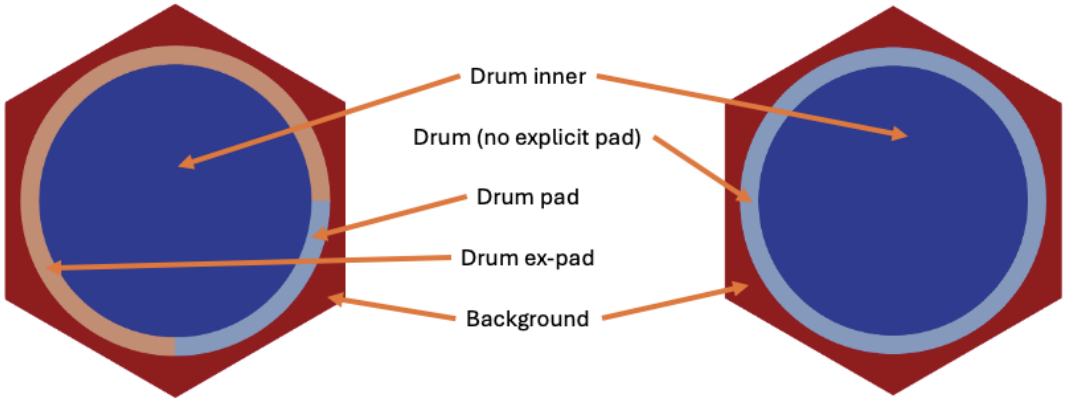


Figure 42. Various regions for a control drum mesh structure with a drum pad region explicitly defined (left) and without a drum pad region defined (right).

3. If a pad region is requested by the user, iterate through all mesh elements of the drum region and set the region ID / block name of elements that fall within the range of the start and end angle of the pad region (both angles are defined by the user). If the pad angles do not align with the azimuthal discretization of the drum region, define additional azimuthal nodes to ensure that the pad placement lines up exactly with the azimuthal discretization of the drum region.

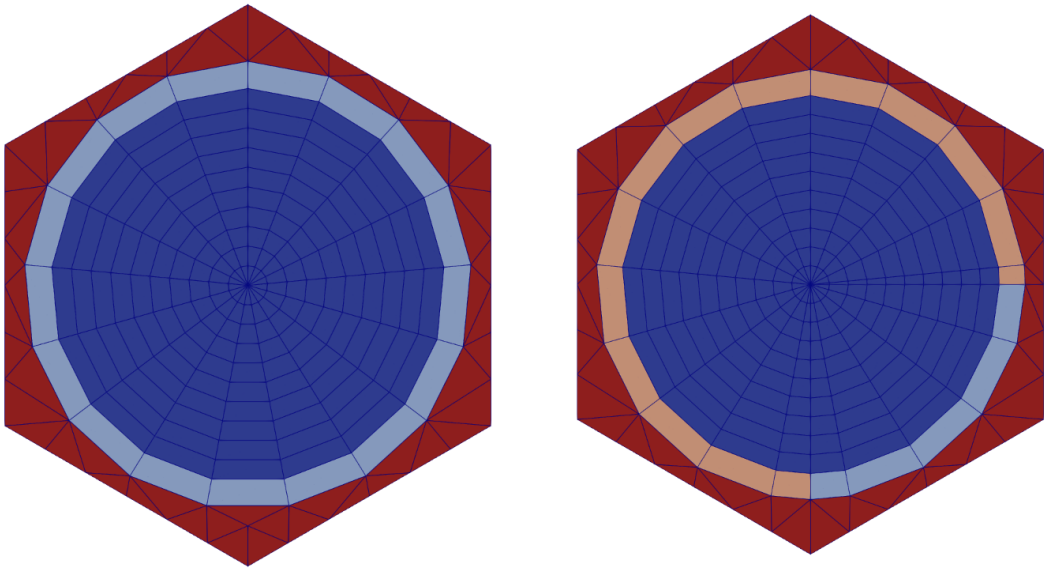


Figure 43. Mesh discretization for a control drum region where the azimuthal drum discretization does not line up with the start and end angles of the drum pad region. The left image shows what the control drum mesh looks like without a drum pad region defined, and the right image shows the same control drum mesh with drum pad start and end angles of 90 and 180 degrees, respectively.

To illustrate the details of the final step, Figure 43 shows the mesh discretization of a control drum mesh where the azimuthal discretization of the drum region does not line up with the placement of the drum pad region. In this example, the left image shows a scenario in which 17 azimuthal discretizations are requested in the drum region. In the right image, the drum pad start and end angles are 90 and 180 degrees, respectively (the convention for drum angles is to start in the positive y direction and rotate clockwise). Therefore, *ControlDrumMeshGenerator* is also responsible for automatically adding additional azimuthal nodes at the start and end angles of the drum region.

To use *ControlDrumMeshGenerator*, *ReactorMeshParams/flexible_assembly_stitching* needs to be set to true. Moreover, *ControlDrumMeshGenerator* preserves basic RGMB functionality found in other mesh generators such as extra element ID mappings, data-driven generation, and setting the block name from the region ID mapping. When the region IDs of the control drum mesh are being set, three values need to be specified per axial layer when a drum pad region is not defined. These values represent the three radial regions (drum inner, drum, and background) of the mesh. When a drum pad is explicitly defined, an additional region ID value that represents the region ID of the drum pad region needs to be provided per axial layer.

