In presenting this dissertation in partial fulfillment of the requirements for an advanced degree at Idaho State University, I agree that the Library shall make it freely available for inspection. I further state that permission to download and/or print my dissertation for scholarly purposes may be granted by the Dean of the Graduate School, Dean of the College of Science and Engineering, or by the University Librarian. It is understood that any copying or publication of this dissertation for financial gain shall not be allowed without my written permission.

_____ _____

Signature Date

Functional Expansions Methods: Optimizations, Characterizations, and Multiphysics Practices

Generalized Data Representation and Transfer Solutions in Multiphysics Simulations through the Characterization and Advancement of Functional Expansion Implementations

by

Brycen Linn Wendt

A dissertation

submitted in partial fulfillment

of the requirements for the degree of

Doctor of Philosophy in the Department of Nuclear Engineering

Idaho State University

Fall 2018

To the Graduate Faculty:

The members of the committee appointed to examine the dissertation of Brycen Linn Wendt find it satisfactory and recommend that it be accepted.

Leslie Kerby
Major Advisor

Mary Lou Dunzik-Gougar
Committee Member

Chad Pope
Committee Member

Jaakko Leppänen
Committee Member

David Beard
Graduate Faculty Representative

For Tanya

committed everything
spared nothing
surrenders never
loves forever

Acknowledgments

We are like dwarfs sitting on the shoulders of giants. We see more, and things that are more distant, than they did, not because our sight is superior or because we are taller than they, but because they raise us up, and by their great stature add to ours.

— John of Salisbury

Undoubtedly, I am who and where I am today because of all those whom I have been privileged to interact with throughout my life and education. My gratitude goes out to all those, whether named here explicitly or otherwise, who have touch my life for good.

To Dr. Leslie Kerby, who stepped in with unwavering support on the fateful day when a series of unfortunate events left me without a dissertation topic or research support. The confidence and trust instilled in me fostered daily growth. Guidance was always just an email or office visit away; kindness, calmness, compassion, and understanding defined every interaction.

To Dr. Jaakko Leppänen and Dr. Ville Valtavirta, whose patience and responsiveness prevailed always. I would not have been able to accomplish what I did in Serpent without their expertise and willingness to assist.

To Derek Gaston, Cody Permann, and the whole MOOSE development team, whose answers to my many questions paved the road to success. Their insights and guidance were always provided professionally and respectfully, both qualities I treasured deeply.

To April Novak, whose grace and humility painted every word, comment, and critique. Her assistance, wisdom, and inquiring mind helped to forge many code designs for the contributions to MOOSE.

To Dr. Mary Lou Dunzik-Gougar, who has had the patience to continuing working me since my early college days. Throughout my education, she has provided a wellspring of inspiration, kindness, trust. Fearless, passionate, and insightful; I was always courteously informed whenever an action or behavior needed reconsidering.

To Dr. Chad Pope, whose strength and dedication have driven me to seek for constant improvement. Despite an ever-hectic schedule, there was always time for one-on-one leadership, mentoring, and advice.

To Dr. David Beard, whose guiding hand has been present since my first foray into graduate studies. An everlasting passion for learning, philosophy and versatility, paired with an overwhelmingly caring persona, resulted in innumerable edifying and enjoyable interactions.

To Ken (Dr. Ken Bosworth), whose love of life and the outdoors was a frequent breath of fresh air. There was never a dull moment during any of his lectures; learning was truly and exciting and engaging pursuit with him around. Calling me "Paragon," he started me on an educational trajectory into computer science that has defined both my research and future career.

To Dr. Eric Burgett, who planted my tentative feet firmly on the path to a PhD. Under his supervision, I learned much about myself and was able to reach heights previously unreachable.

To Dr. Michael Lineberry, who provided years of advice and guidance. I believe he saw more in me than I saw in myself. I pray he is pleased with where I am today.

To Dr. Jay Kunze, whose eternal energy instilled an desire to learn something new every day. A bastion of experience and knowledge, to me he is a "renaissance man" of nuclear engineering.

To the staff of CoSE and the Graduate School, who were always kind and courteous no matter how odd my questions were. A strong force that moves silently inside the machine, they were there to just make everything work.

To my parents, who loved me unconditionally. Under their warm, watchful care I developed love of learning and a passion for science. They always wanted the best for me, and I hope I have accomplished a good work in their eyes.

To my son Lucas, who patiently understood why his dad was gone to school so much. I will forever treasure his love and undeniable cuteness. In his own way, he help by praying nightly for over a year that I would finish school and get a job.

To Tanya, who nearly gave up her life for this endeavor. She has been a constant fountain of love and support. A planner and scheduler by heart, she provided daily encouragement and motivation to progress toward completion. She selflessly managed the home so I could focus on my schooling, yet gave me freedom each night to spend quality time with Lucas before his bedtime.

Finally, to God who has granted me a life filled with faith, hope, and love. I write this today because of the many blessings I have received at His merciful hand.

This research was supported [in part] by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of two U.S. Department of Energy organizations (Office of Science and the National Nuclear Security Administration) responsible for the planning and preparation of a capable exascale ecosystem, including software, applications, hardware, advanced system engineering, and early testbed platforms, in support of the nation's exascale computing imperative.

Table of Contents

List of Figures

List of Tables

List of Files

Acronyms and Initialisms

ATF	accident tolerant fuel
ATR	Advanced Test Reactor
BE	best estimate
BEAVRS	benchmark for evaluation and validation of reactor simulations
BWR	boiling-water reactor
CAES	Center for Advanced Energy Studies
CSG	constructive solid geometry
CSV	comma-separated values
DOE	Department of Energy
FE	functional expansion
FET	functional expansion tally
FLOP	floating point operation
FOM	figure-of-merit
ICSBEP	International Criticality Safety Benchmark Evaluation Project
ID	identification
INL	Idaho National Laboratory
ISU	Idaho State University
JPEG	Joint Photographic Experts Group
LTO	link-time optimization
LWR	light-water reactor
MC	Monte Carlo (*see* stochastic)
MCNP	Monte Carlo N-Particle [1]
MFC	Materials and Fuels Complex
MIT	Massachusetts Institute of Technology
MPI	message passing interface
M&S	modeling and simulation
OMP	OpenMP

OOP	object-oriented programming
PNG	portable network graphics
R&D	research and development
RGB	red-green-blue
ROI	region of interest
TREAT	Transient Reactor Test Facility
US	United States of America
V&V	validation and verification

List of Symbols

Greek

Γ	mathematical domain
$\delta_{i,i'}$	Kronecker delta, defined in eq. (3.6)
Λ	functional expansion coefficient set, defined by eq. (3.1)
ρ	weighing function for an orthogonal functional ψ
Σ	macroscopic cross section
σ	1) standard deviation 2) microscopic cross section
σ^2	variance
Φ	Zernike polynomial azimuthal component
X	reactions
χ	reaction event
ψ	set of functional terms

Roman

a	standard coefficient set for a functional series set
b	orthonormalized coefficient set for a functional series set
$\mathfrak{C}$	total computational cost of an algorithm measured in FLOPS
c	orthonormalization constant, defined by eq. (3.13)
F	true distribution
j	linear Zernike polynomial index defined by eq. (3.48)
l	Legendre polynomial term order
ℓ^2	Euclidian norm
m	Zernike polynomial rank defined by eq. (3.29)
N	1) population 2) Zernike polynomial series order
n	Zernike polynomial order defined by eq. (3.28)
P	Legendre polynomial set
p	1) particle 2) absolute Zernike polynomial order defined by eq. (3.32)
R	1) Zernike polynomial radial component 2) reaction rate
S	sampled distribution
s	value of reaction χ
t	time
V	volume
w	stochastic weight
$\mathbf{x}$	event coordinates

Z 2D Zernike polynomial set, a combination of radial components R and azimuthal components Φ

Symbol Modifiers

$\overline{abc}$ mean value
$\widetilde{abc}$ orthonormalized
$\widehat{abc}$ statistical estimate
$\underset{\sim}{abc}$ expected value

Glossary

1D one dimensional
2D two dimensional
3D three dimensional

abstract in OOP, a class in which all the procedures are declared but at least one is virtual and lacking a definition (*comp*: interface)

bloat an increase in code size and features in an attempt to appease a broader user base

class in OOP, a collection of data and procedures that are closely related
code **1** *noun* **a:** a computer program **b:** the instruction set of a computer program in a human-readable form (*also:* source *or* source code) **c:** the compiled and machine-instruction binary counterpart of source code
2 *verb* to write the source code to a computer program
concrete in OOP, a class in which all procedures have a definition (*comp*: abstract)

declaration in code, the description of a procedure by its name, input(s), and output(s)
definition in code, the instruction set that comprises a procedure or function
deterministic a purely mathematical model of a system that depends on the state history of the variables (*comp*: stochastic)
domain a region characterized by its properties [2, p. 198]

exascale a system capable of 1×10^{18} FLOPS per second (*rel*: petascale)
explicit coupling a time-discretized coupling method in which the input data are the results from the previous time steps (*comp*: implicit coupling) [3]
extensive a property of a system with physical quantities that are additive and based on the amount of material in the system
extrema the largest and smallest values of a function within a domain, determined either locally or globally also

field a set of physics, or physical behavior, modeled by a system of equations
finite element a single element in the discretized mesh of a finite element analysis solution
finite element analysis a numerical method for solving a set of equations using a discretized mesh of a continuous domain
first principle a basic and established scientific proposition that is not based on assumptions, empirical models, or fitted parameters

full coupling a time-independent coupling method in which distinct fields, operating within a shared domain and length scale, are solved simultaneously (*comp*: loose coupling, tight coupling)

functional expansion a representation of data using a series of functions, similar to a representation of periodic data by a Fourier series (*acronym*: FE)

functional expansion tally an application of functional expansions to represent a distribution of discreet events generated by a simulation (*acronym*: FET)

hash value a value that is a smaller and (ideally) unique converted/collapsed representation of a much larger data type/set

implicit coupling a time-discretized coupling method in which all the governing equations for each field, including time derivatives, are assembled together and solved simultaneously (*comp*: explicit coupling) [3]

inheritance in OOP, the process of deriving a new class from an existing class to reuse the functionality while adding additional features

intensive a property of a system with values that are independent of the amount of material in the system

interface in OOP, a description of the actions that an inheriting class must implement via virtual procedures (*see*: abstract, concrete)

loose coupling a time-independent coupling method in which distinct fields, operating within a shared domain and length scale, are solved sequentially in a one-pass approach (*comp*: full coupling, tight coupling) [4]

macroscale a length scale with features visible to the naked eye

map in C++ (and many other OOP languages), a data storage class that references a (usually larger) data set by a (suaully simple) key value

maxima the largest values of a function within a domain, determined either locally or globally also

memoize to store the result of a (usually complex) calculation for easy recall later, effectively attempting to prevent repeated calculations by incurring a slight memory storage and access cost instead

mesh tally a tally that represents a sampled distribution by subdividing the ROI into discrete bins, then recording events on a bin-by-bin level (*also*: histogram, histogram tally)

minima the smallest value of a function within a domain, determined either locally or globally also

moment a scalar quantity used to characterize the similarity in features of an object w.r.t. a basis function; mathematically this is the projection of that object onto the corresponding function [5, p. 26]

multidomain a system characterized by interactions between different domains, i.e., regions with drastically different properties [2, p. 198]

multifield a system characterized by simultaneous excitation and response to multiple physical fields, such as electric, magnetic, heat, and pressure [2, p. 198]

multiphysics a solution space consisting of multiple domains, fields, and/or scales [2, p. 198–199]

naive an intuitive, fundamental, or common approach
nanoscale a small length scale measured in nanometers
norm a mathematical quality that describes the extent of an object, herein the square root of the inner product $\sqrt{\langle f(x), g(x) \rangle}$ over an defined interval $[a, b]$ [6][7, pp. 500]

object a runtime instance of a concrete class that has a location in an application's memory
open source a codebase that is free, within the limitations of a designated license, for all to view, use, modify, and submit improvements
orthogonal a set of polynomials whose inner product is equal to zero [7, p. 500]
orthonormal a specialization of orthogonal functions ψ such that the norms $\left\| \widetilde{\psi}_i \right\|^2$ are equal to one [7, p. 500]
override in OOP, the replacement of a parent class's existing method definition by a new definition within a deriving class

petascale a system capable of 1×10^{15} FLOPS per second (*rel*: exascale)
Picard iteration an iteration at a single time step in an explicitly coupled context, typically used to drive a multiphysics result toward the ideal implicit solution (*also*: fixed-point iteration; *rel*: tight coupling) [3]
Pu$^{\mathbf{x}}$ a mixture of unspecified Plutonium isotopes
policy-based in OOP, a polymorphic class design in which the templated parameter provides the realizatation of one or more actions required by the templating class
polymorphism in OOP, the ability to extend an interface to multiple data types
pulse a nuclear reactor operation characterized by a brief superprompt supercritical reaction, usually terminated via intrinsic feedback mechanisms

realm a distinct attribute space spanning a single domain, field, and scale
recurrence relation a property of a series that enables the calculation of a term using the values of previously evaluated terms

scale the length or time characterization of a system defined by the smallest possible measurable feature
standardized a function set $\psi(x)$ scaled such that $\left|\psi(x)\right| \leq Y$ over the interval $[\mathbf{x}_1, \mathbf{x}_2]$; typically $Y = 1$ and $\mathbf{x}_2 = -\mathbf{x}_1 = 1$
static a type of analysis characterized by a lack of a time domain
steady-state a type of analysis in which time is allowed to advance while holding operations parameters fixed, ideally allowing the system to converge on a single operational mode
stochastic a mathematical model of a system that utilizes randomness to mimic natural processes (*comp*: deterministic; *rel*: MC)

tight coupling a time-independent coupling method in which distinct fields, operating within a shared domain and length scale, are iteratively solved sequentially until convergence criteria are met (*comp*: full coupling, loose coupling; *rel*: Picard iteration) [4]
transient a type of analysis characterized by both changes in operational parameters and advancement of time

uniphysics a solution space consisting of a single realm, typically for generating a field solution (*comp*: multiphysics)

UO_2 Uranium dioxide

validation a demonstration that a model or simulation is solved accurately, or that the physics are implemented correctly [8]

vector in C++ (and many other OOP languages), a array-type data storage class with a dynamic size that is internally managed

verification a demonstration that the chosen model or simulation represents the desired system, usually established via comparison to test data from a real system [8]

virtual in OOP, a declared procedure that is lacking a definition and/or has a definition that can be overridden via inheritance

Zernike polynomials a specialized class of 2D orthogonal polynomials on the unit disk [9]

Functional Expansions Methods: Optimizations, Characterizations, and Multiphysics Practices

Dissertation Abstract—Idaho State University (2018)

Modern modeling and simulation efforts are becoming larger and more complex with corresponding evolutionary advances in computation hardware. High-fidelity physics data representation and communication are critical issues to any multiphysics simulation, no matter the scale. Functional expansions have been previously shown to have characteristics desirable for these situations.

In this dissertation, the understanding of functional expansion tallies was advanced through characterization in the Serpent reactor physics Monte Carlo code. New algorithms were developed that significantly improved the computational efficiency of functional expansion-based data representation methodologies. Next, a figure-of-merit was developed for functional expansion tallies, which was then used to demonstrate their advantage in overall computational time.

Functional expansion tools were then developed as a module in MOOSE, a finite element analysis framework. Developments included classes to generate, store, and reconstruct variable fields. These functional expansion tools were then adapted in MOOSE to couple with Serpent's multiphysics interface. All these developments were accompanied by testing to demonstrate both the viability and use of these tools.

This work is unique in the sense that it: 1) characterized—and improved upon—the computational efficiency of functional expansion algorithms, 2) developed a generalized functional expansion-based coupling framework in MOOSE, 3) coupled MOOSE and Serpent using functional expansions, and 4) performed all these functions in a fully-3D fully-multivariate context.

Key Words: functional expansions (FEs), functional expansion tallies (FETs), algorithm optimization, statistical convergence, multiphysics coupling, MOOSE, Serpent

Part I

Introduction

Chapter 1

Background

> *A painter should begin every canvas with a wash of black, because all things in nature are dark except where exposed by the light.*
>
> — Leonardo da Vinci

For millennia humankind has sought for answers to explain the mysteries that surround us. Conceptualization of Truth is scattered throughout our history: mythological deities as sources of natural events; Earth as the center of the Universe; air, water, earth, and fire as the basic components of matter. Today we are still on that quest, yet with each new answer we expose an entirely new set of questions.

A basic understanding of gravity is rooted in experiences we can all draw upon—gravity is an accepted phenomenon, observable both on earth and in the heavens. Much like Sir Isaac Newton and his anecdotal apple [10, pp. 15–16], objects falling toward Earth's center are daily experiences, routine and expected. But why does an apple, freed from its mother tree, descend to earth and not 'fall' toward the sky or another arbitrary direction? Classical mechanics have been used for centuries to define principles of interactions between different bodies. Albert Einstein, a few centuries later, reformed humankind's understanding of the Universe with his general theory on relativity.