Additionally, the structure created by *ControlDrumMeshGenerator* can be used directly within *CoreMeshGenerator* as an input assembly structure. This allows for the stitching of heterogeneous assemblies with a pin lattice directly with control drum regions. As an example, Figure 44 shows the 2D Empire mesh defined entirely with RGMB mesh generators. The images on the left and right hand sides show the mesh

discretizations in regions where flexible assembly stitching is used to stitch dissimilar assemblies together.

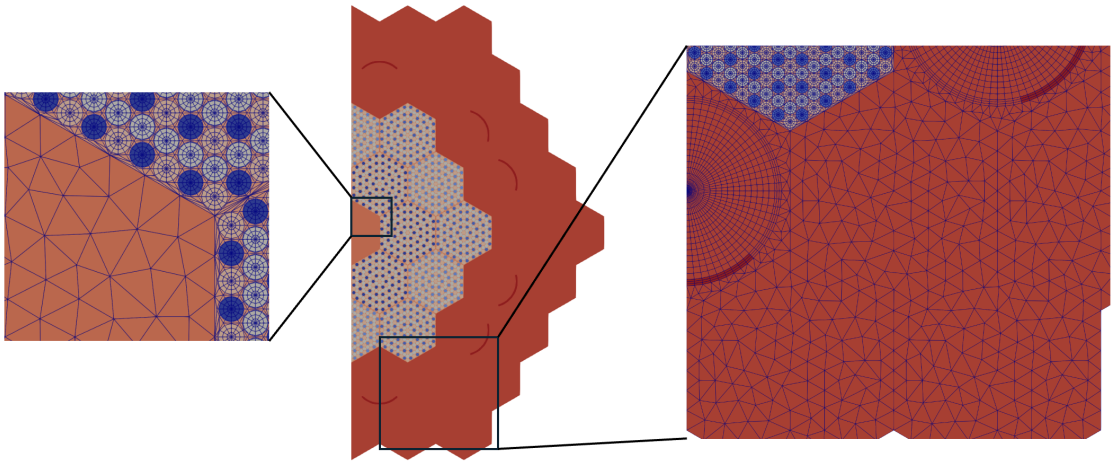


Figure 44. 2D Empire mesh generated from RGMB mesh generators (center), and zoomed in areas (left and right) showing mesh discretizations of regions where dissimilar assemblies are stitched together.

4.9.5 Adding Depletion ID Generation to RGMB

A depletion calculation in Griffin requires an integer ID tag, named depletion ID, to specify the depletion region setting in the core domain. Elements are grouped into separate depletion zones based on their depletion ID value. Unique isotopic inventories of depletable materials are defined in each depletion zone. This enables RGMB to generate core mesh files that can be used for fuel cycle calculation. Users can enable the depletion ID generation option by setting the input parameter *generate_depletion_id* to "true" in in the final level of mesh generation stage. The following mesh generators support depletion ID generation: *AssemblyMeshGenerator*, *ControlDrumMeshGenerator* and *CoreMeshGenerator*. For example, if a user generates an assembly mesh, the depletion ID option should be specified in *AssemblyMeshGenerator*. If a core mesh is generated, the depletion ID option should appear only in the *CoreMeshGenerator*.

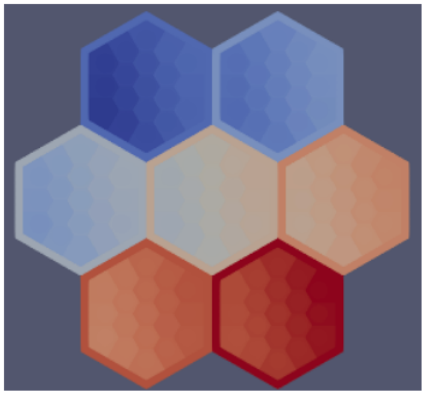
The input parameter *depletion_id_type* controls the level of details in the depletion ID assignment. Users can select "pin" or "pin_type" for *depletion_id_type*. All pins in the core will have separate depletion ID values if *depletion_id_type* is set to "pin", whereas using "pin_type" will result in unique ID values for each individual pin type in the assemblies.

4.10 MONTE CARLO CONSTRUCTIVE SOLID GEOMETRY SUPPORT

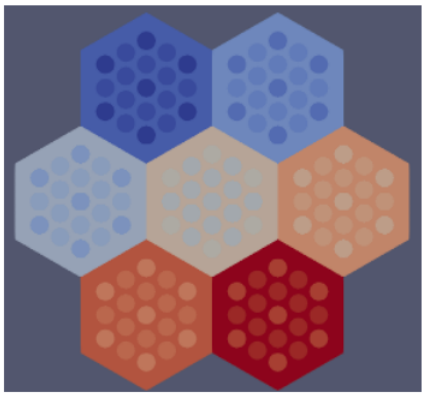
The goal of this work is to create a generalized and automated process for creating Monte Carlo (MC) constructive solid geometry (CSG) definitions from MOOSE geometry input to support the reactor analysis needs in the reactor physics community. Specifically, users of MOOSE-based nuclear reactor analysis software currently must manually develop the MC CSG models of the geometries to generate group cross sections (such as for deterministic neutronics analysis), create reference results, and do code-to-code comparisons. Manually creating these MC CSG models is a complex and tedious task that requires users to be familiar with the syntax and complexities of a specific MC code and to understand how CSG models are created. Because these models can be large and complex, this process is both time consuming and prone to human error. Implementing an automated process for converting MOOSE-based geometries to MC CSG geometries will improve the workflow efficiency and reduce user error in neutronics analysis.

The goal this year was to investigate how a generic MOOSE-to-MC process could be implemented into

```
[core]
  type = CoreMeshGenerator
  ...
  pattern = ' 1 1;
            1 0 1;
             1 1'
  generate_depletion_id = true
  depletion_id_type = 'pin' or 'pin_type'
[]
```



depletion_id_type = pin



depletion_id_type = pin_type

Figure 45. Depletion IDs generated by RGMB.

the MOOSE framework and to determine the effort required and timeline for implementation. At a high level, the process will be to generate a new CSG object that contains the equivalent CSG description of the MOOSE mesh generator's (MG's) input. This section outlines the details of the planned code implementation, how it will connect to specific MC codes, and the anticipated implementation timeline.

4.10.1 Code Implementation Plan

A new base CSG class is to be added at a high level to the MOOSE framework (located at `/moose/framework/[src/include]/csg/`). This base class will have subclasses that contain the basic common infrastructure required for generating CSG objects for all MGs, including the overarching generic `generateCSG()` method (akin to `generate()`, which already exists on all MGs) and any common utilities or common data types that are used across multiple MGs (creating simple surfaces, transformations, universes, and for data retrieval etc.). Each MG will then have its own unique implementation of the equivalent CSG operations or geometry definitions. If an MG does not have an implemented CSG generation method, the default behavior will be to throw an error in the event that a CSG geometry is requested by the user. These generation methods will be called in the same order that MGs are processed during the standard mesh generation process. It is important to note that mesh generation will not be required to generate the CSG object because the data required comes from the MG directly and not from the mesh. A visual depiction of this workflow is shown in Figure 46 where a CSG object (right) is created and modified at the same time and in the same order that the MGs are processed (left), using the information from the MGs to create the equivalent CSG definition (right flowing arrows).

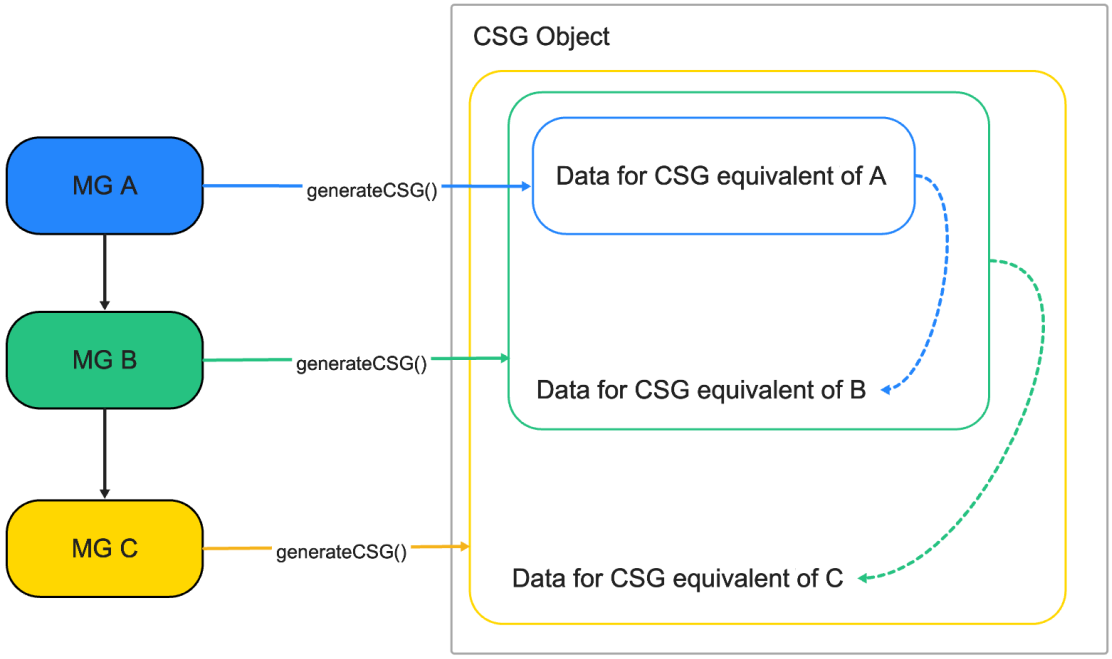


Figure 46. A visual representation of the process for generating a complete CSG object defined by the MGs A, B, and C (left). The `generateCSG()` process for each MG uses data defined on that particular MG to create the equivalent CSG representation on the right. The dotted lines on the right indicate where data may be necessary to flow between CSG representation to ensure consistency.

Implementing the CSG workflow such that it parallels the MG generation process will ensure the presence of data that may need to be transferred or referenced throughout the creation of the CSG object. This CSG workflow assumes a level of self-consistency between parent and child MGs that parallels the level of

consistency required for CSG definitions. Therefore, the assumption is that all information necessary for defining an equivalent CSG object of a particular MG is theoretically retrievable from that MG directly. In the case where some information is needed from a previous MG to build the current CSG definition, that data can be accessed from within the CSG object's data (dotted lines on the right side of the diagram in Figure 46). An important note regarding this workflow is that the MGs and any generated meshes are never modified in the CSG generation process; data is only retrieved from the MGs and used to modify the CSG object.