Similar statements can be made in each field, from biological sciences to computer science, and from quantum physics to astrophysics. Everyday, respected and knowledgeable

scientists are questing for more meaningful answers. For example, nanoscale research has garnered huge interest in the past few decades. By delving into smaller scales and characterizing fundamental interactions, researchers hope to be able to explain and engineer macroscale behaviors. Various methodologies have been employed to parametrize and define these systems of behaviors.

1.1 Modeling

The Universe is the best representation of itself. Such tautology is self evident from these astronomical distances down to the size of a simple protozoa and beyond. The only guaranteed method to completely characterize and decompose a system—whether real or conceptual—is to actually observe and perturb it while in action. This is widely applicable, ranging from simple [11] to complex [12], incorporating elements biological [13] or artificial [14], and spanning the infinitesimal [15] to astronomical [16].

Maxim *Divining the behavior of a system within a desired state often requires assembly and configuration of the system itself. Adopting this approach for all situations would require limitless availability of three critical resources: time, material, and labor. Since these resources are not obtainable* ad infinitum, *predictive approaches must be used.*

Prototypes have been used extensively for this purpose. A significant amount of information can be gained by fabricating a smaller replica of a system and then testing the behaviors. The consumption of these three key resources—time, material, and labor—is drastically reduced, and the results of the testing can be used to design the optimal full-scale system [17].

Modeling and simulation (M&S) is another alternative for predicting a system's behavior. This approach ideally requires fewer resources while producing quasi-representative evaluations. If constructing a system is the end-goal, then M&S can be used to establish an expansive characterization and optimal design prior to full resourcal commitment. This can also be used to evaluate the safety/risk balance of using an existing system in a new or modified operational regime.

1.2 Digital Revolution

The advent of the Digital Age has revolutionized science-based modeling for system prediction and analysis. A system can be assembled and tested in a *virtual* reality. First principles provide the foundational governing equations. Data from integral and separate effects experiments are incorporated where first principles are impractical or not available.

Virtual M&S methodologies are now well-established, and continued research and development (R&D) efforts continue to expand and improve the current capabilities. Key areas of focus are: 1) support hardware, 2) parallelization, 3) validation, 4) optimization, and 5) fidelity.

Computer science continues to innovate in stride with hardware developments. Parallelized software is largely equipped to leverage the raw computational power of massive petaflop supercomputing centers appearing world-wide [18]. Additional performance optimizations are effected through parametrization studies, improved coding, and enhanced models. Research is now underway with a focus on developing the next generation of exascale-level hardware and software capabilities [19].

The ultimate prize, however, is the fidelity to reality: is the behavior of the digital system equivalent to the physical system? Experience indicates that, when reducing analyses with aggressive simplifications, unforeseen localized behaviors can remain undiscovered until actually observed in reality [20]. A significant requirement for production-level code—validation and verification (V&V)—ensures the virtual results can be trusted by certifying both functionality in design (validation) and capability in simulation (verification). Multiphysics approaches, enabled by advancements in computing capabilities over the last decade, are answering the call for high-fidelity M&S [21].

1.3 Multiphysics

Multiphysics M&S developments provide the foundation of next generation of high-fidelity tools. A multiphysics simulation, as described in *Modeling and Simulation of Multiphysics Systems* [2], is a solution space consisting of multiple domains, fields, and/or scales. The

converse approach, herein termed uniphysics, often involves assumptions regarding the behaviors of all but a few tightly related aspects of a system.

Primarily uniphysics M&S capabilities were developed historically due to limited computing power. These uniphysics methods are designed with a specific purpose, frequently intended for a static or steady-state process analysis. However, the uniphysics breadth of applicability is severely limited by the underlying specialization and assumptions; certain parameters may be deemed inconsequential because of the scale or scope of the simulation, even without computing limitations.

A multiphysics approach is designed to study the complex relationships between different domains, fields, and scales. This intrinsically realizes a larger scope of system behaviors, enabling high-fidelity transient analyses while also improving on the quality and applicability of static and steady-state simulations. Transient analyses in multiphysics simulations importantly depend on the coupling mechanism between time steps. Two approaches are commonly used to produce a multiphysics simulation: explicit coupling and implicit coupling.

Implicit coupling is a time-discretized coupling method in which all the governing equations for each field, including time derivatives, are assembled together and solved simultaneously. This is frequently an integral approach, in which all the desired fields, scales, and domains management schemes are organized within a single application. These codes are often purpose-built for a single problem. As such, they can be optimized with a specific purpose in mind, and are often the ideal M&S tools within their verified scope. However, the underlying codebase may be very large, and extending or expanding it can become arduous and result in significant bloat. Further, these codes are often commercially-developed products, and come with a large price tag.

From a holistic M&S standpoint, an integral or impicitly coupled approach is often ideal but not always possible—especially in the case where separate and independently-developed uniphysics codes are coupled together to create a multiphysics solution. This other case is an explicitly coupled approach, in which each set of equations are solved independently using the results from the previous time step's solution. Such approaches can even leverage

smaller independently-created open source projects, each created to solve a particular set of uniphysics or multiphysics, to assemble a broadly applicable multiphysics solution.

In the explicitly coupled approach, each code may have separate or distinct domains, fields, and scales. An additional application layer, the driver, is therefore required to form the communication platform between the codes and set the order of execution [22–25]. Additional attention is required to determine at which points information is passed between the individual codes, especially when the simulation involves multiple time scales. Picard iterations, for example, are a simple scheme that can be used to converge the combined state of several separate solutions to nearly the same result as would have been produced with an implicit approach [3]. Occasionally one of the independent codes is already capable, or can be augmented accordingly, to function as the driver [26, 27].

Chapter 2

Literature Review

Very rarely is something done that has not been done before.[1]

— Author

Multiphysics simulations are use in many disciplines to model everything from molecular bonding interactions to the merging of galaxies. A quick search for multiphysics applications yields literature in many research disciplines such as: astrophysics [25], cancer research [13], combustion dynamics [28], electronics [29], energy systems [15], fluids engineering [30], materials science [31], plasma physics [32], plate tectonics [33], and weather prediction [34].

2.1 Nuclear Codes

Narrowing the focus, computational tools enable the work of nuclear researchers and professionals alike. These tools, frequently referred to as codes, exist for every conceivable need: energy balance economics [35], fuel burnup [36, 37], criticality calculations [38–40], data acquisition [41, 42], isotope depletion [36], materials and fuels performance [26, 43], neutronics and particle tracking [40, 44], nuclear fuel cycle [45], probabilistic risk assessment [46], radionuclide identification [41], release pathways [47], spectra analysis [48], and thermal hydraulics [49, 50].

[1] Coincidentally, these exact words were probably uttered by other more-notable historical figures.

Nuclear technology research and industry activities are focused on one key factor: ensuring the public safety. Utilities, vendors, and other associated industry entities must balance the insurance of public safety in line with their business economics. Conservative models, developed at the inception of nuclear technologies, help ensure this safety. However, these analyses can inhibit the implementation of (otherwise) safe practices for flexible and efficient operations. More recently, these conservatisms are being replaced by best estimate (be) methodologies [21, 51]. Even non-revenue seeking entities, such as the United States of America (us) Navy, have adopted be practices in their safety analyses due to the enhanced benefits.

The technological innovations in advanced nuclear technologies have continued to stretch or surpass the limits of validity for the current fleet of nuclear codes and supporting benchmark data. One example is the safety analyses required for licensing a molten-salt reactor, another is the safety and performance analyses of advanced accident tolerant materials. To keep pace with these technology developments, multiphysics tools must be able to correctly model behaviors on shorter time scales, under more extreme environmental conditions, and with smaller uncertainties. Transient simulations are needed to capture events lasting only nanoseconds, then advance the same models for weeks or months and predict the long-term material responses to variable conditions [52].

Lastly, high availability of massive supercomputing systems [18] have enabled code developers to integrate more physics at smaller scales. These have manifested in many ways, such as the multidomain-multifield coupling between reactor neutronics and thermal hydraulics [53–60]. Other efforts have expanded on this by incorporating fuel performance/burnup models [61–63]. These included the addition of model parameters such as Pu^x rim concentrations [61, 64] or Doppler broadening [65] effects. Many of these aspects are simultaneously modeled in an expansive, high-fidelity fuels and materials performance modeling code called BISON [66]. Another group has further developed a comprehensive modeling package for existing light-water reactor (lwr) technologies [67].

2.2 Multiphysics Coupling

Domain coupling of field data in multiphysics applications typically come in two flavors: volumetric and surface. In volumetric couplings, the different fields operate over the same domain and volumetric state data are mapped between the separate models. An example of a simulation based on volumetric data is a temperature-dependent feedback study of a fuel element using a fission event generator (which provides a heat generation term) coupled to a heat conduction code. Surface-based scenarios intersect the field interactions only at domain boundaries, providing information about both intensive property states or flow of extensive quantities between domains. An example of a simulation based on surface data is a temperature study of a fuel pin by a fuel performance code (which both produces and transports heat) coupled to a thermal-hydraulics code (which transports heat away from the outside surface).

The simplest implementation of volumetric coupling utilizes identical domains and model structures for each field in all domains [68, 69]. Another approach join models in which the model structures are not aligned. This requires an interpolation or translation mechanism to properly map data between the disparate models [56, 57, 70, 71]. In some cases modifications are required to normalize or refine the data to ensure conservation of extensive properties [61, 72]. As another layer of complications, different codes use widely varying geometrical constructions. Most systems are designed based on the underlying solver method; for example, many Monte Carlo (MC) codes use constructive solid geometry (CSG) while finite element codes use mesh-based models. Even mesh-based models with identical domain dimensions are not guaranteed to be equivalent, as a number of different element shapes are available; `libMesh`, a library used in many finite element analysis codes, contains no less than 14 basic element shapes [73].

A simple implementation of surface coupling computes the average (zeroth-order) representation of the field at the domain boundary [74]. This results in a single easy-to-transfer value for each interface, assuming that the domain boundary is similarly defined in the other corresponding code. Again, however, this is not always the case. One model

may divide the domain interface into multiple sections, or layers, and collect zeroth-order information for each. These multiple individual values will then need converted into a value, or set of values, compatible to the other codes' models within the multiphysics framework.

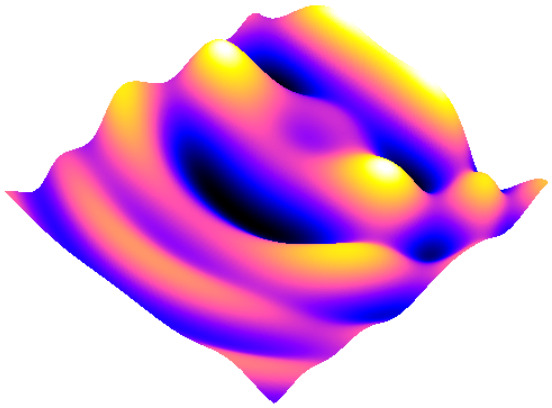
2.3 Functional Expansions

Many multiphysics M&S R&D projects include a directive to develop and advance current, new, or novel coupling mechanisms. Functional expansions come from a recent branch of research seeking to identify desirable alternatives to zeroth-order data quantification and communication. Such zeroth-order techniques include histogram bins, meshed data, and integrated average quantities. Initial functional expansion (FE) research was based on methods already established in theory but revolutionary in implementation [75, 76].

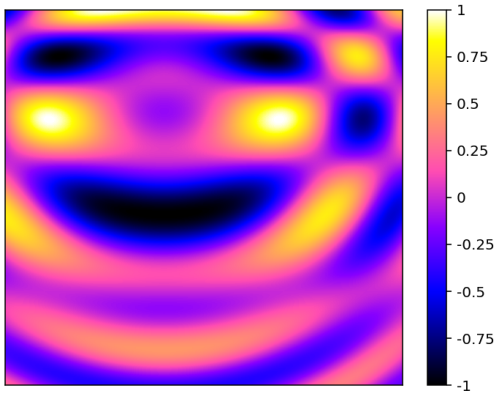
A functional expansion (FE) is a representation of data using a series of functions, similar to a representation of periodic data by a Fourier series. These representations are defined by sets of coefficients to the function series. Suitable basis sets include orthogonal polynomial series such as the Legendre and Hermite polynomials [77]. Coincidentally, bounded Fourier series can also be used as a functional basis set.

A derivative concept is the functional expansion tally (FET), which is an application of functional expansions to represent a distribution of discreet events generated by a simulation. It is important to establish a usage distinction between these two terms. FEs use an integral equation, which can be evaluated explicitly or numerically, to generate the coefficients of representation. FETs are an extension of FE methodologies for use in a MC context, the difference being that the coefficient-generating integral equation is evaluated via stochastic sampling. The principles of both approaches are provided in chapter 3. For now, it is sufficient to establish that these methods are especially suitable for representing continuous data. In support of this claim, fig. 2.1 shows a 2D FE built with Legendre polynomials.

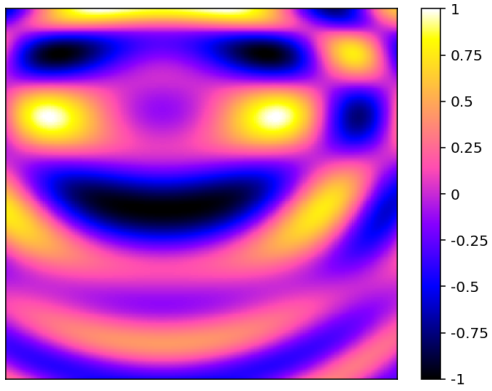
Under most conditions, FETs also provide superior convergence and uncertainty reduction when compared to a mesh tally [76]. This is illustrated in fig. 2.2. The weaknesses of using FETs are exposed when the underlying data: 1) are collected over an unusual geometry,



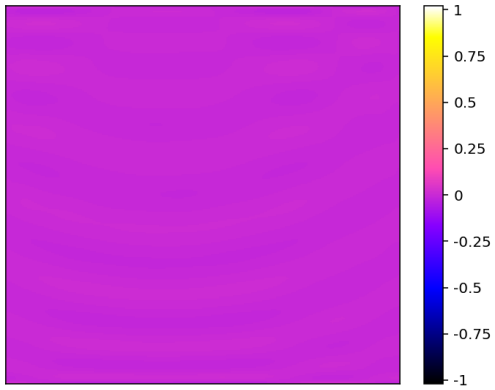
(a) 3D representation of the fabricated solution



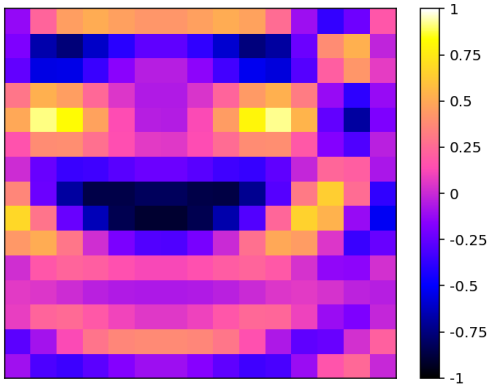
(b) 2D plot of the fabricated solution



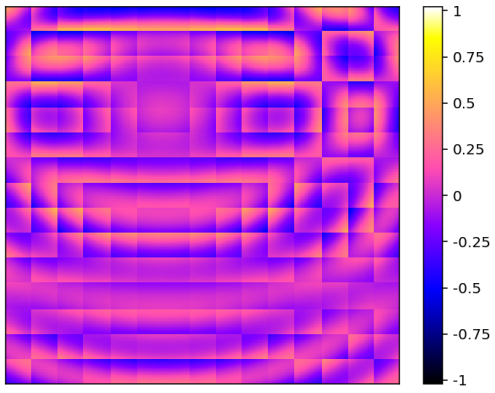
(c) Optically resolved FE representation using 225 coefficients



(d) The FE difference compared to the solution



(e) Mesh representation using 225 elements



(f) The mesh difference compared to the solution

Figure 2.1 – Comparison between a fabricated solution field, a 2D functional expansion based on Legendre polynomials, and a mesh representation using the same number of data points as the FE.