Figure 47 shows a specific example of this workflow for a basic model of a core that uses the RGMB `PinMeshGenerator`, `AssemblyMeshGenerator`, and `CoreMeshGenerator`. The CSG definitions of the pins are generated first, followed by the assembly, and then the full core. The data for defining a pin is stored in the pin universe of the CSG object, but the assembly CSG universe has knowledge of the pin universe and its unique ID, and can gather information about the pin if necessary. The core CSG universe operates similarly in how data is stored and accessed.

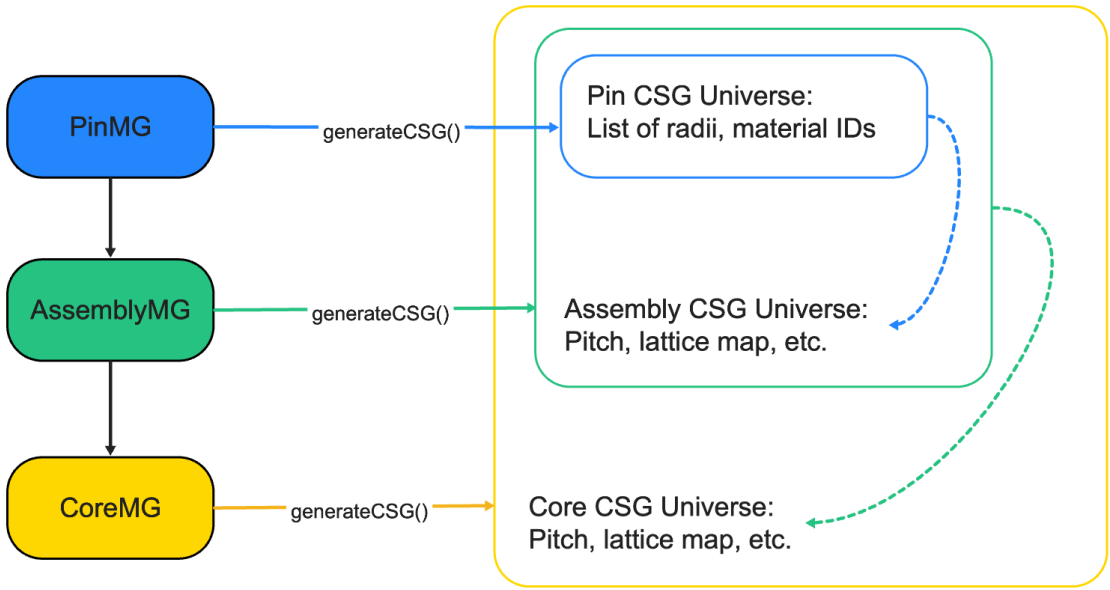


Figure 47. A specific example of the CSG generation workflow when used on a full core model using RGMB MGs.

4.10.2 Connection to Monte Carlo Codes

The goal of this implementation is to be able to create a generic, fully equivalent CSG model of the MOOSE mesh geometric definition (not the generated faceted mesh) that can then be connected to any MC code. The process for generating the exact code-specific syntax or input will need to be implemented as code-specific methods (eg, `toTitanCSG()` or `toOpenMCCSG()`) in the CSG class. This final conversion step to an MC code will occur after the full equivalent CSG object has been generated for the MOOSE geometry.

Of important note is the use of the terminology "equivalent CSG" throughout the description of this process. At the most basic level, CSG models are built by defining surfaces, applying Boolean operators to surfaces to create cells, applying various transformations, and repeating cells through uses of universes. However, most MC codes used for reactor physics support user-defined engineering units that remove the requirement to define rudimentary surfaces and cells directly, such as the common definition of a pin in OpenMC (Romano et al. 2015) or Shift's frontend Titan (Pandya et al. 2023). The equivalent of this level of engineering unit support for reactor physics is the MOOSE Reactor Module MGs, with the RGMB MGs

being near equivalents of these reactor engineering units. This means that an "equivalent CSG" definition of the `PinMeshGenerator` in RGMB would similarly be lists of radii and material IDs, which are what are typically used in MC codes to define a pin unit as well. However, if the MG used in the model is the `AdvancedConcentricCircleMeshGenerator` (which can also be used to define a pin-like structure), the equivalent CSG definition would be a series of cylinder surfaces joined via Boolean unions and/or intersections to form a series of cells with a specified material ID fill. Creating equivalent CSG definitions rather than strict rudimentary CSG definitions in all cases should make the direct connection to most MC codes more straightforward. In some cases, there may not be support for a particular engineering unit in an MC code, in which case it would be the requirement of that MC code's specific implementation to further break down the definition into usable units.

Additional requirements to fully define an MC model that can be used for a simulation are the material compositions and the physics/simulation settings. Defining materials is fairly standard across MC codes: a material is given a unique user-defined name or ID and then built by specifying the canonical `ZAID` for each nuclide and the nuclide amount or concentration (exact syntax may differ). In most MC codes, the details of a material definition are stored as data separate from the geometry definition. The geometry definition has information about the material ID that fills each region, which is then used to access material information during a simulation. This data separation is similar in MOOSE: materials are defined separately and region IDs are used to define the fill in a specific region of space. The same type of region or material ID connection would be done for the CSG definitions as well. Methods could be implemented in the CSG class to convert MOOSE-based material definitions to MC code-specific definitions that can be exported to users (or using block IDs as material placeholders if materials are not explicitly defined), further reducing the burden for manual generation of the MC input. Defining simulation parameters (number of histories, source parameters, tallies, etc.) from direct MOOSE input would not be as easy or possible because those parameters are specific to MC simulations only. This part of the MC input would be required to be manually created by users or implemented as code-specific methods in MOOSE. The latter method would require additional MOOSE input blocks. The former method is recommended because the definition of most simulation parameters for MC codes is generally not considered complex or time consuming for users.