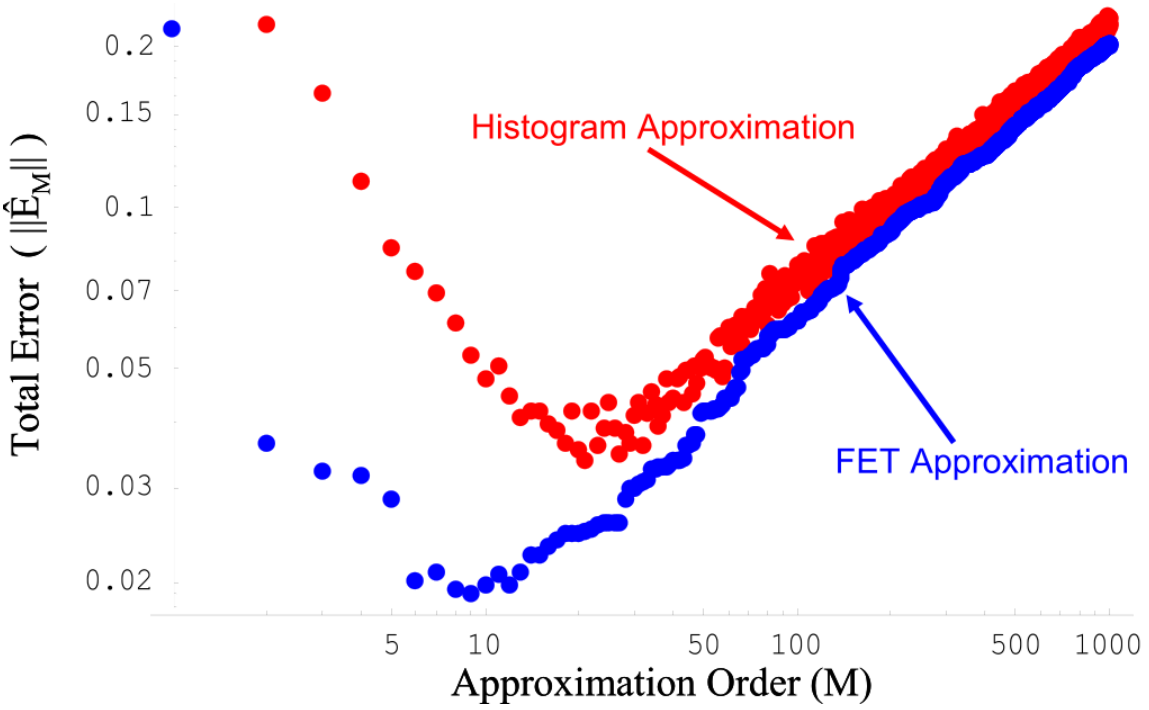


Figure 2.2 – A comparison of the total error present in representations of a stochastically sampled data set plotted against the approximation order. For mesh tallys this represents the number of evenly-spaced bins; for FETs this represents the functional order. Each data point was generated from identical MC simulations containing 10 000 histories. Image used with permission from *Functional Expansion Tallies for Monte Carlo Simulations* [76].

2) contain numerous steep gradients, or 3) are piecewise continuous. Conformal mapping techniques may be applied to transform the data from an unusual geometry into a shape supported by the underlying functional basis. In the case of steep gradients, the comparison mesh tally must have expertly- or serendipitously-placed bin edges to also capture the distribution features. However, under both circumstances a piecewise FET can outperform both the mesh tally and continuous FET [76, Ch. 3].

Many references to FE/FET implementations are high-fidelity data storage mechanisms, from cross section input formats to output tally/detector results [58, 78–80]. Other applications similar to FETs have yielded results such as accelerated reduced source sampling [81]. Studies reconfirm that there are an optimal number of coefficients and geometrical regions that can be used to represent a distribution [71].

Another group tested FETs in the MC simulation of a large region containing material discontinuities. This test included a parametric study on both the functional basis and

number of coefficients used. Legendre polynomials were shown to generally perform better than their Laguerre counterpart. A higher number of coefficients enabled a higher resolution in the resulting distribution, but it also introduced significant levels of noise into the results [82]. A generalized lesson learned is that unphysical results may occur, such as regions with negative values, when using too many coefficients. Fortunately, when an orthogonal function series is used as the basis set—which is often the case—then any unreasonable terms can be conveniently discarded without impacting the remaining coefficients. Methods for checking the validity of these terms was commented on previously [76].

2.4 FE-based Methods in Multiphysics

In one of the earliest multiphysics applications, functional expansions based on Zernike polynomials were used in Denovo, a deterministic neutron transport code, to: 1) smooth a coarse-mesh deterministic power profile, and 2) establish a communication mechanism in a multiphysics environment with dissimilar meshes. Specifically, a functional expansion derived from a coarse mesh was compared to the solution generated using a fine mesh. Denovo was able to perform the coarse-mesh solve much faster than the fine-mesh solve. The final outcome was positive, demonstrating a 50× speed-up with "very little loss in accuracy" [71].

Further developments using FE-based methods for full multiphysics coupling have recently surfaced [83]. OpenMC, a testbed MC code spearheaded by Massachusetts Institute of Technology (MIT), was coupled to the deterministic multipurpose MOOSE framework using FETS. The MOOSE results were then coupled back into OpenMC with FE derived from the finite element analysis solution. The relevant aspects include:

- Steady-state 2D FE-based multiphysics coupling
- Utilization of Legendre and Zernike polynomials as functional basis sets
- Testing based on the MIT benchmark for evaluation and validation of reactor simulations (BEAVRS) geometry and material definitions [84]

- Investigation of the impact on fidelity and convergence by varying simulation parameters in a MC simulation
  - FET orders
  - Sample population sizes

However, as yet each of these FES-in-multiphysics applications have been performed only with localized implementations. The creation of a generalized FE-based coupling framework into MOOSE, which can be used as a multiphysics code driver, is presented in chapter 6. Such an implementation allows MOOSE to act as a hub for connecting various FE-enabled codes for multiphysics M&S based purely on coefficient transfers.

2.5 Modern Challenges

The grand expectation of M&S is to support both existing and future technologies. For example, relicensing and longeterm operation of the current nuclear reactor fleet is depending on a concerted effort utilizing both high-fidelity M&S and experiments. Specifically, the industry is seeking to demonstrate a capability to operate nuclear power plants far longer than originally anticipated, while still improving margins of safety [85]. These include performance, reliability, and economics. One key component of establishing these characteristics as part of any R&D process is through high-fidelity multiphysics M&S [86]. Despite the need for high-fidelity models, and the accompanying R&D performed to provide solutions, many challenges still exist in performing multiphysics simulations [3, 25, 70, 87, 88].

2.5.1 Case Study: ATF, TREAT, and MAMMOTH

An essential path to establishing the longevity and viability of current and future nuclear reactors is the development of accident tolerant fuels. Accident tolerant fuels (ATFS) are being engineered to maintain integrity while undergoing the extreme reactor conditions resulting from beyond design basis accident scenarios [52, 89]. The paramount importance of developing ATFS was punctuated by events in Japan during March of 2011. A combined earthquake and tsunami natural disaster duo led to a sequence of unfortunate events at the

Fukushima Daiichi nuclear power plants that resulted in fuel failures, radioactive releases, and widespread public panic.

In response to these demonstrated needs, the US Department of Energy (DOE) has established an ATF development program. This program is tasked with developing new fuels that can be emplaced in current nuclear reactors by 2020 [90]. A fundamental component of this program is a rigorous testing programme to ensure accident tolerant characteristics. The Idaho National Laboratory (INL) is leading the development program, leveraging its expertise and existing facilities.

One significant player at INL is the Advanced Test Reactor (ATR) complex, housing a uniquely designed reactor. The reactor itself enables the performance evaluation of fuels and materials in meticulously designed experimental modules with carefully controlled and characterized conditions [91]. Additional companion locations at INL, such as the Materials and Fuels Complex (MFC) and Center for Advanced Energy Studies (CAES), complement this capability with world-class destructive and nondestructive analysis facilities and researchers.

Another reactor at INL, Transient Reactor Test Facility (TREAT), was brought back online in November 2017. It is designed to perform a broader range of transient tests, capable even of pushing materials to failure in well controlled and characterized conditions. These tests are focused on next-generation fuel development and certification [92]. TREAT, placed into standby in 1994, was designed specifically for pulsed transient experiments. In practice, each experiment required numerous correction factors to perform transient analysis calculations. These were approximated using several time-consuming low-power operations before and after each experiment. Now back online in the modern era, the availability of multiphysics methods enable M&S efforts to reduce the operational burden of each experiment; researchers at INL are engaged in developing a full multiphysics modeling capability [63]. An emphasis has been placed on using M&S to ensure effective and safe operations while minimizing time and costs [93].

In order to accomplish these tasks, an ideal multiphysics model of TREAT must satisfy the following requirements:

- Validated core neutronics within the temporal operational regime of TREAT

  - Pulse duration
  - Time-dependent energy release
- Neutronic distinction between the TREAT reactor core and experiment vessel
- Capture material changes, transitions, or migrations
  - Temperatures
  - States of matter
  - Crystalline phases
  - Chemistry
  - Isotopics

As a result, a representation of the TREAT reactor has been assembled utilizing MAMMOTH, a multiphysics code based on the MOOSE framework [93, 94]. MAMMOTH itself is a herd (a collection of multiple MOOSE applications) for reactor physics analysis. This herd consists of neutronics (Rattlesnake), material and fuel performance (MARMOT and BISON), and thermal hydraulics (RELAP-7) applications [72].

The beauty of MAMMOTH is that, because all the subset codes are MOOSE-based applications, multiphysics coupling is natively supported and orchestrated via the standard input files. Rattlesnake generates the neutron flux and power profiles, which are then fed into RELAP-7 and BISON. RELAP-7 takes the power profile and uses it to solve the temperature and fluid profiles in the core. BISON uses the neutron flux with the temperature and power profiles to calculate material and geometry changes in the core. These feed back into the inputs for Rattlesnake and RELAP-7 for additional simultaneous solves until convergence is reached.

One current challenge facing MAMMOTH-based modeling of TREAT centers on one of the underlying applications: Rattlesnake, a diffusion-based deterministic neutron transport solver. The cross-sections and geometry homogenization inputs for Rattlesnake must be generated by an external program with access to nuclear data libraries and core geometry. The reactor physics code Serpent (Leppänen *et al.* [37]) is often used to generate these

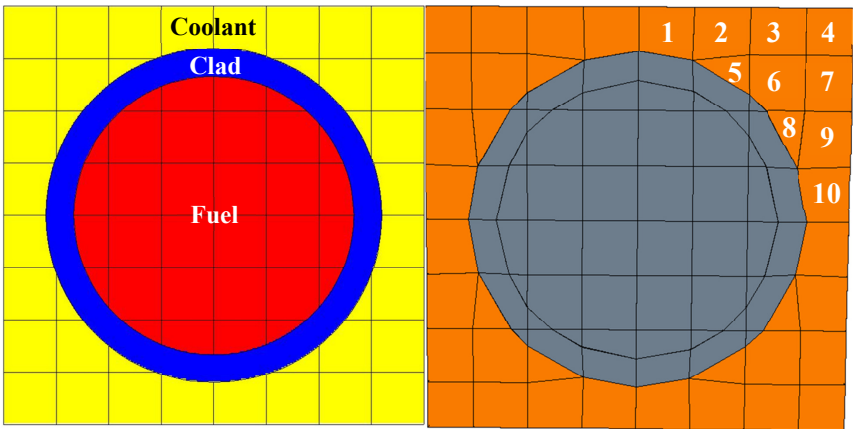
inputs for Rattlesnake. Interestingly enough, Rattlesnake calculations are frequently cross-checked against Serpent simulations for verification. Further, the core construction of TREAT introduces geometrical anisotropies that are difficult for a diffusion code like Rattlesnake to handle natively without artificial modification of parameters [63].

These deficiencies are a handicap to high-fidelity simulations, even though the coupling of Rattlesnake and MAMMOTH was straightforward because of the MOOSE commonality. Rattlesnake may be best suited, perhaps, for investigative M&S where simplicity and speed are desired. However, high-fidelity multiphysics M&S for the TREAT reactor requires a solution that can provide the necessary physics while effectively coupling to the MOOSE framework. Much R&D has been put into solving such issues that plague deterministic codes, but a large number of approximations and assumptions (such as angular discretization) are still required. Monte Carlo codes, on the other hand, are capable of naturally using continuous data, such as pointwise-continuous cross-sections and spatially-varying data, throughout the simulation process. These characteristics are ideal for high-fidelity multiphysics M&S of reactor cores.

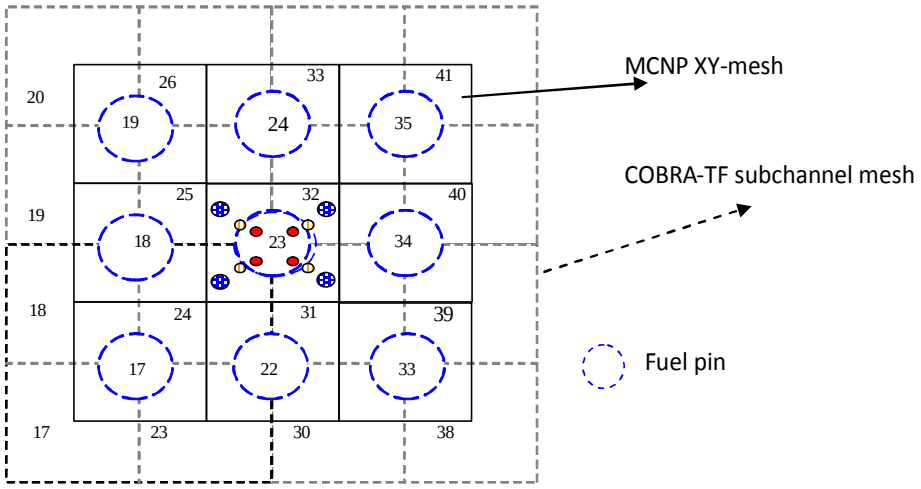
2.5.2 Case Study: Data Mapping between Dissimilar Geometries

A fundamental requirement of multiphysics is the capability to sensibly move data between different models. To ensure accurate and precise coupling, the data contained in different model constructions must be communicated with care.

Conservation of extensive property values must be maintained throughout any sort of mapping/conversion process. A failure to account for the entirety of the property valuation will change the apparent state in the other coupled models. For example, if a mapping methodology fails to correctly account for mass transfer through a volume or over an interface then the simulation may begin to exhibit unrealistic feedback effects related to a deficiency in total system mass. In such circumstances a scaling factor is often introduced that normalizes the property to an expected value, completely disregarding the inconsistency that lead to the deficiency in the first place.



(a) Mostly similar meshes, with MCNP5 on the left and STAR-CCM+ on the right (from *Nuclear Reactor Multi-Physics Simulations With Coupled MCNP5 and STAR-CCM+* [57])



(b) Completely dissimilar meshes (from *Development of a Coupling Scheme Between MCNP and COBRA-TF for the Prediction of the Pin Power of a PWR Fuel Assembly* [56])

Figure 2.3 – Model construction differences between different codes used in actual multiphysics coupling simulations.

Conversely, intensive property evaluations must accurately reflect the actual state of the fields they represent. This becomes an issue, for example, when transferring the information of a spatially-varying temperature field. A single average value conserves only zeroth-order information. This approximation may be acceptable if the field values are near-uniform or the feedback mechanisms are very weak. If these assumptions are unfounded, however, then the use of this zeroth-order value will bias the simulation outcome away from the true solution.

Figure 2.3 uses previously published multiphysics coupling scenarios to illustrate these issues. In fig. 2.3a, two similar meshes are presented. The left mesh was used in the MC neutronics code MCNP5, and the right mesh from the thermal-hydraulics code STAR-CCM+. The mesh used for STAR-CCM+ was generated internally, and then a roughly equivalent MCNP model was generated using a helper Fortran program. Information was passed back and forth between cells with the nearest centroids, although not all the cells have equivalent volumes. This means that the MCNP fission heat generation had the potential to create 'hotspots' in the fractional cells that have a volumetrically smaller counterpart in the STAR-CCM+ mesh. Further, the cladding temperature is transferred as a volume weighted average, which will ignore any temperature profile shaping in the cladding region. In fig. 2.3b only average values can be used because of the dissimilarity. This severely constrains the maximum fidelity of the simulation.

Chapter 3

Principles of Functional Expansion Methods

> *All the truths of mathematics are linked to each other, and all means of discovering them are equally admissible.*
>
> — Adrien-Marie Legendre

The use of series in mathematical analysis, whether constructed of values or functions, is not a new concept by any stretch of the imagination. Infinite series and other equivalent concepts have been used throughout history to solve some of humankind's most fundamental problems.

One of the first known applications of functional expansions principles used Fourier series to represent a distribution. In his work on random noise, Rice used an approach similar to functional expansions and quantified the distribution of noise present in an electrical signal as a Fourier series expansion:

$$I(t) = \frac{a_0}{2} + \sum_{n=1}^{N} \left(a_n \cos\left(\frac{2\pi n t}{T}\right) + b_n \sin\left(\frac{2\pi n t}{T}\right)\right)$$

for the random current noise $I(t)$ measured over the interval T.