4.10.3 Implementation Plan

The full implementation of CSG support in MOOSE will require multiple years and involve different groups of developers. Table 1 shows the anticipated timeline for implementation and prototyping tasks and indicates which NEAMS technical area will complete that task. Generally speaking, all MC code-agnostic tasks are to be completed by the Multiphysics technical area, whereas any MC code-specific tasks are to be completed by the Reactor Physics technical area or other MC code developers (as shown in Figure 48). The table indicates any task dependency (shown in third column) and the order of the tasks listed implies preliminary prioritization of which MGs will be supported first. Adhering to this timeline depends on adequate funding in each technical area to complete the designated tasks.

4.11 USER SUPPORT

The recent launch of the MOOSE Reactor Module sparked a surge in adoption of MOOSE meshing tools among NEAMS application users. Comprehensive support was provided to ensure that users could harness these tools effectively. The transition to MOOSE meshing tools has been swift and widespread across NEAMS Multiphysics Applications Drivers activities. Yet, some applications still push beyond the current limits of the Reactor Module, presenting opportunities for future enhancements. This section describes the team's efforts in developing meshes for various reactor types, presenting the versatility of MOOSE Reactor Module.

Microreactor Support The Kilowatt Reactor Using Stirling TechnologY (KRUSTY) is an experimental

Table 1. Implementation plan and targeted prototyping goals for CSG support

Task	NEAMS TA Ownership	Depends On
1. Implement the generic CSG framework in MOOSE that can be used by an MG	Multiphysics Applications	
2. Design the specific CSG methods for RGMB and Patterned Generator MGs in the Reactor Module	Multiphysics Applications	Task 1
3. Implement CSG support for remaining Reactor Module MGs	Multiphysics Applications	Task 1
4. Develop MC code-specific conversion methods for supported MGs	Reactor Physics	Completed implementation for at least a portion of MGs in Tasks 2 and 3
5. Implement generic material composition support in MOOSE	Multiphysics Applications	Task 1
6. Prototype Goal: generate a complete CSG model of a detailed 3D full core regular hexagonal lattice reactor	Multiphysics Applications	Tasks 2 and 3
7. Prototype Goal: generate a complete CSG model for a specific MC code of a detailed 3D full core regular hexagonal lattice reactor	Reactor Physics	Tasks 4-6
8. CSG and MC code-specific support for additional MOOSE MGs used to generate irregular reactor geometries	Multiphysics Applications	Tasks 2-4

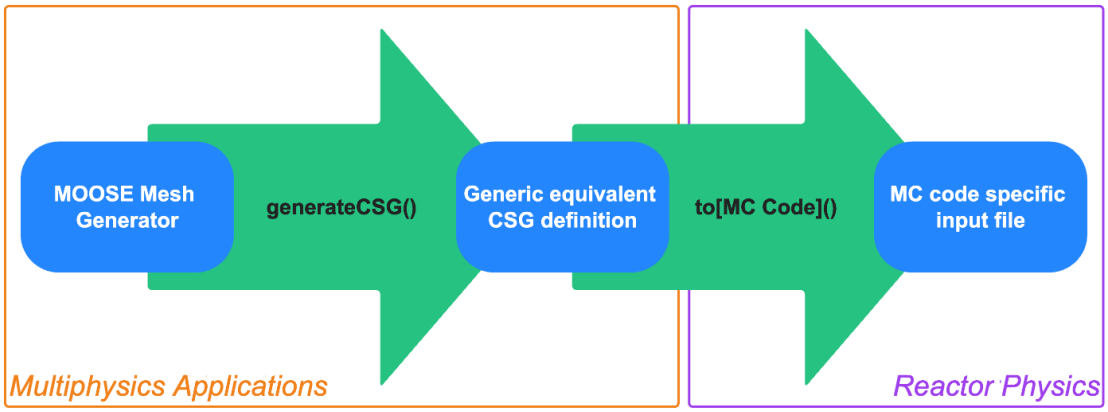


Figure 48. A high-level depiction of the MOOSE-to-CSG workflow and indication of which NEAMS technical area is in charge of the development tasks for that part of the workflow.

reactor developed by NASA and the Department of Energy at the Nevada National Security Site. KRUSTY is designed to demonstrate a full-scale Kilopower reactor operation for space applications (Gibson et al. 2018). KRUSTY utilizes the highly enriched uranium fuel system for low-power (1-10 kWe) space and surface power systems and the heat pipes to transfer thermal energy to Stirling engines (Poston et al. 2020). The initial KRUSTY mesh was developed in FY 2022 using MOOSE mesh generation tools. This early version was based on a simplified model from Los Alamos National Laboratory. As meshing technology progressed in FY 2023, it became possible to create more complex geometric meshes (Shemon, Miao, Kumar, Mo, Jung, Oaks, Lee, et al. 2023). This advancement led to the development of a more detailed KRUSTY model, which utilized recently implemented MOOSE Reactor Module objects, allowing for the creation of intricate mesh structures while preserving mesh volumes. In the current FY, the team developed a visual guide that demonstrates the process of KRUSTY mesh generation using the MOOSE Reactor Module (Kun Mo, Emily R. Shemon, Yinbin Miao, Yan Cao, Soon Kyu Lee, Aaron J. Oaks, Nicolas E. Stauff 2024). This instructional video showcases the steps involved in creating complex 3-D KRUSTY geometries and components. The visual aid has been employed to highlight the capabilities of the MOOSE Reactor Module at two significant events: the 2024 Modeling, Experiment and Validation School and the STARFIRE workshop.