Later, functional expansions were implemented in MC processes. The earliest known reference was in 1975, in which functional expansions techniques were used to estimate the

spherical harmonics expansion coefficients representing the angular distribution of x-ray photoemission in electron transport simulations [75]. The motivating case for developing FETs was MC variance reduction and reduced sample size. It was further acknowledged that expansion into orthonormal polynomials was a powerful technique that had been applied to solve many classes of problems in 'mathematical physics.' However, the existing approach was extended into the realms of MC sampling to directly calculate statistical estimates of the expansion coefficients. Subsequently, the first known use of functional expansions as functional expansion tallies for neutron transport MC simulations was published in 1984. FETs were used to estimate the angular and spatial distributions of the neutron flux in 1D shielding problems. One significant result was the demonstration that the FET's statistical variance was much lower—by a factor of 3 on average—than the traditional tally's variance.

FETs techniques have not yet found wide-spread adoption, although the number of instances are growing. The most likely reason is lack of exposure; FET techniques have not yet been implemented in the de-facto standard neutron transport code, Monte Carlo N-Particle (MCNP), nor is implementation likely in the near future. Until this point, the primary source of information about the behavior of FETs lay in the qualitative claims of the researchers. Chadsey *et al.* [75] had provided a statement on evaluating FET uncertainties, which was perpetuated in Noel and Wio [96]; however, it was originally presented using canonical statistical rationale rather than being forged from the properties of the functions themselves. Not until Griesheimer's PhD dissertation *Functional Expansion Tallies for Monte Carlo Simulations* in 2005 were the statistical properties and fundamental behaviors of FETs firmly established mathematically.

3.1 Fundamentals

Given a distribution F and a set of basis functions ψ, there exists a functional expansion coefficient set Λ such that:

$$F \wedge \psi \implies \Lambda = \{a_0, a_1, \ldots\} \tag{3.1}$$

where a_i are the expansion coefficients, or moments, of ψ over F. These moments are calculated as:

$$a_i = \int \psi_i(x)\rho_\psi(x)F(x)dx \tag{3.2}$$

where $\rho_\psi(x)$ is the weight function of ψ. The coefficient set Λ and basis function set ψ can be used to recreate the distribution F as:

$$F \propto \Lambda \cdot \psi = \sum_{i=0}^{\infty} a_i \psi_i \tag{3.3}$$

The proportionality is based on the orthonormalization of ψ, as defined in section 3.2.2.

However, it is highly improbable that explicitly evaluating an infinite number of terms will be realistic within the lifetime of this universe. Thus, truncated functional expansions are used instead. The error of the approximation involved with this approach is discussed in section 3.4.2.2. Interestingly, the application of truncated functional expansions provides properties both advantageous and obstructive. A discussion of these tradeoffs is found in section 3.4.3.

Order

The term 'order' will be used herein referring to the functional term number, or degree. The order corresponds to the highest power of the non-zero terms in a function's definition. This order of a term is indicated by the subscript. For example, the function P_0 is a zeroth-order Legendre polynomial, while P_3 is the third-order Legendre polynomial.

3.2 Orthogonal Functions

Orthogonal function series have traditionally been used in the application of functional expansions, although this is not a mathematical requirement. Using an orthogonal function basis simply equips the functional expansion to represent an unknown distribution accurately; as an orthogonal function series, each term ψ_i contributes unique and independent information to the solution. Thus, although not a requirement of functional expansion

methods, it must be understood that throughout this work that *only* orthogonal functions are assumed.

Further, in the computational world there are two specific properties of orthogonal functions that make them highly desirable: recurrence relations and standardization. These properties: 1) enable computational optimizations for improved algorithmic performance, and 2) provide a bounded domain for mapping transformations between the spaces of a functional expansion and the computer model. Many polynomial-class function series are either inherently orthogonal or can be orthogonalized [97].

Recurrence Relations

Recurrence relations are a property of a series that enables the calculation of a term using the values of previously evaluated terms. In practice, recurrence relations allow algorithmic implementations to quickly iterate through the series and calculate the required values in a single pass. The usual form of the recurrence relation for a polynomial function set ψ is [5, p. 435]:

$$\psi_i(x) = \left(A_{\psi_i} x + B_{\psi_i}\right) \psi_{i-1}(x) + C_{\psi_i} \psi_{i-2}(x) \tag{3.4}$$

with series- and order-specific constants A_{ψ_i}, B_{ψ_i}, and C_{ψ_i}. Initial cases are always provided, often of the form $\psi_0(x) = 1$ and $\psi_1(x) = x$.

Standardization

Most orthogonal function sets ψ are intrinsically standardized, viz. are a function set $\psi(x)$ scaled such that $\left|\psi(x)\right| \leq Y$ over the interval $[\mathbf{x}_1, \mathbf{x}_2]$; typically $Y = 1$ and $\mathbf{x}_2 = -\mathbf{x}_1 = 1$. The standardization of a functional set ψ over a specified interval is a property that is essential to implementation of FETs for an arbitrarily-sized geometry.

3.2.1 Mathematical Basis

The following discussion is adapted from Advanced Engineering Mathematics, *pp. 500–501.*

A functional set $\psi = \{ \psi_i \mid i = 0, 1, 2, \dots \}$ is considered orthogonal over the interval $[x_1, x_2]$ if it satisfies the following relation:

$$\int_{x_1}^{x_2} \psi_i(x)\psi_{i'}(x)\rho_\psi(x)dx = \left\|\psi_i\right\|^2 \delta_{i,i'} = \left\|\psi_{i'}\right\|^2 \delta_{i,i'} \tag{3.5}$$

where $\delta_{i,i'}$ is the Kronecker delta:

$$\delta_{i,i'} = \begin{cases} 0 & i \neq i' \\ 1 & i = i' \end{cases} \tag{3.6}$$

and $\left\|\psi_i\right\|^2$, the ℓ^2 norm of ψ_i, is non-zero and defined as:

$$\left\|\psi_i\right\|^2 = \int_a^b \psi_i^2(x)\rho_\psi(x)dx \tag{3.7}$$

In other words, a functional series is considered orthogonal if eq. (3.5) evaluates to zero for all $i \neq i'$. Note that for many functional series (Legendre, Fourier) the weight function $\rho(x) = 1$. Under this condition eq. (3.5) can be simplified as:

$$\int_{x_1}^{x_2} \psi_i(x)\psi_{i'}(x)dx = \left\|\psi_i\right\|^2 \delta_{i,i'} = \left\|\psi_{i'}\right\|^2 \delta_{i,i'} \tag{3.8}$$

Based on these relations, a definite form of eq. (3.2) can be rewritten as:

$$a_i = \int_{x_1}^{x_2} \psi_i(x)\rho_\psi(x)F(x)dx \tag{3.9}$$

The domain of the distribution F in simulation space is rarely the same as the orthogonal series domain, and must often be mapped into the space defined by $[x_1, x_2]$. Fortunately this is a lossless and simple linear transformation.

3.2.2 Orthonormal Variants

The following discussion is adapted from Advanced Engineering Mathematics, *pp. 504–509.*

A set of orthonormal relations are required to close the proportionality loop between eqs. (3.3) and (3.9). The decorator $\widetilde{abc}$ will be to indicate an orthonormalized component.

First, eq. (3.3) is rewritten as an equality by inserting a constant:

$$F = \sum_{i'=0}^{\infty} a_{i'} c_{\psi_{i'}} \psi_{i'} \tag{3.10}$$

This result is then inserted into eq. (3.9) and simplified:

$$\begin{aligned} a_i &= \int_{x_1}^{x_2} \psi_i(x) \rho_\psi(x) F(x) dx \\ &= \int_{x_1}^{x_2} \psi_i(x) \rho_\psi(x) \left(\sum_{i'=0}^{\infty} a_{i'} c_{\psi_{i'}} \psi_{i'}(x) \right) dx \\ &= \sum_{i'=0}^{\infty} a_{i'} c_{\psi_{i'}} \int_{x_1}^{x_2} \psi_i(x) \psi_{i'}(x) \rho_\psi(x) dx \end{aligned} \tag{3.11}$$

$$a_i = a_i c_{\psi_i} \left\| \psi_i \right\|^2 \tag{3.12}$$

Because of orthogonality in ψ the only non-zero term in eq. (3.11) is when $i = i'$. Upon inspection of eq. (3.12), the term $c_{\psi_i} \left\| \psi_i \right\|^2$ must equal unity, resulting the complete solution:

$$c_{\psi_i} = \left\| \psi_i \right\|^{-2} \tag{3.13}$$

The constant c_{ψ_i} is known as the orthonormalization constant, and has an explicit form for each polynomial series ψ based on the order i. Later, these forms will be specified along with the definitions of the actual polynomial series.

For clarity, the closed forms of eqs. (3.3) and (3.9) will be rewritten using the coefficient b_i:

$$b_i = \int_{x_1}^{x_2} \breve{\psi}_i(x) \rho_\psi(x) F(x) dx \tag{3.14}$$

$$F(x) = \sum_{i=0}^{\infty} b_i \psi_i(x) \tag{3.15}$$

with the implied orthonormal relationships:

$$\breve{\psi}_i = \frac{\psi_i}{\left\| \psi_i \right\|^2} = c_{\psi_i} \psi_i \tag{3.16}$$

$$b_i = c_{\psi_i} a_i \tag{3.17}$$

These orthonormalized coefficients can also be used to compose a coefficient set Λ, just like a_i are used in eq. (3.1).

Orthonormality

In essence, the orthonormal relationships ensure that each term contributes equally to the representation in ℓ^2 norm space. While the range of an orthogonal functional set ψ is typically standardized, the orthonormal variant $\breve{\psi}$ is not. Scaling by c_{ψ_i}, as shown in eq. (3.16), causes greater minima and maxima over the standardized interval $[\mathbf{x}_1, \mathbf{x}_2]$, i.e., the extrema of $\breve{\psi}_i(x)$ are equal-to or greater than the extrema of $\breve{\psi}_{i-1}(x)$.

3.2.3 Legendre Polynomials

Legendre polynomials are employed a wide range of established uses. As such, a number of definitions exist for generating the polynomial orders P_l. One compact naive form is:

$$P_l = \sum_{n=0}^{l} \binom{l}{n} \binom{-l-1}{n} \binom{1-x}{2}^n \tag{3.18}$$

where the binomial coefficient is calculated as:

$$\binom{a}{b} = \frac{a!}{b!\,(a-b)!} \tag{3.19}$$

The subscript l is adopted to indicate individual polynomial terms P_l.

Note that the Legendre set of polynomials as defined by eq. (3.18) are inherently standardized, ranging between $[-1, 1]$ over the bounded domain $[-1, 1]$. Figure 3.1 plots the first thirteen orthogonal Legendre polynomials in this functional space.

3.2.3.1 Recurrence Relation

The recurrence relation for calculating a Legendre polynomial is:

$$P_0 = 1 \tag{3.20}$$

$$P_1 = x \tag{3.21}$$

$$P_l(x) = \frac{(2l-1)\,xP_{l-1}(x) - (l-1)\,P_{l-2}(x)}{l} \tag{3.22}$$

which is numerically stable within the range of orders that are practical for FE methods.

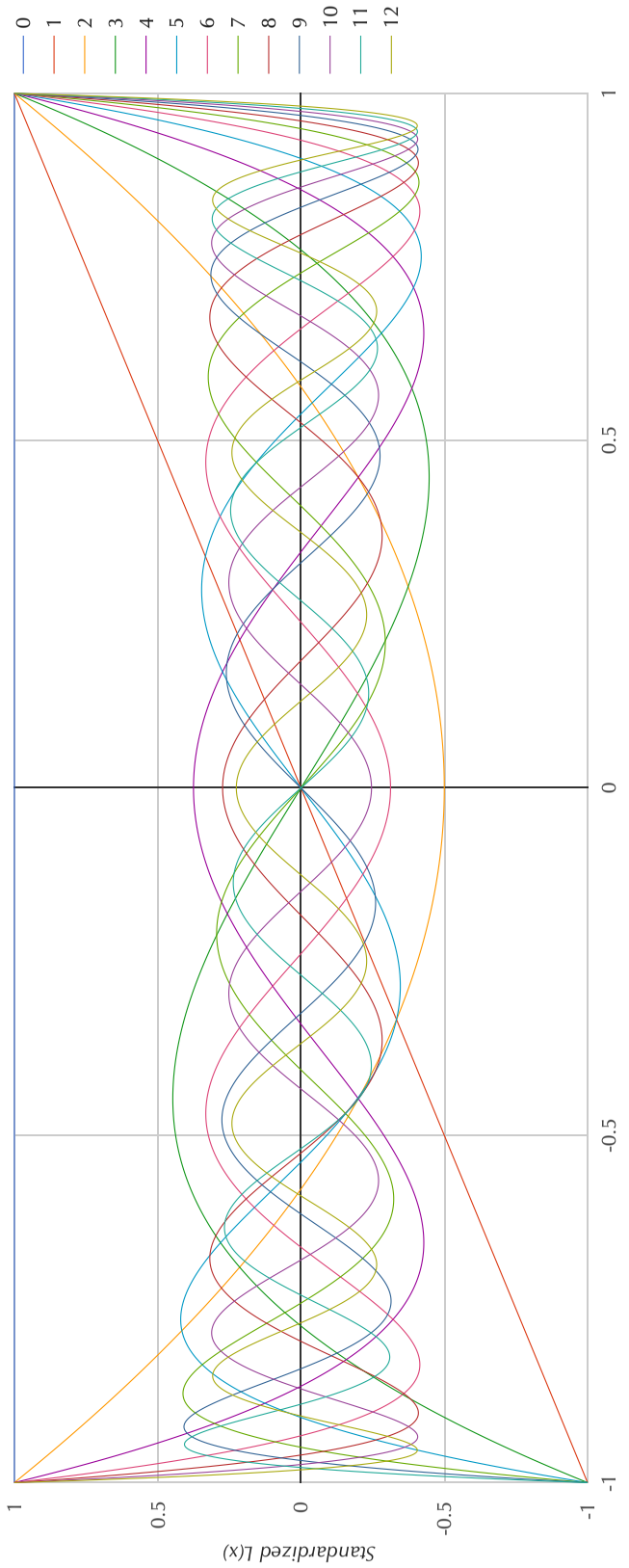


Figure 3.1 – Plot of the first thirteen orthogonal Legendre polynomials $\{P_0(x), P_1(x), \ldots, P_{12}(x)\}$.

3.2.3.2 Orthonormalization

The Legendre polynomials obey the orthonormal relationship:

$$\int_{-1}^{1} P_l(x) P_{l'}(x) \rho_P(x) dx = \frac{2}{2l+1} \delta_{l,l'} \tag{3.23}$$

The form of the orthonormalization constant c_{P_l} can be extracted form this known relationship; when eq. (3.23) is compared to eq. (3.8), the ℓ^2 norm is found to be:

$$\|P_l\|^2 = \frac{2}{2l+1} \tag{3.24}$$

yielding the Legendre orthonormalization constant c_{P_l} as:

$$c_{P_l} = \frac{2l+1}{2} \tag{3.25}$$

The resulting orthonormalized Legendre relationships, corresponding to eqs. (3.16) and (3.17), are:

$$\breve{P}_l(x) = \frac{2l+1}{2} P_l(x) \tag{3.26}$$

$$b_l = \frac{2l+1}{2} a_l \tag{3.27}$$

3.2.4 Zernike Polynomials

Zernike polynomials are a specialized class of 2D orthogonal polynomials on the unit disk. As polynomials on the unit disk, they are functions of radius r and azimuthal angle ϕ, respectively over the intervals $[0, 1]$ and $[0, 2\pi)$. They are named in honor of Fritz Zernike, a Dutch physicist, who pioneered their use to characterize wavefront aberrations in optical systems. Zernike is often more well-known as the inventor of the phase-contrast microscope, for which he won the Nobel Prize for Physics in 1953 [98].

An excellent review of Zernike polynomials and their properties can be found in a 2011 review by Lakshminarayanan and Fleck, published in Journal of Modern Optics. Although the Zernike polynomials are commonly used in optics application, they exhibit an optimal set of characteristics in the unit disk that have resulted in an expanded use far beyond their original design. A 2D representation is of the Zernike polynomials is shown in fig. 3.2.

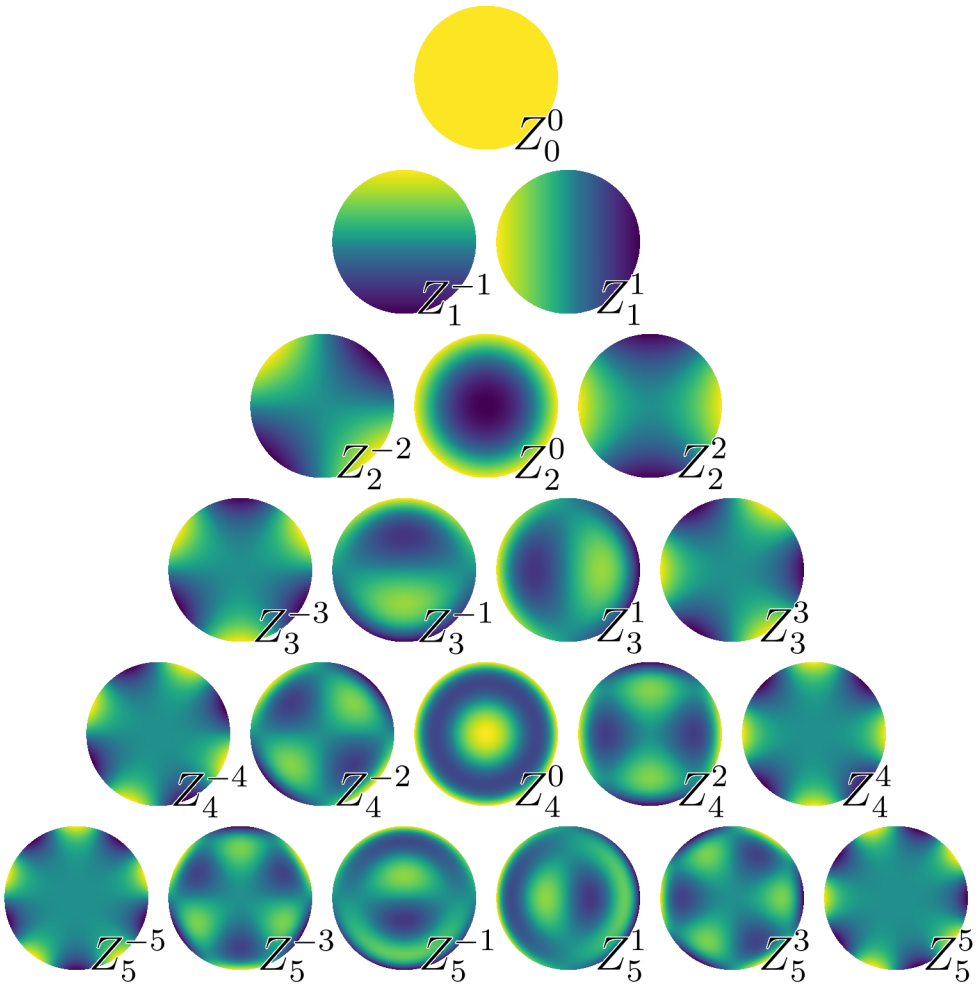


Figure 3.2 – Plot of the first twenty-one Zernike polynomials showing the full angular distribution.
Attribution: From Zernike polynomials2 by Zom-B / CC BY 3.0, modified in GIMP to use 'viridis' colormap

3.2.4.1 Definition

Because Zernike polynomials are 2D, two terms are required to identify the dimensionality of a particular term. 'Order' will be used for the radial polynomial degree and represented by the subscript n or p. 'Rank' will be used for the azimuthal angular frequency and represented by the superscript m or q. The orders n and p are equivalent in all respects, only separated for contextual clarity. Namely, the notations n and m are used in contexts where the rank m is allowed to be negative. Conversely, p and q are used in contexts where the rank q is always non-negative.

Formally, the order and rank representations are defined as:

$$n \in \mathbb{N} \tag{3.28}$$

$$m \in \mathbb{Z} \tag{3.29}$$

$$|m| \leq n \tag{3.30}$$

$$n - |m| \text{ is even} \tag{3.31}$$

The rank q is always positive and still bounded by the order p:

$$q \leq p \tag{3.32}$$

$$\left|q\right| = q \tag{3.33}$$

Ps and Qs

An advantage of the positive-only notation using p and q is that it recognizes the reflective property of the Zernike polynomials. Only the positive rank radial polynomials are required to be calculated when evaluating a large set of Zernike polynomials. The results can then can be mapped over and reused for the negative-rank polynomials, with the azimuthal component providing the orthogonality condition. For example, inspection of fig. 3.2 will reveal the radial symmetry between the positive- and negative-ranked terms.

This characteristic can be leveraged when optimizing computational evaluations.

The radial and azimuthal variables are independent. As such, each can be defined and operated on separately. The radial polynomials $R_n^{|m|}(r)$ are:

$$R_p^q(r) = \sum_{k=0}^{\frac{p-q}{2}} (-1)^k \binom{p-k}{k} \binom{p-2k}{\frac{p-q}{2}-k} r^{p-2k} \tag{3.34}$$

The azimuthal components are represented as:

$$\Phi_m(\phi) = \begin{cases} \cos(m\phi) & m > 0 \\ 1 & m = 0 \\ \sin(|m|\phi) & m < 0 \end{cases} \tag{3.35}$$

The combined naive definition of the Zernike polynomials is:

$$Z_n^m(r,\phi) = \begin{cases} R_n^m(r)\Phi_m(\phi) & m > 0 \\ R_n^0(r) & m = 0 \\ R_n^{|m|}(r)\Phi_m(\phi) & m < 0 \end{cases} \tag{3.36}$$

3.2.4.2 Recurrence Relations

Unfortunately, and notwithstanding powerful hardware, the computational requirement for calculating a single Zernike polynomial Z_n^m using eqs. (3.34) and (3.36) is relatively time consuming and inefficient. Using the naive form, the number of floating point operations (flops) required to compute all polynomials Z up through a certain order N increases exponentially as $\mathcal{O}(N^3)$.

Many have devoted extensive research into developing computationally fast numerical algorithms for the Zernike polynomials. This is a natural yet critical requirement for implementation in any code—further discussion is provided in section 3.3.3 with regards to use in functional expansions applications. In summary, no 'best' algorithm exists. Rather the optimal choice depends on the application and use [5, p. 436].

Zernike Savings

All known 'fast' methods require at least $\mathcal{O}(n^2)$ flops, likely because the Zernike polynomials Z exist on the 2D unit disk. Conveniently, Zernike polynomials represent a 2D space more efficiently than the standard practice of melding two 1D polynomial function bases to create a new 2D basis. An order N Zernike function set approximately requires only half as many terms.

For any series order N, the Zernike polynomials functional basis uses $(N+1)(N+2)/2$ terms to represent the 2D space. A square 2D basis would use $(N+1)^2$. The result is a savings ratio of $2 - \frac{2}{N+2}$ in favor of the Zernike polynomials.

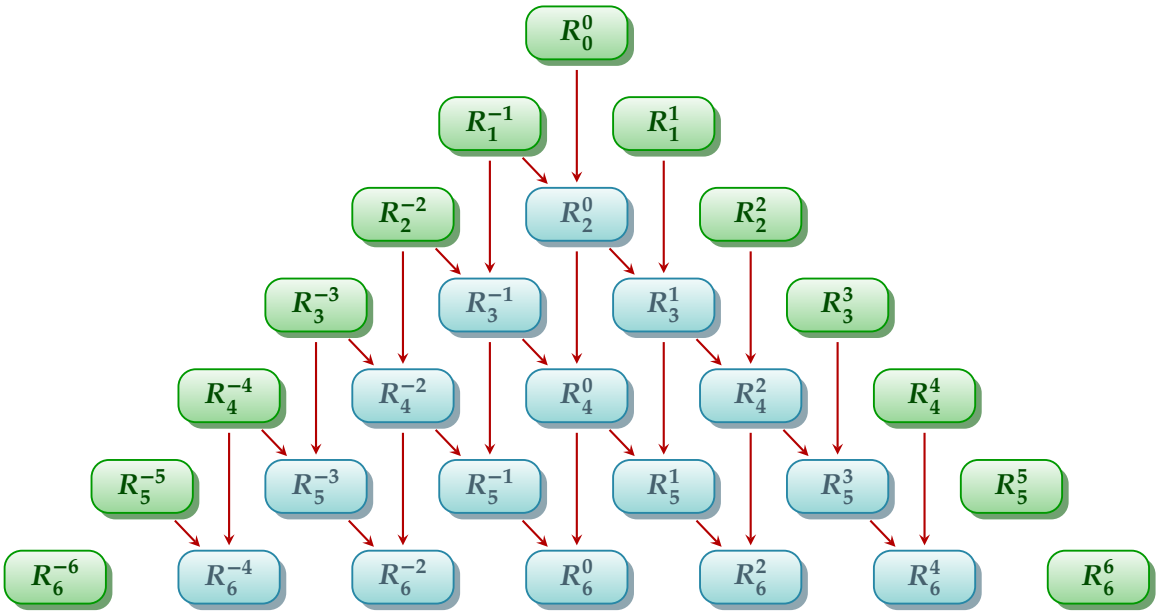


Figure 3.3 – Zernike radial polynomials: computational flow of Prata's method. The colors indicate the equation used to calculate the radial polynomial: green is eq. (3.37), and blue is eq. (3.38).

Before presenting the available recurrence relations, a standard definition is used by all implementations for the radial polynomials R_n^m when the order and rank are identical:

$$R_n^{m=n}(r) = r^n \tag{3.37}$$

Prata's Method The simplest recurrence relation, known as Prata's method, has better performance than eq. (3.36) with $\mathcal{O}(n^2)$ multiplications to calculate all Z for any order n:

$$R_n^m(r) = \frac{(2rn)R_{n-1}^{m-1}(r) - (n-m)R_{n-2}^m(x)}{n+m} \tag{3.38}$$

A diagram of the computational flow is depicted in fig. 3.3.

Although it is the simplest implementation, Prata's method is challenged by a high recursive complexity. This is observable in fig. 3.3 by the dependence of all inner radial polynomials R_n^m on previous polynomials of both different order and rank.

Modified Kintner's Method The most common recurrence relation is the Modified Kintner's method.[1] This method starts an edge nodes, i.e., nodes sharing the same order p and

[1]The original Kintner's method did not include the relation in eq. (3.40), and thus depended on a different calculation method such as direct calculation via eq. (3.34)

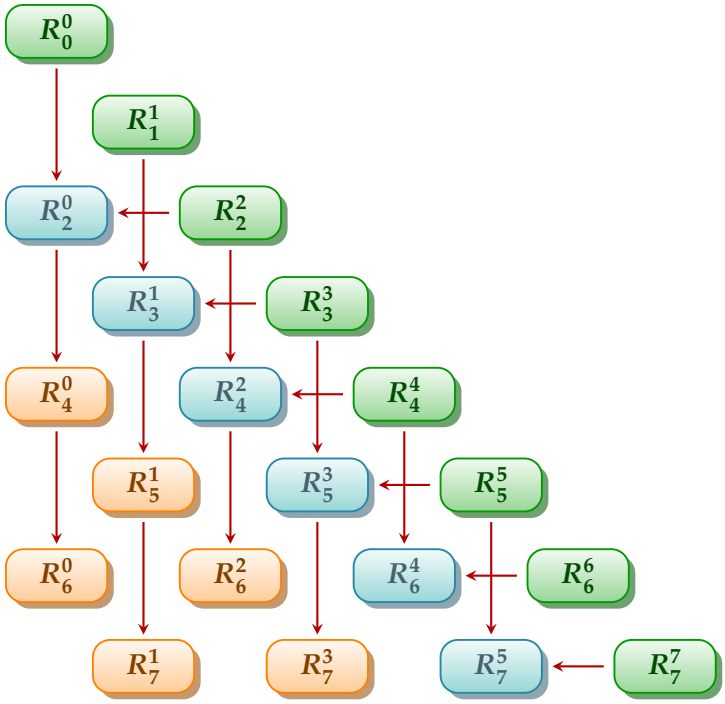


Figure 3.4 – Zernike radial polynomials: computational flow of the modified Kintner's method. Only the solutions for $m \geq 0$ is shown; the terms for $m < 0$ differ only in the angular component and can be generated from these evaluations. The colors indicate the equation used to calculate the radial polynomial: green is eq. (3.37), blue is eq. (3.40), and orange is eq. (3.41).

rank q, and recurses to the request polynomial through ascending orders p as shown in fig. 3.4. The number of steps required to calculate a specific polynomial of order p and rank q is:

$$\frac{p-q}{2}+1 \tag{3.39}$$

The definition for the radial polynomial R using the Kintner method is:

$$R_{q+2}^{q}(r) = (q+2)R_{q+2}^{q+2}(r) - (q+1)R_{q}^{q}(r) \tag{3.40}$$

$$R_{p}^{q}(r) = \frac{\left(K_2 r^2 + K_3\right) R_{p-2}^{q}(r) + K_4 R_{p-4}^{q}(r)}{K_1} \tag{3.41}$$

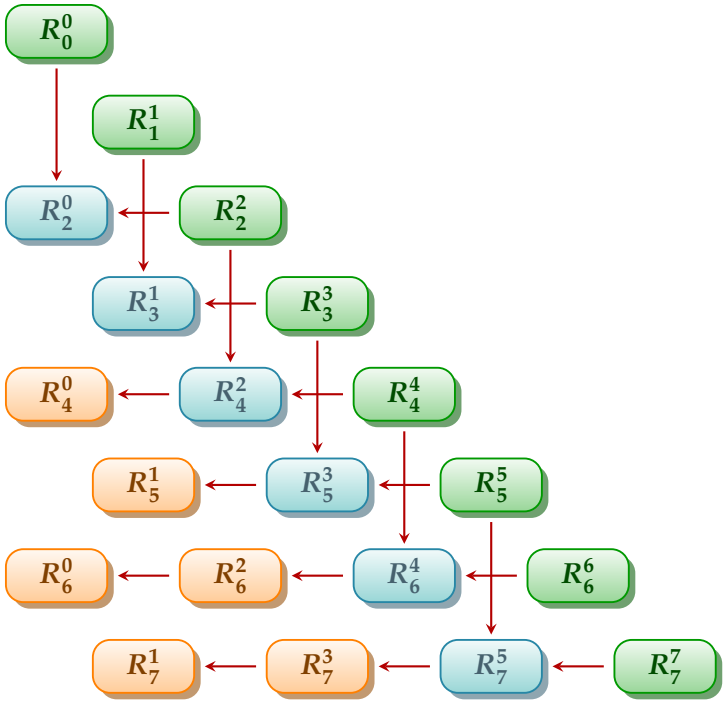


Figure 3.5 – Zernike radial polynomials: computational flow of Chong's method. Only the solutions for $m \geq 0$ is shown; the terms for $m < 0$ differ only in the angular component and can be generated from these evaluations. The colors indicate the equation used to calculate the radial polynomial: green is eq. (3.37), blue is eq. (3.42), and orange is eq. (3.43).

where:

$$K_1 = \frac{(p+q)(p-q)(p-2)}{2}$$

$$K_2 = 2p(p-1)(p-2)$$

$$K_3 = -q^2(p-1) - p(p-1)(p-2)$$

$$K_4 = \frac{-p(p+q-2)(p-q-2)}{2}$$

Chong's Method A algorithm named 'q-recursive' was presented by Chong *et al.* in 2003 [100]. Herein it will be referred to as Chong's method following the terminology set by *2D & 3D image analysis by moments* [5]. Chong's method starts its recursion from the edge nodes and recurses through descending ranks q as illustrated in fig. 3.5. The number of recurrence steps required to calculate a specific polynomial of order p and rank q is actually the same as the Modified Kintner's method, defined in eq. (3.39).

The recurrence relation for the radial polynomial R is:

$$R_p^{p-2}(r) = pR_p^p(r) - (p-1)R_{p-2}^{p-2}(r) \tag{3.42}$$

$$R_p^{q-4}(r) = H_1 R_p^q(r) + \left(H_2 + \frac{H_3}{r^2}\right) R_p^{q-2}(r) \tag{3.43}$$

where:

$$H_3 = \frac{-4\,(q-2)\,(q-3)}{(p+q-2)\,(p-q+4)} \tag{3.44}$$

$$H_2 = \frac{H_3\,(p+q)\,(p-q+2)}{4\,(q-1)} + (q-2) \tag{3.45}$$

$$H_1 = \frac{q\,(q-1)}{2} - qH_2 + \frac{H_3\,(p+q+2)\,(p-q)}{8} \tag{3.46}$$

Again, eq. (3.37) is used for the case $q = p$.

A special condition must be defined for $r = 0$ due to the r^{-2} term in eq. (3.43). Fortunately, $R_p^q(0)$ can be directly determined from the values of n and m:

$$R_n^m(0) = \begin{cases} 0 & m \neq 0 \\ -1 & n/2 \text{ is odd} \\ 1 & \text{otherwise} \end{cases} \tag{3.47}$$

Finally, note that eq. (3.40) and eq. (3.42) are equivalent in nature but defined differently; the first uses q as the indexing basis, the second uses p. These minor differences better match the recurrence relationship of each methodology. However, in a purely computational world, eq. (3.42) has one less arithmetic operation than eq. (3.40). Further, as will be seen later, Chong's method has some advantages when using an orthonormalized form of the recurrence relation. Therefore, Chong's method is the preferred option for FE implementations.