Molten Salt Reactor Support The Molten Salt Reactor Experiment (MSRE), an 8MWth reactor utilizing molten fluoride salt with highly enriched U-235 in the fuel salt, was successfully operated in 1960s (Haubenreich and Engel 1970). This experiment demonstrated the practicality of the molten salt reactor concept for the first time. In FY 2022, the original full-core mesh was developed with MOOSE mesh generators, primarily relying on *FillBetweenSidesetsGenerator* and *AdvancedExtruderGenerator* objects. This original mesh was subsequently modified with the latest mesh generators to incorporate additional reactor components, preserve volumes for precise neutronics calculations, and reduce mesh density for improved computational cost. In FY 2023, the updated mesh was employed to conduct Multiphysics transient simulations coupling Griffin and SAM codes (Shemon, Miao, Kumar, Mo, Jung, Oaks, Lee, et al. 2023). At the request of the Molten Salt Reactor Application Drivers team, the MSRE dome mesh was developed by transforming a 3-D cylinder into a 3-D spherical cap, as shown in Figure 49. This mesh alteration was achieved using *ParsedNodeTransformGenetator*, which allows for precise control of curvature radius and height of the dome and cylinder according to user requirements.

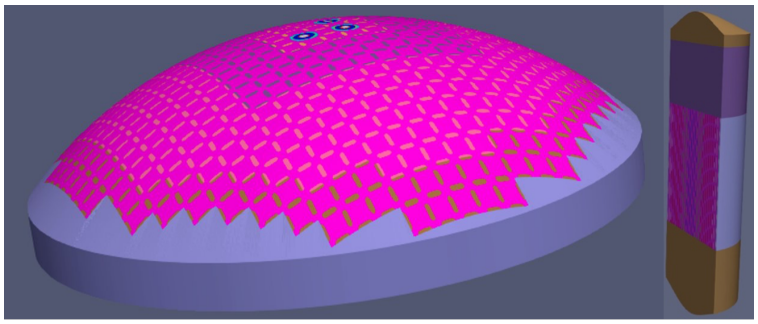


Figure 49. The curved upper head mesh and MSRE mesh with upper and lower heads.

Test Reactor Support The INL reactor physics analysis team requested assistance in utilizing the MOOSE Reactor Module to mesh the Transient Reactor Test Facility (TREAT) reactor geometries for neutronics analysis. Located at the National Reactor Testing Station, TREAT is an air-cooled, graphite-moderated reactor designed to safely conduct high-power integrated burst transients over a large sample volume. This facility operated for 35 years before being assigned in standby in 1994 and then restarted in 2017 to fulfill transient fuel testing research (Okrent et al. 1960; Pope et al. 2019). In response to the INL team's request,

the team demonstrated that the MOOSE Reactor Module is capable of generating the TREAT reactor geometries. The 2-D reactor mesh comprises standard fuel assemblies and control rod fuel assemblies, as illustrated in Figure 50. The fuel assembly was generated using *PolyLineMeshGenerator* and *XYDelaunayGenerator*, creating and meshing octagon-shaped fuel blocks and square shaped zircaloy cans in accordance with geometrical specifications listed in the baseline assessment report (Bess and DeHart 2015). For the control rod fuel assembly, the center circle and rings were generated with *ParsedCurveGenerator* and meshed with *XYDelaunayGenerator*. After two assembly types were generated, *PatternedCartesianMeshGenerator* was employed to arrange the assemblies, and *AddMetaDataGenerator* was used to add metadata to individual assemblies.

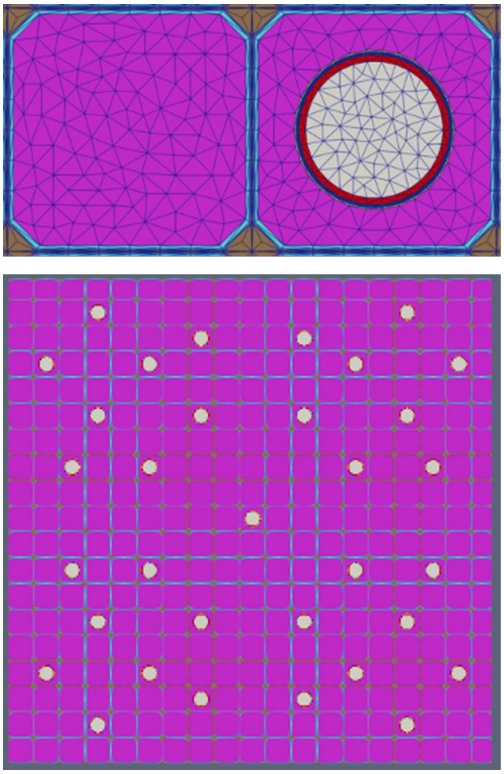


Figure 50. TREAT 2-D standard fuel assembly (top-left) and control rod fuel assembly (top-right) meshes and the reactor mesh (bottom).