3.2.4.3 Indexing

Zernike polynomials are often remapped to a sequence requiring only a single index. This simplifies the representation when storing data in linear form, e.g., as coefficients in a

program's data array. One such mapping—the most common sequential indexing scheme—was introduced by Noll. It has properties suitable for optical applications, namely: 1) similar but opposite-ranked polynomials are neighbors, and 2) negative-ranked polynomials are assigned an odd index while positive-ranked polynomials are assigned an even index.

However, these properties are not actually needed nor convenient for the non-optical implementation of a functional expansion. First, the algorithms to map between the order-rank notation and Noll's indices are complicated and require multiple conditional checks. Second, the indexing starts at one. This may not initially seem important, but in most programming languages this requires an additional arithmetic operation for conversion to a zero-based array index.

An alternative indexing scheme was found that is much more suitable [99, p. 458]. The mapping can be performed in one line of code—without impacting readability[2]—and starts with zero as the first index. The forward conversion from order-rank notation (n, m) to single index j is:

$$j = \frac{n\,(n+2) + m}{2} \tag{3.48}$$

The reverse conversion from single index j to order-rank notation (n, m) is:

$$n = \left\lceil \frac{\sqrt{9 + 8j} - 3}{2} \right\rceil \tag{3.49}$$

$$m = 2j - n\,(n+2) \tag{3.50}$$

where eq. (3.50) requires the result for n from eq. (3.49). This mapping was verified via brute-force methods to be self-consistent through $j = 1 \times 10^9$. The first 21 mappings are shown in table 3.1.

This introduces another benefit to using Chong's method. The linear index j groups all the ranks m of an order n together, just like Chong's method recurses based on order and not on rank like Kinter's method. This makes the in-algorithm indexing implementation simpler to work with for Chong's method.

[2]One-liners can be performed in any code that uses a line-end character, such as `C` or `C#` which use the ';' character. However, just because it *can* be done means that it *should* be done, nor that the code is legible. Consider the case of *The International Obfuscated C Code Contest* [101].

Table 3.1 – The first 20 indices for sequentially mapping the Zernike polynomials

	Rank m										
Order n	-5	-4	-3	-2	-1	0	1	2	3	4	5
0						$j=0$					
1					$j=1$		$j=2$				
2				$j=3$		$j=4$		$j=5$			
3			$j=6$		$j=7$		$j=8$		$j=9$		
4		$j=10$		$j=11$		$j=12$		$j=13$		$j=14$	
5	$j=15$		$j=16$		$j=17$		$j=18$		$j=19$		$j=20$

3.2.4.4 Orthonormalization

Recalling that the variables are independent and can be operated on separately, the Zernike polynomials can be split into the radial and angular components. Further, the radial orthonormal relationship is dependent solely on the order n, while the angular component depends only on the rank m. Individual orthonormalization constants c_{R_n} and c_{Φ^m} will be developed for each component, then used to find a relation for the combined constant $c_{Z_n^m}$.

Radial Component The radial component obeys the orthogonal relationship:

$$\int_0^1 R_n(r)R_{n'}(r)\rho_R(r)rdr = \frac{1}{2(n+1)}\delta_{n,n'} \tag{3.51}$$

The form of the orthonormalization constant c_{R_n} can be extracted form this known relationship; when eq. (3.51) is compared to eq. (3.8), the ℓ^2 norm is found to be:

$$\|R_n\|^2 = \frac{1}{2\,(n+1)} \tag{3.52}$$

yielding the Zernike's radial orthonormalization constant c_{R_n} as:

$$c_{R_n} = 2\,(n+1) \tag{3.53}$$

Angular Component The angular component is, interestingly, slightly more complex:

$$\int_0^{2\pi} \Phi^m(\phi)\Phi^{m'}(\phi)\rho_\Phi(\phi)d\phi = \int_0^{2\pi} \left\{ \begin{array}{l} \cos(m\phi)\cos(m'\phi) \\ \sin(m\phi)\sin(m'\phi) \\ \cos(m\phi)\sin(m'\phi) \\ \sin(m\phi)\cos(m'\phi) \end{array} \right\} d\phi = \left\{ \begin{array}{l} \pi\,(1+\delta_{m,0})\,\delta_{m,m'} \\ \pi\delta_{m,m'} \\ 0 \\ 0 \end{array} \right. \tag{3.54}$$

From eq. (3.35) it is observed that sin() is used only for $m < 0$, likewise cos() for $m > 0$. Thus, the zero terms are implicitly so via orthogonality, and eq. (3.54) can be reduced to:

$$\int_0^{2\pi} \Phi^m(\phi)\Phi^{m'}(\phi)\rho_\Phi(\phi)d\phi = \int_0^{2\pi} \left\{ \begin{array}{l} \cos(m\phi)\cos(m'\phi) \\ \sin(m\phi)\sin(m'\phi) \end{array} \right\} d\phi = \left\{ \begin{array}{l} \pi\left(1+\delta_{m,0}\right)\delta_{m,m'} \\ \pi\delta_{m,m'} \end{array} \right. \tag{3.55}$$

The form of the orthonormalization constant c_{Φ^m} can be extracted form this known relationship; when eq. (3.55) is compared to eq. (3.8), the ℓ^2 norm is found to be:

$$\|\Phi^m\|^2 = \begin{cases} 2\pi & m = 0 \\ \pi & \text{otherwise} \end{cases} \tag{3.56}$$

yielding the Zernike's angular orthonormalization constant c_{Φ^m} as:

$$c_{\Phi^m} = \begin{cases} \dfrac{1}{2\pi} & m = 0 \\ \dfrac{1}{\pi} & \text{otherwise} \end{cases} \tag{3.57}$$

Combined Multiplying eqs. (3.53) and (3.57) produces the combined orthonormalization constant $c_{Z_n^m}$ as:

$$c_{Z_n^m} = c_{R_n} c_{\Phi^m} = \begin{cases} \dfrac{n+1}{\pi} & m = 0 \\ \dfrac{2\,(n+1)}{\pi} & \text{otherwise} \end{cases} \tag{3.58}$$

The resulting orthonormalized Zernike polynomial relationship, implied from eqs. (3.16) and (3.58) is:

$$\breve{Z}_n^m(r,\phi) = \begin{cases} \dfrac{n+1}{\pi} Z_n^m(r,\phi) & m = 0 \\ \dfrac{2(n+1)}{\pi} Z_n^m(r,\phi) & \text{otherwise} \end{cases} \tag{3.59}$$

$$b_n^m = \begin{cases} \dfrac{n+1}{\pi} a_n^m & m = 0 \\ \dfrac{2(n+1)}{\pi} a_n^m & \text{otherwise} \end{cases} \tag{3.60}$$

3.3 Monte Carlo Processes

MC processes sample from unknown 'true' distributions F using complex stochastic techniques to produce representative distributions S. A sampled distribution S can be said to provide an estimate of the true distribution F:

$$S = \widehat{F} \tag{3.61}$$

Often the true distribution F is analytically intractable, i.e., an explicit solution is not readily available or possible due to complexities in geometries, materials, etc... of the model. This is often the primary motivation for using MC sampling techniques. Further, herein the notation is adopted that the distribution S can be sampled to produce a representative stochastic valuation such that:

$$S(p_\chi) = s_{p_\chi} w_{p_\chi} \tag{3.62}$$

where s_{p_χ} and w_{p_χ} are the stochastic value and weight, respectively, of particle p's reaction χ. Note that there is an inherent event location $\mathbf{x}_{p_\chi}$ associated with each sample.

3.3.1 Sample Distribution

The number of samples N corresponds to the confidence of the sampled distribution S; according to the law of large numbers:

$$S \to F \quad \text{when} \quad N \to \infty \tag{3.63}$$

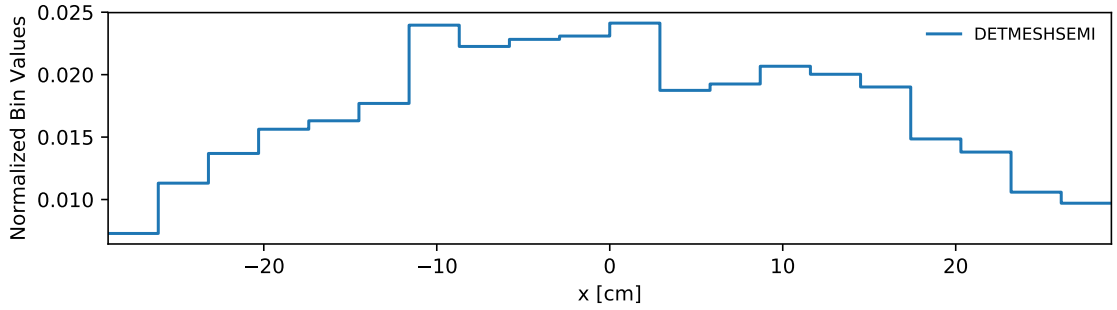
Small sample sets are statistically unlikely to produce a sampled distribution S even remotely similar to the true distribution F, especially if F has complex characteristics. Conversely, the sampled distribution S continues to better approximate the true distribution F with an increasing number of samples. Although it is impossible to make a general statement about how many samples N are 'enough,' many MC tools are equipped with metrics to help evaluate the quality of the sampled distribution S. Further scrutiny—even a qualitative analysis such as an inspection of plotted data—can provide information about if the sample population N is sufficient.

Consider fig. 3.6 as an example of a visual verification process. This demonstrates the evolution of a 20-bin histogram, the sampled distribution S, as a function of the sample population N. A rounded dome-type shape is expected based on prior knowledge of the model description; the true distribution F is unknown. At a mere 2×10^5 samples (fig. 3.6a), the sampled distribution has some obvious defects. Increasing the sample population N to 2×10^6 (fig. 3.6b) resolves some of the more glaring issues, but still exhibits anomalous features. Next, at 2×10^7 samples (fig. 3.6c), the distribution S looks comparable to what is expected. In many cases, this could be considered sufficient; however, the centroid of the distribution S is shifted slightly to the left. Applications requiring ultra-high fidelity representation would demand an even larger population N. Finally, a large 2×10^8 sample population N (fig. 3.6d) results in a near-perfect distribution; it is statistically unlikely that increasing the sample population N would add any more information to the sampled distribution S. Thus, inference based on visual information alone shows that the optimal sample population N likely lies between 2×10^7 and 2×10^8.

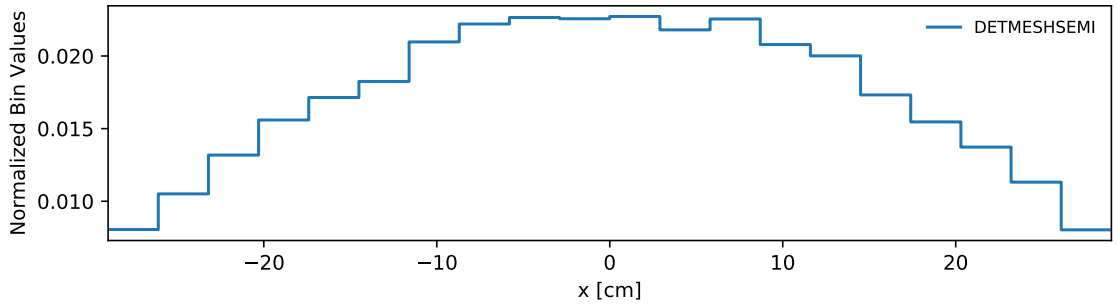
3.3.2 Tallies

MC processes typically use mesh tallies, such as those shown in fig. 3.6, to track properties of interest. In nuclear applications these include, but are not limited to: dose rates, fission heat generation, fission source terms, collision densities, flux rates, and angular distributions. The tracking is performed by segmenting the region of interest (ROI) into smaller tallying volumes, hence the term 'mesh'. The bins are usually equally-spaced throughout, although not necessarily so. A ROI can be defined in an arbitrary number of dimensions—it could be as a simple as a line in 1D, or more complex than a 3D volumetric phase space with additional dimensionality for time, energy, and angular dependencies.

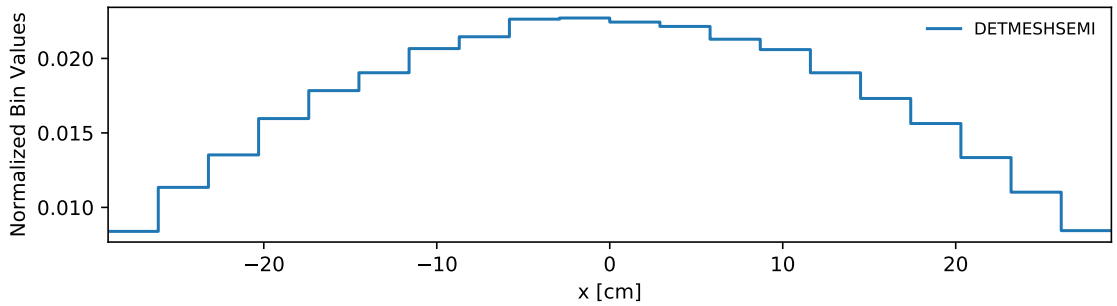
Recording a tally is conceptually simple. Each event is checked to see if it occurred within the ROI. If so, the corresponding bin value is incremented. Primitive tallies' algorithms may simply add '1' to the bin's value, while advanced tallies may compute the addend from geometrical-, material-, or event-based properties. For example, a collision-based tally for



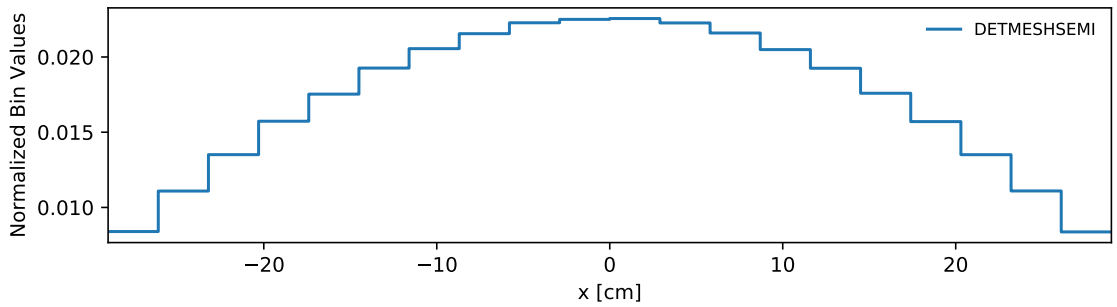
(a) 2×10^5 histories



(b) 2×10^6 histories



(c) 2×10^7 histories



(d) 2×10^8 histories

Figure 3.6 – Evolution of a sample distribution S as a function of the sample population N with a 20-bin histogram, where the true underlying distribution F is a concave function.

the oft-used flux distribution estimate of a bin b's value $\widehat{\phi}_b$ uses the form:

$$\widehat{\phi}_b = \frac{1}{NV_b\Sigma_{t,b}} \sum_{p=1}^{N} \sum_{\chi=1}^{p_{X,b}} S(p_\chi)$$

where N is the number of particles p simulated, V_b is the bin volume, $\Sigma_{t,b}$ is the total macroscopic cross section of the interaction material in bin b, $p_{X,b}$ is the number of events χ from particle p in bin b, and $S(p_\chi)$ is the event's stochastic valuation from eq. (3.62).

Intended as distribution representations themselves, functional expansions are highly suitable for stochastic sampling processes. As mentioned previously, the MC specialization of functional expansions is the FET, or functional expansion tally. FETs can be generated for any number of dimensions, even expanded beyond physical space to cover the phase space of other parameters. For example, an FET could be applied to track energy-dependent neutron flux through a 2D slab problem as easily as fission heat in a 3D cylinder.

The primary limitation is the availability of suitable function series. Both the ROI geometry and the underlying distribution's expected shape are factors for consideration when selecting a basis set ψ.

3.3.3 Implementation Concerns

The exact representations of most functional bases ψ are incredibly computationally cumbersome. Consider the naive definitions for the Legendre and Zernike polynomials: eqs. (3.18) and (3.36). Each presents a perfectly valid and mathematically true method for calculating polynomials, but should only be used when expenditure of time is not a concern or in research situations where a pedantic approach is required (see the opening maxim in section 1.1 for a statement on the availability of time with regards to M&S).

Algorithmic burden is a significant concern in MC applications; a smaller computational load improves sampling efficiency, meaning a increased number of samples produced per unit time. A larger sample population effects better statistics, which improves confidence in the results per the law of large numbers. Thus, care must exacted when selecting an algorithm for computing polynomial values. Faster algorithms are almost always optimal—as long as they maintain accuracy within the intended scope.