Fast Reactor Support The Civil Nuclear Energy Research and Development Working Group (CNWG), established in 2012, is a collaborative effort between the United State and Japan. CNWG aims to facilitate joint civil nuclear research and development (R&D) work and to build upon collaborative R&D objectives outlined in the US-Japan Joint Nuclear Energy Action Plan (U.S. Department of Energy 2024a). As part of its sub-working groups, research activities in advanced reactor R&D include exploring innovative metal fuel core designs, materials, modeling and simulation, and advanced fuels that offer improved safety and efficiency (U.S. Department of Energy 2024b). At the request of the NEAMS Fast Reactor Applications team at Argonne National Laboratory (ANL), fast reactor meshes were developed using the MOOSE Reactor Module. The CNWG neutronics benchmark problem's fuel and sodium assembly meshes were developed using *PolygonConcentricCircleMeshGenerator* and combined using *PatternedHexMeshGenerator* to form a 3-D core. Notably, the benchmark problem required a selected assembly to be translated and tilted by user-specified amounts to examine the impact of new neutron leakage pathways. The team employed *ParsedNodeTransformGenerator* to select specified nodes on the target assembly and applied numerical

functions to translate and tilt them, as shown in Figure 51.

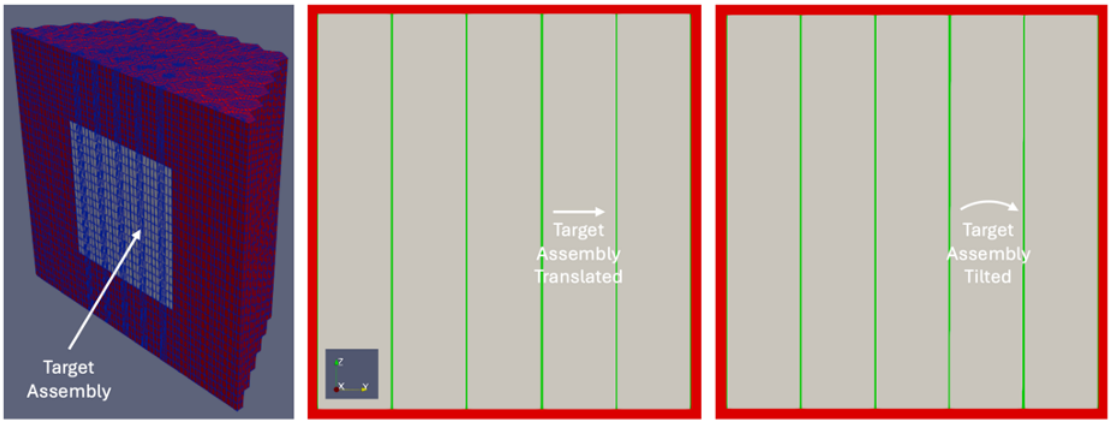


Figure 51. Fast reactor benchmark problem meshes with translated and tilted assemblies.

5. CONCLUSIONS

In FY 2024, significant improvements in MOOSE user interface integration were completed. Improvements to input processing enabled commonly requested features: file includes, block merger, parameter overrides, and autocompletion of partial input. Additionally, MOOSE language server updates increased user efficiency with increased documentation accessibility, more capable navigation, accelerated input editing via snippets and inclusion of required parameters, and an official VSCode 1.0 MOOSE input editor plugin that makes available to application developers the same user interaction advantages but within their familiar application code editor. These improve both MOOSE application developer and user interaction accuracy and efficiency and help shorten the feedback loop. Progress was made with the Cardinal application integration into the NEAMS Workbench with the addition of an available INL high-performance computing configuration that supports single-input interaction and job launch in the NEAMS Workbench.

This report also documents recent improvements to MOOSE's meshing capabilities. Improvements include (1) streamlining input for repeated meshing objects; (2) enabling support for quadratic elements (preserving the volume of circular surfaces like fuel pins while also reducing mesh density requirements for physics applications); (3) implementing 3D meshing capabilities such as revolving mesh construction (useful for PBR conical geometries or MSR tanks); (4) adding more features, such as control drum construction and the ability to stitch dissimilar assemblies, to the RGMB; (5) assessing the optimal path to adding Monte Carlo CSG support within MOOSE; and (6) continuing to support users with mesh generation and understanding their evolving needs. All of the new features added directly support the use of NEAMS tools for advanced reactor analysis. Next year, the Monte Carlo CSG generation task will kick off in earnest, and 3D mesh generation options will be further matured and integrated for specific reactor applications.

6. REFERENCES

Bess, John Darrell, and Mark David DeHart. 2015. *Baseline Assessment of TREAT for Modeling and Analysis Needs.* Technical report. Idaho National Lab.(INL), Idaho Falls, ID (United States).

Commission, US Nuclear Regulatory, et al. 2020. *NRC Non-Light Water Reactor (Non-LWR) Vision and Strategy, Volume 1–Computer Code Suite for Non-LWR Plant Systems Analysis.*

Gibson, Marc A, David I Poston, Patrick McClure, Thomas Godfroy, James Sanzi, and Maxwell H Briggs. 2018. "The Kilopower Reactor Using Stirling TechnologY (KRUSTY) nuclear ground test results and lessons learned." In *2018 International Energy Conversion Engineering Conference,* 4973.

Giudicelli, Guillaume, Alexander Lindsay, Logan Harbour, Casey Icenhour, Mengnan Li, Joshua E. Hansel, Peter German, et al. 2024. "3.0 - MOOSE: Enabling massively parallel multiphysics simulations." *SoftwareX* 26:101690. ISSN: 2352-7110. https://doi.org/https://doi.org/10.1016/j.softx.2024.101690. https://www.sciencedirect.com/science/article/pii/S235271102400061X.

Giudicelli, Guillaume L., Abdalla Abou-Jaoude, April J. Novak, Ahmed Abdelhameed, Paolo Balestra, Lise Charlot, Jun Fang, et al. 2023. "The Virtual Test Bed (VTB) Repository: A Library of Reference Reactor Models Using NEAMS Tools." *Nuclear Science and Engineering* 0 (0): 1–17. https://doi.org/10.1080/00295639.2022.2142440. https://doi.org/10.1080/00295639.2022.2142440.

Haubenreich, Paul N, and JR Engel. 1970. "Experience with the molten-salt reactor experiment." *Nuclear Applications and technology* 8 (2): 118–136.

Kun Mo, Emily R. Shemon, Yinbin Miao, Yan Cao, Soon Kyu Lee, Aaron J. Oaks, Nicolas E. Stauff. 2024. *KRUSTY Mesh Generation using the MOOSE Reactor Module.* https://www.youtube.com/watch?v=xvwhdcug14g.

Novak, A.J., D. Andrs, P. Shriwise, J. Fang, H. Yuan, D. Shaver, E. Merzari, P.K. Romano, and R.C. Martineau. 2022. "Coupled Monte Carlo and Thermal-Fluid Modeling of High Temperature Gas Reactors Using Cardinal." *Annals of Nuclear Energy* 177:109310. https://doi.org/10.1016/j.anucene.2022.109310. https://www.sciencedirect.com/science/article/pii/S0306454922003450.

Okrent, D, CE Dickerman, J Gasidlo, DM O'shea, and DF Schoeberle. 1960. *The reactor kinetics of the transient reactor test facility (TREAT).* Technical report. Argonne National Lab., Ill.

Pandya, Tara, Tarek Ghaddar, Rike Bostelmann, Matthew Jessee, and Philip Britt. 2023. *Modeling Enhancements and Demonstration of Shift Capabilities for PBR and MSR.* Technical report. Oak Ridge National Laboratory (ORNL), Oak Ridge, TN (United States).

Pope, Chad L, Colby B Jensen, Douglas M Gerstner, and James R Parry. 2019. "Transient Reactor Test (TREAT) facility design and experiment capability." *Nuclear Technology.*

Poston, David I, Marc A Gibson, Thomas Godfroy, and Patrick R McClure. 2020. "KRUSTY reactor design." *Nuclear Technology* 206 (sup1): S13–S30.

Romano, Paul K., Nicholas E. Horelik, Bryan R. Herman, Adam G. Nelson, Benoit Forget, and Kord Smith. 2015. "OpenMC: A state-of-the-art Monte Carlo code for research and development." *Annals of Nuclear Energy* 82:90–97. ISSN: 0306-4549. https://doi.org/https://doi.org/10.1016/j.anucene.2014.07.048. https://www.sciencedirect.com/science/article/pii/S030645491400379X.

Schwen, D and Giudicelli, G. 2024. *MOOSE Language Support,* March. https://marketplace.visualstudio.com/items?itemName=DanielSchwen.moose-language-support.

Shemon, Emily, Yinbin Miao, Shikhar Kumar, Kun Mo, Yeon Sang Jung, Aaron Oaks, Soon Lee, Cody J Permann, Guillaume Giudicelli, Logan Harbour, et al. 2023. *Meshing, Language Server Protocol, and other User-Oriented MOOSE Framework Improvements to Enhance Reactor Analysis Capabilities and Workflows.* Technical report. Argonne National Laboratory (ANL), Argonne, IL (United States).

Shemon, Emily, Yinbin Miao, Shikhar Kumar, Kun Mo, Yeon Sang Jung, Aaron Oaks, Scott Richards, Guillaume Giudicelli, Logan Harbour, and Roy Stogner. 2023. "Moose reactor module: An open-source capability for meshing nuclear reactor geometries." *Nuclear Science and Engineering* 197 (8): 1656–1680.

Stauff, Nicolas E, Yinbin Miao, Yan Cao, Kun Mo, Ahmed Amin E Abdelhameed, Lander Ibarra, Christopher Matthews, and Emily R Shemon. 2024. "High-Fidelity Multiphysics Modeling of a Heat Pipe Microreactor Using BlueCrab." *Nuclear Science and Engineering,* 1–17.

U.S. Department of Energy. 2024a. *Fact Sheet: United States-Japan Joint Nuclear Energy Action Plan.* https://www.energy.gov/articles/fact-sheet-united-states-japan-joint-nuclear-energy-action-plan.

———. 2024b. *Meeting Summary: Civil Nuclear Energy Research and Development Working Group.* https://www.energy.gov/ne/articles/meeting-summary-civil-nuclear-energy-research-and-development-working-group. Accessed: 2024-09-04.