3.3.4 Monte Carlo-based Functional Expansion Integration

Founded on functional expansion theory, the generation of FETs for a sampled distribution S are rooted in eq. (3.14). Instead of an explicit solution, however, the integral is numerically sampled using MC techniques:

$$b_i = \lim_{N\to\infty} \frac{1}{N} \sum_{n=0}^{N} S(n)\breve{\psi}_i(\mathbf{x}_n)\rho_\psi(\mathbf{x}_n) \tag{3.64}$$

where $\mathbf{x}_p$ are the event coordinates for sample n mapped into the space of the functional basis set ψ. $S(n)$ is generalization of eq. (3.62); unbiased sampling is assumed, meaning each sample n is independent and uncorrelated to all other samples.

Unfortunately, resourcal constraints mean that an infinite number of samples is not feasible. Instead, the statistical estimator $\widehat{b}_i$ for a finite number of samples N is used:

$$\widehat{b}_i = \frac{1}{N} \sum_{n=0}^{N} S(n)\breve{\psi}_i(\mathbf{x}_n)\rho_\psi(\mathbf{x}_n) \tag{3.65}$$

3.4 Statistical Properties of Tallies

The basis for the statistic properties of FETs in MC contexts were established in a seminal work by Griesheimer in his PhD thesis *Functional Expansion Tallies for Monte Carlo Simulations*. Often the approximation order, i.e., number of mesh tally bins or the functional basis order, can be increased to improve the resolution and accuracy of the model. However, the statistical error and truncation error are inversely related. A higher order approximation means fewer events per bin or moment with a fixed sample population N. The obvious solution is to increase the sample population N, but finite computing resources and time often enforce an upper limit on the population size. Sure, a million-order mesh tally or FET has the potential to capture every conceivable characteristic of the true distribution F, but who has time to wait an eternity to collect enough samples for good statistics?

3.4.1 Mesh Tally Errors

Consider the mesh tally divided into B bins, with x_{b-1} and x_b as the boundaries and width Δx_b for bin b. The expected value $\widetilde{j}$, also the average value $\bar{j}$ over the interval $[x_{b-1}, x_b]$, is:

$$\widetilde{j} = \bar{j}_b = \frac{\int_{x_{b-1}}^{x_b} F(x)dx}{\Delta x_b} \tag{3.66}$$

3.4.1.1 Statistical Error

The estimate $\widehat{j}_b$ realizes the shape of the distribution $F(x)$ as events χ_b are collected within b:

$$\widehat{j}_b = \frac{\chi_b}{N} \tag{3.67}$$

The statistical error of the estimate $\widehat{j}_b$ is calculated as the variance $\sigma^2_{b,s}$ of the bin. One relation for the variance is:

$$\sigma^2_{b,s} = \left(\widetilde{j}_b - \widehat{j}_b\right)^2 \propto \frac{1}{\chi_b} \tag{3.68}$$

The statistical error vanishes as $N \to \infty$, which implies that $\chi_b \to \infty$, such that the estimator $\widehat{j}_b$ approaches to the expected value:

$$\lim_{N\to\infty} \widehat{j}_b \to \widetilde{j}_b \tag{3.69}$$

Finally, assuming negligible covariances, the total statistical error σ^2_s of the mesh tally can be calculated as:

$$\sigma^2_s = \sum_B \sigma^2_{b,s} \tag{3.70}$$

3.4.1.2 Truncation Error

The truncation error $\sigma^2_{b,t}$ is represented as the variance due to the discontinuous nature of the mesh tally. This is quantified at each bin b by taking a measure of the difference between the true distribution $F(x)$ and the expected value $\widetilde{j}_b$:

$$\sigma^2_{b,t} = \frac{\int_{x_{b-1}}^{x_b} \left(F(x) - \widetilde{j}_b\right)^2 dx}{\Delta x_b} \tag{3.71}$$

Note that as the bin width $\Delta x_b \to 0$ (which implies that $B \to \infty$), the instantaneous value $F(x) \to \widetilde{j}_b$, thus the truncation error also vanishes.

Again assuming no covariances between bins, the total truncation error σ_t can be written as:

$$\sigma_t^2 = \sum_B \sigma_{b,t}^2 \tag{3.72}$$

3.4.1.3 Combined Error

There is an obvious negative correlation between the statistical and truncation errors. With large bin widths each bin is able to collect a higher portion of samples; however, large bin widths drive up the truncation error. Conversely, the truncation error benefits from smaller bins, but to the degradation of the statistical error as fewer total samples are collected by the proportionally smaller bins.

The total error σ_T^2 for a mesh tally, combined from eqs. (3.70) and (3.72) (again simplified by neglecting any covariances), is:

$$\sigma_T^2 = \sigma_s^2 + \sigma_t^2 \tag{3.73}$$

A sample distribution S with negligible error—a high-fidelity estimate of F—occurs in the limits of infinitely many infinitesimal mesh tally bins and an infinite number of samples. Unfortunately the hardware to perform such a flawless calculation will probably not exist in the lifetime of this universe (again, see the opening maxim in section 1.1).

3.4.2 FET Errors

In his PhD thesis, Griesheimer [76] provided a quantitative derivation for the statistical error of an FET based on the mathematical underpinnings of FES themselves. Additionally, Griesheimer demonstrated that FETS have a much faster convergence rate than a similarly-ranked mesh tally (as shown in fig. 2.2). These proofs were followed-up with an actual demonstration comparing the convergence rates between the two types of tallies as a function of the number of samples.

3.4.2.1 Statistical Variance

First, a the statistical variance estimator $\widehat{\sigma}^2_{\widehat{b}_i}$ of the estimate $\widehat{b}_i$ was derived. Griesheimer's result, modified for the orthonormal approach presented in eq. (3.17), is:

$$\widehat{\sigma}^2_{\widehat{b}_i} = \frac{\sum_{p=1}^{N}\left(\sum_{\chi=1}^{p_X} S(p_\chi)\breve{\psi}_i(\mathbf{x}_{p_\chi,i})\rho_{\psi_i}(\mathbf{x}_{p_\chi,i})\right)^2 - \frac{1}{N}\left(\sum_{p=1}^{N}\sum_{\chi=1}^{p_X} S(p_\chi)\breve{\psi}_i(\mathbf{x}_{p_\chi,i})\rho_{\psi_i}(\mathbf{x}_{p_\chi,i})\right)^2}{N\,(N-1)} \tag{3.74}$$

Again, as with eq. (3.70) for the statistical error of mesh tallys, the FET statistical error decreases with an increasing number of samples N.

3.4.2.2 Truncation Error

As expected, all finite functional expansion are also subject to truncation error effects. Theoretically, any distribution could be represented by an infinite-order functional expansion. Conversely, the functional expansion is only an approximation if only a finite number of terms I are included. For any approximated distribution $\widehat{F}(x)$:

$$\widehat{F}(x) = \sum_{i=0}^{I} b_i \breve{\psi}_i(x) \tag{3.75}$$

where the approximation is represented as:

$$\widehat{F}(x) = F(x) + \epsilon_I(x) \tag{3.76}$$

and for the error of truncation $\epsilon_I(x)$ is:

$$\epsilon_I(x) = \sum_{i=I+1}^{\infty} b_i \breve{\psi}_i(x) \tag{3.77}$$

Griesheimer developed a methodology for determining an upper-limit on the error from a truncation approximation. Generically, it was demonstrated that setting a general form of the truncation error's upper bound is not possible because the infinity-norm convergence cannot be assumed for all orthogonal functional bases. However, a relation for the upper bound of the error in ℓ^2 space was possible:

$$\|\epsilon_I(x)\|^2 \leq \sum_{i=I+1}^{\infty} \frac{b_i^2}{c_i} \tag{3.78}$$

3.4.3 Advantages and Disadvantages

An infinite functional series is not just unobtainable but actually undesirable due to the statistical uncertainty of the MC process. Higher order functions create higher order terms in the functional expansion. Using the actual function definition of a distribution to generate the expansion coefficients, as shown in eq. (3.14), produces perfect reconstruction via eq. (3.15). However, in MC processes the expansion coefficients are randomly sampled from the distribution and are subject to statistical error. Equation (3.74) demonstrates that the variance of the expansion coefficient $\widehat{\sigma}^2_{\widehat{b}_i}$ is dependent on the values of the orthonormalized function $\breve{\psi}_i(\mathbf{x}_{p_\chi,i})$. As discussed in section 3.2.2, the extrema of the orthonormal functions $\breve{\psi}_i$ increase with higher orders. The net results is that the statistical error for an expansion coefficient $\widehat{\sigma}^2_{\widehat{a}_i}$ naturally increases with order. According to eq. (3.74), this can only be brought down with a larger sample population N.

Thus, the use of truncated functional sets in FETs is a blessing with caveats. The regrettable aspect is that the true distribution F is only approximated. The benefit is that, by using a carefully selected truncated function set, the sampled distribution S can be a *very close* approximation with only a finite number of samples N. Griesheimer demonstrated—both theoretically and numerically—that the approximation of FETs are superior to their conventional mesh tally counterparts. In addition, FETs are continuous, a benefit that mesh tallys simply cannot provide except toward the limit of an infinitely-many number of bins.

3.5 Sibling Principles in Image Analysis

An analogue to the functional expansion technique is the 2D analysis of images by moments [5]. In this shared context, moments are a scalar quantity used to characterize the similarity in features of an object w.r.t. a basis function; mathematically this is the projection of that object onto the corresponding function. The zeroth-order moment is typically called the 'piston', and provides the average value of the image. The first-order moments provide information about the image's centroid. Additional moments continue to add details as the order approaches infinity.

Equivalently in functional expansions, the zeroth-order coefficient provides the average value of the distribution over the domain. Comparing the remaining process to eq. (3.3), image moments and functional expansions are different contextual approaches based on similar principles.

Both techniques are capable of working in multiple dimensions, although image analysis in 1D is meaningless because the medium itself is not presented in anything less that two dimensions. The 2D analysis of images are typically performed using a normalized representation of images, such as pixel luminosity or hue. Additional information, such as red-green-blue (RGB) channels, are not considered as dimensions of the representation. Instead, the channels are considered a subset of the 2D image, and each pixel is represented with a vector instead of a single scalar value.

These moments are used primarily for analysis of the images. One aspect in particular is in pattern recognition and object matching using only the moments, a notoriously difficult challenge for artificial intelligence. Moment-based analysis of images seeks to create a representation of an image that is easily comparable to other moment sets, whether directly or through transformation (reflection, rotation, scaling, translation) and deformation (ballooning, pinching, projection distortion, shearing). These approaches are also tolerant to information loss including blurring, poor contrast/illumination, (de)saturation, noise, and occlusion. Simplistically, analyses are performed by mathematically transforming or deforming an image's set of moments, then comparing to another image's moment set.. The image moments data are also more compact than the original image, creating a 'thumbprint' characteristic of the image. This thumbprint has minimal storage requirements, can be used as an indexer for the full image, and enables quick and efficient comparisons among a large set of images via the other thumbprints.[3]

[3]Note that image moment sets are not meant—nor designed—to store actually store images. To do so would be the digital equivalent of reading a plot synopsis for *Les Miserables*, then presume a deep appreciation for Hugo's detailed account for the Battle of Waterloo. Image moments are meant to provide only a high-level tool for analysis and comparison.

Theoretically, higher-order moments *could* be employed until the original image is recreated faithfully. In reality, this approach is very inefficient—both computationally and storage-wise—compared to most standard image formats that are specifically designed for storing images. Common examples include the lossy JPEG or lossless PNG formats. Nevertheless, reconstructions based on Chebyschev moments have been demonstrated to duplicate the original image flawlessly [5, p. 374-377].

3D image analysis cases are historically less studied, but have attracted interest in the past two decades with the emergence of numerous imaging technologies. The most prominent area is the medical field, where 3D images are generated by technologies such as magnetic resonance imaging or x-ray computed tomography. Surface mapping technologies are more common; even household items like the Kinect[4] can be used to generate 2.5D images.[5]

Notwithstanding, the same function series used for image moments can be used, or easily converted for use, with functional expansions methods. The primary difference between the two approaches is in the usage intent. In image analyses the moments are calculated once for an image, thus the methodology is not overly concerned with finding the lowest polynomial order that can accurately represent an image in order to save on repetitive evaluations. These moments are not reproductions of the original image,[6] but intended to provide a reduced-order mathematically robust thumbprint. Functional expansions, on the other hand, seek to preserve or recreate the shape of the represented distribution F as closely as possible. Fets are designed to replace or augment the established storage mechanism, mesh tallies, in mc applications. It is also important to note that images are defined by edges, or discontinuities in the distribution. Conversely, functional expansions are most successful when they are applied to a domain in which the represented distribution is smooth. A model known or expected to contain quickly changing features is often subdivided at the anticipated discontinuities and an independent fet assigned to each.

[4]The Kinect sensor is a motion controller for the Xbox gamins system, but has been adapted by researchers and hobbyists alike for a broader range of purposes [103].

[5]The classification 2.5D is used because surface mapping provides depth, i.e., a 3D description of the object's exterior, but cannot provide information about features within of the object's volume.

[6]Nevertheless, moments *could* be used to reproduce an image (see footnote 3).

Part II

Methodology

Chapter 4

Functional Expansion Algorithm Optimization

> *If you find that you're spending almost all your time on theory, start turning some attention to practical things; it will improve your theories. If you find that you're spending almost all your time on practice, start turning some attention to theoretical things; it will improve your practice.*
>
> — Donald Knuth

One pillar of the opening maxim is that time is a limited resource. This is also true in M&S environments, where and optimal balance is sought between computation time and simulation fidelity.

The current availability of petascale computing, and the imminent advent of exascale resources [105], will only dull the disadvantages of inefficient algorithms. Supercomputing resources are not free, either, and these inefficiencies are reborn monetarily as operational electricity costs. Thus, it is important to produce efficient and well-designed code.

Functional expansions are not isolated from efficiency issues. A few optimizations were discovered and implemented that significantly improved the computational efficiency of FE calculations.

This chapter is adapted from Wendt *et al.* [104].

4.1 Direct Formulae vs. Recurrence Relations

In many cases, the direct calculation of a polynomial term is faster than any other algorithmic form. By direct calculation, it is meant the evaluation through a hard-coded formula such as:

$$P_4 = \frac{1}{8}(35x^4 - 30x^2 + 3) \tag{4.1}$$

for the fourth Legendre polynomial. Unfortunately, this approach limits the maximum functional order available to the user. It can also become incredibly cumbersome to code and maintain multidimensional functional bases; a full 18$^{\text{th}}$-order Zernike polynomial definition can easily span over 1000 lines of code. A single erroneous algebra operation can quickly get lost in the swamp of formula after formula.[1]

Alternatively, as described in section 3.2, most functional bases used for FES will inherently have one or more recurrence relation forms. The primary benefits of recurrence relations are two-fold. First, toward higher-order terms they become increasingly efficient when compared to a function's corresponding naive algorithm. Second, they provide a single-fail point that can be checked and verified; a corrected bug in the algorithm will instantly correct the computation of all terms throughout the functional series.

The performance of the recurrence relation is superseded by the pure performance of a direct calculation formula for lower-order functional terms. However, there is a trade-off order at which computing a term using the recurrence relation becomes more efficient than the direct calculation. Using pure FLOPS as the measurement metric, each direct calculation will have a larger cost than the previous order; conversely, the recurrence relation cost is constant for each order. For a hypothetical orthogonal basis (ignoring the overhead costs for looping and memory operations), the first two terms are typically just assignment operations as shown in eqs. (3.20) and (3.21). Beyond these initializations, the i^{th} term of an optimized direct formula will require about $i + 1$ FLOPS; conversely, based on eq. (3.4), the recurrence relation will require around 9 FLOPS per order up through i.

[1] Direct calculation formulae were already used in the OpenMC FET implementation [106]. Legendre polynomials were implemented through order 10 and Zernike polynomials through order 18.

If just a single term is to be calculated then the direct formula has an obvious advantage. However, the situation becomes more complex if all terms are required. Using the symbol $\mathfrak{C}$ to represent the total computational cost of an algorithm measured in FLOPS, for direct calculations the relation is:

$$\mathfrak{C}_d(i) \approx \frac{i^2 + 3i}{2} \tag{4.2}$$

while the recurrence relation requires:

$$\mathfrak{C}_r(i) \approx 9i - 1 \tag{4.3}$$

Equating the two and solving shows an inflection at $i = 14$. As expected, for lower-order terms the pure direct formula is more efficient, while the pure recurrence relation is optimal when moving to higher-order terms.

To emphasize the point, consider the specific case of the Legendre polynomials. The symbol $\mathfrak{c}$ will be used to indicate the individual contributing cost of the i^{th} order calculation. When evaluating one order at a time, the direct Legendre polynomial calculations have a FLOP cost equal to the order value:

$$\mathfrak{c}_d(i) = \begin{cases} 1 & i = 1, 2 \\ i & \text{otherwise} \end{cases} \tag{4.4}$$

$$\mathfrak{C}_d(i > 1) = \frac{i^2 + i + 2}{2} \tag{4.5}$$

while the recurrence relation is:

$$\mathfrak{c}_r(i) = \begin{cases} 1 & i = 1, 2 \\ 10 & \text{otherwise} \end{cases} \tag{4.6}$$

$$\mathfrak{C}_r(n > 1) = 10i - 8 \tag{4.7}$$

Equating eqs. (4.5) and (4.7) shows an inflection point at order 18.

4.2 Hybrid Algorithms

A hybrid calculation approach can be leveraged to extract the maximum available performance from both methods. The direct calculation can be used for lower-order evaluations,

Table 4.1 – Demonstration of the number of iterations required to compute all estimators $\widehat{b}_i$ as a function of domain dimensionality and the order of the basis set for each dimension.

	Functional Order per Dimension			
Dimensions	**1**	**3**	**7**	**15**
1D	2	4	8	16
2D	4	16	64	256
3D	8	64	512	4096
4D (3D + energy)	16	256	4096	65 536
5D (3D + energy + time)	32	1024	32 768	1 048 576

Note that the presence of a zeroth order term means that the number of iterations—and thus expansion coefficients—is one greater than the order, an assumption is based on using a generic 1D functional series as the basis set for each dimension. Additionally, the example domains for the 4D and 5D cases are purely illustrative, not a fixed requirement. Any domain could conceivably be used to match the problem dimensionality. A standard 2D domain in Euclidean space could even be expanded to a higher phase space by the addition of a non-spatial domain such as time, energy, or angular dependence.

and the recurrence relations for all others. This approach, consequently, removes any fixed ceiling for accessible functional orders if only a direct calculation approach were employed.

Considering the case of the Legendre polynomials again, the direct calculation would be used as long as eq. (4.4) is less than eq. (4.6); otherwise, the recurrence relation would be used. The means that an optimal swap-over would occur at order 10; the direct calculations should be used for $i \leq 10$, after which orders the recurrence relation should take over by using the direct calculation values for $i = 9, 10$ to seed the algorithm. It should be noted that, in practice, the recurrence relation often incurs some additional overhead cost from condition checks, looping, and indexing increments. In effect, the optimal swap-over point is actually slightly higher.

4.3 Vector Approach

The sampling of an estimator $\widehat{b}_i$ for the expansion coefficient b_i is often a multi-dimensional problem. The estimates $\widehat{b}$ have exactly the same dimensionality as the functional basis set ψ. Thus, the entire dimension space must be iterated over for each event in order to calculate the contribution to each estimator $\widehat{b}_i$. The problem becomes exponentially more intensive with the addition of dimensions or orders, as shown in table 4.1.

FE Orders

Squareness in the number of functional terms per dimension is not required, but just used here for simplicity. Realistically, a 3D Cartesian functional basis set could contain a different number of terms in each of x, y, and z. For example, a 3D FE with corresponding orders of 7, 4, and 13, respectively, could contain 364 expansion coefficients. The functional order in any dimension can be tuned to best represent the projection of the true distribution F in that dimension. A smooth or otherwise featureless projection will require a simple low-order functional basis set, while a complex or unknown projection may use a higher-order basis to capture the desired features.

Up to this point all known FE/FET applications have performed functional evaluations in a sequential manner. This means that each term is completely evaluated independently of all other term evaluations in the series set. Of necessity, recurrence relationships use all terms in a series up to the requested order I. This means that a request for the I^{th} term will also compute all terms i in $0 \leq i < I$. Thus, in a sequential FE evaluation approach, the i^{th}-term will be calculated $I - i + 1$ times. All but the final evaluation is discarded in the process, effectively wasting a significant amount of computing resources.

When using a recurrence relation, only the highest-order term I needs to be requested. All terms i can be collected into an array as they are calculated in consequence of recursing up to the highest order term I. An appropriately enabled calling routine will be able to efficiently use this set of vector data, rather than sequentially requesting each term one at a time. This approach is termed herein as 'vector-based' but is also commonly known as memoization.

Additionally, the hybrid optimization can be used with a vector-based approach. In fact, the utilization of a vector array greatly simplifies the implementation of a hybrid approach. This is most easily implemented as a `switch-case` statement coupled with a recurrence relation. The `switch-case` statement should be configured to start at the highest possible direct calculation order and fall through to any lower-order terms. If all requested terms

are coded in as direct calculations then this enables a single-shot collection of all terms without any additional overhead processing. Otherwise, the recurrence relation continues the evaluations beyond the maximum term in the `switch-case` statement by using the two highest direct calculations as the previous terms.

This approach is demonstrated by the following C code chunk for the Legendre polynomials, based on a highest-implemented direct calculation order of 8:

```
#define MAX_ORDER 8
void HybridVectorLegendre(const unsigned int order, const double x, double *const vector)
{
  unsigned int i;
  const double x2 = x * x; /* Saves FLOPS when "order > 2", a very common situation. */

  switch (order)
  {
  default: /* Fill in "vector" with direct calculations when "order > MAX_ORDER" */
  case MAX_ORDER: /* MAX_ORDER = 8 */
    vector[8] = ((((6435 * x2 - 12012) * x2 + 6930) * x2 - 1260) * x2 + 35) / 128;
    /* No "break;" means all higher orders automatically drop thru to lower orders */
  case 7:
    vector[7] = (((429 * x2 - 693) * x2 + 315) * x2 - 35) * x / 16;

  .
  .
  .

  case 2:
    vector[2] = (3 * x2 - 1) / 2;

  case 1:
    vector[1] = x;

  case 0:
    vector[0] = 1;
  }

  for (i = MAX_ORDER + 1; i <= order; ++i) /* Only evaluate when "order > MAX_ORDER" */
    vector[i] = ((i + i - 1) * x * vector[i - 1] - (i - 1) * vector[i - 2]) / i;
}
```

Complete evaluation of all requested terms is enabled by the numerically reverse `case` ordering, the `default` statement on line 9, and a conspicuous lack of inter-`case` `break` statements. All requested/available direct calculations are collected from the top down using a single conditional check. A precomputation of x^2 on line 5 allows for additional FLOP reductions throughout the direct calculations. The recurrence relation `for`-loop, on lines 28 to 29, only evaluates if the requested order is higher than what was provided by the direct computations. Further, the first two iterations (when `i` equals 9 or 10) depend on the 7^{th}- and 8^{th}-order vector values computed via direct calculation. Finally, it is assumed that the vector array `vector` is allocated outside of the routine, is appropriately sized to `order` + 1, and is passed in as a valid pointer.

4.3.1 Convolution Methods

A critical component of computing higher-order FE coefficients is the convolution algorithm used to generate the cross-terms and form a multivariate 3D functional series. The options are dependent on the term evaluation methods presented previously. This is especially true of the vector-based approach, which requires a carefully-crafted convolution method to maximize the potential efficiency.

The progressive derivation of the vector-enabled convolution method will start with the naive sequential algorithm alluded to previously. A three-series convolution will be demonstrated, which can be adapted to fewer/more series by removing/adding nested loop levels. The focus will be on generating the expansion coefficients, but an adaptation will be demonstrated afterward for reconstructing an expansion.

4.3.1.1 Basic Looping

The basic looping is a naive implementation. Nested loops iterate over the orders in all three series, and the functional evaluations are performed within the deepest loop. This approach is captured by algorithm 1. All together, this algorithm requires $3IJK$ term evaluations. The computational cost $\mathfrak{C}$ of this algorithm increases exponentially with higher-order bases.

Algorithm 1 Convolution algorithm applying deeply-nested loop calculations to generate coefficients for a 3D FE using three 1D functional series, evaluated out to the maximum term $\{I, J, K\}$, agaisnt a generic value s.

```
procedure BasicMakeFE(x, y, z, s, Λ)
    for all i in I do
        for all j in J do
            for all k in K do
                Λ[i, j, k] ← Λ[i, j, k] + s * Series1(i, x) * Series2(j, y) * Series3(k, z)
            end for
        end for
    end for
    return Λ
end procedure
```

4.3.1.2 Modified Looping

It was observed that the functional evaluations could be moved out to the loop corresponding to their dimensionality. This is because the corresponding order of each term is not modified by deeper-nested loops. This approach is outlined in algorithm 2.

Algorithm 2 Convolution algorithm leveraging split-nested calculations to generate coefficients for a 3D FE using three 1D functional series, evaluated out to the maximum term $\{I, J, K\}$, against a generic value s.

```
procedure ModifiedMakeFE(x, y, z, s, Λ)
    for all i in I do
        a ← Series1(i, x)
        for all j in J do
            b ← Series2(j, y)
            for all k in K do
                c ← Series3(k, z)
                Λ[i, j, k] ← Λ[i, j, k] + s * a * b * c
            end for
        end for
    end for
    return Λ
end procedure
```

4.3.1.3 Precomputation Looping

The next step is enabled by the realization that x, y, and z are constant throughout the scoring algorithm. Even though it may add more lines of code, the number of function evaluations

can be further reduced by pre-calculating all the values prior to the primary loop. Evaluation still requires the fully nested `for`-loops to iterate over the entire phase space. However, each term evaluation is acquired from the pre-computed vectors. This approach is outlined in algorithm 3. This algorithm requires $I\,(1 + J\,(1 + K))$ term evaluations. Compared to algorithm 1 this is a savings of $I\,(J\,(2K - 1) - 1)$.

Algorithm 3 Convolution algorithm leveraging split-nested calculations to generate coefficients for a 3D FE using three 1D functional series, precomputed out to the maximum term $\{I, J, K\}$ into corresponding vectors $\{\mathbb{I}, \mathbb{J}, \mathbb{K}\}$, against a generic value s.

procedure ModifiedMakeFE(x, y, z, s, Λ)
  for all i **in** I **do**
    $\mathbb{I}[i] \leftarrow$ Series1(i, x)
  end for
  for all i **in** I **do**
    $\mathbb{J}[j] \leftarrow$ Series2(j, y)
  end for
  for all i **in** I **do**
    $\mathbb{K}[j] \leftarrow$ Series3(k, z)
  end for
  for all i **in** I **do**
    for all j **in** J **do**
      for all k **in** K **do**
        $\Lambda[i, j, k] \leftarrow \Lambda[i, j, k] + s * \mathbb{I}[i] * \mathbb{J}[j] * \mathbb{K}[k]$
      end for
    end for
  end for
  return Λ
end procedure

4.3.1.4 Vector Looping

As mentioned before, the recurrence relations intrinsically evaluate all preceding terms. These can be accumulated in one go rather than evaluating each term individually as in algorithm 3. The resulting algorithm is demonstrated in algorithm 4. The overall result reduces the number of term evaluations to the minimum possible: $I + J + K$ per each algorithm execution. The cost savings will be especially noticeable with higher order polynomial sets.

Algorithm 4 Convolution algorithm to generate coefficients for a 3D FE that precalculates the term evaluations of three 1D functional series, evaluated out to the maximum term $\{I, J, K\}$ into corresponding vectors $\{\mathbb{I}, \mathbb{J}, \mathbb{K}\}$, against a generic value s.

```
procedure VectorMakeFE(x, y, z, s, Λ)
    𝕀 ← VectorSeries1(I, x)
    𝕁 ← VectorSeries2(J, y)
    𝕂 ← VectorSeries3(K, z)
    for all i in I do
        for all j in J do
            for all k in K do
                Λ[i, j, k] ← Λ[i, j, k] + s * 𝕀[i] * 𝕁[j] * 𝕂[k]
            end for
        end for
    end for
    return Λ
end procedure
```

4.3.1.5 Vector Looping 2

A modification to algorithm 4 can reduce the number of FLOPS. This is presented in algorithm 5. The improvement requires two extra temporary variables in memory, an insignificant cost on modern computing systems, for a reduction of $I\,(J\,(2K-1)-1)$ FLOPS per algorithm execution.

Algorithm 5 A variation on algorithm 4 to reduce the number of FLOPS.

```
procedure VectorMakeFE(x, y, z, s, Λ)
    𝕀 ← VectorSeries1(I, x)
    𝕁 ← VectorSeries2(J, y)
    𝕂 ← VectorSeries3(K, z)
    for all i in I do
        a ← s * 𝕀[i]
        for all j in J do
            b ← a * 𝕁[j]
            for all k in K do
                Λ[i, j, k] ← Λ[i, j, k] + b * 𝕂[k]
            end for
        end for
    end for
    return Λ
end procedure
```

4.3.1.6 Vector-based Expansion

A promised, algorithm 5 can also be repurposed to expanding an FE about a point. This is shown in algorithm 6, which could even use the same set of coefficients Λ calculated in algorithm 5.

Algorithm 6 A reapplication of algorithm 5 to expand an FE at a point using a set of previously-generated coefficients Λ.

```
procedure VectorExpandFE(x, y, z, Λ)
    s ← 0.0
    𝕀 ← VectorSeries1(I, x)
    𝕁 ← VectorSeries2(J, y)
    𝕂 ← VectorSeries3(K, z)
    for all i in I do
        a ← 𝕀[i]
        for all j in J do
            b ← a * 𝕁[j]
            for all k in K do
                s ← s + b * 𝕂[k] * Λ[i, j, k]
            end for
        end for
    end for
    return s
end procedure
```

Again, this works as long as the appropriate functional forms are used. For example, if the coefficient set Λ was generated from an orthonormalized function series then algorithm 6 would use the corresponding standardized function forms. Conversely, if the coefficients Λ were generated from standardized functions then either: 1) algorithm 6 would need to utilize orthonormalized functions, or 2) an intermediate preprocessing step to algorithm 6 would be required to multiply the appropriate orthonormalization coefficients c into the coefficient set Λ.

4.3.1.7 Resource Implications

Finally, two implications of the vector-based algorithms for computing resources must be mentioned for completeness: looping and memory.

Looping The nested loops themselves are still required, as the evaluation process must iterate through the entire space of the functional expansion basis set and calculate the contribution to each coefficient. What has been done via the vectorization process is to move as much of the computation as possible outside of the loop code. Thereby, the performance of the iterative environment is improved by eliminating unnecessary repetitive calculations. The number of iterations still increases exponentially as shown in table 4.1, but externalizing as many operations as possible to outside the loops significantly decreases the *rate* of growth.

Memory The full functional evaluation of algorithm 3, and subsequently algorithms 4 and 5, have the largest memory requirement. This is an inevitable effect of individually storing each recurrence relation step in a vector array. However, the incremental demand in memory resources is essentially negligible on modern computing systems. Any exponential increase in memory consumption—due to either higher-order terms or more dimensions—is comparatively dwarfed by the corresponding and significantly larger increase in the processing power demands. For example: given a 3D functional basis set of order 16 in each dimension, the vectors $\mathbb{I}$, $\mathbb{J}$, and $\mathbb{K}$ will require only 408 bytes on a modern 64-bit system.

The primary concern is *how* the storage arrays are created. Overhead for dynamic memory allocation can be considerable if performed repetitively, i.e., within each call to the algorithm. Instead, pre-allocating the arrays before beginning any FE evaluations is highly recommended.

4.3.2 Orthonormal Adaptations

It is paramount to recognize that the end-use of the convolved series will determine whether standardized or orthonormalized functional series are used. For example, a reconstruction routine following eq. (3.15) will use the standardized functional series forms in the convolution. Conversely, a coefficient-generation routing following eq. (3.14) or eq. (3.65) will use the orthonormalized forms so that the coefficients c are already incorporated into the evaluations.

Both direct and recurrence relations expressions need to be converted to orthonormalized forms. The direct calculations are very straighforward, as the c_i are known for each term

and can be simply inserted during coding. For example, generating the orthonormalized form of eq. (4.1), the fourth-order Legendre polynomial, can be performed by using eq. (3.27) to calculate c_{P_4} and multiplying:

$$\begin{aligned} c_{P_4} &= \frac{9}{2} \\ \breve{P}_4 &= \frac{73}{8}(35x^4 - 30x^2 + 3) \end{aligned} \tag{4.8}$$

The recurrence relations are more challenging to implement efficiently. The solution is to algebraically insert the orthonormalization constant equation into the recurrence relation itself and simplify. A full orthonormalization of the recurrence relation also enables explicit linking to the orthonormalized direct calculations without any intermediate steps. Further, this preserves algorithmic resiliency against unwanted round-off error in the floating-point operations; the forms of recurrence relations are often configured to minimize error contributions from round-off error [99]. In one case a *low-order* orthonormalized Zernike polynomial recurrence relation actually required *fewer* operations than the corresponding direct calculation.

4.3.2.1 Legendre Polynomials

The direct calculation forms of the first 12 orthonormalized Legendre polynomials are collected in table A.1.

The vector-based process is designed to calculate each required value only once, but eq. (3.22) does not generate orthonormal terms. Thus, it is necessary to develop a modified recurrence relation form suitable for use with algorithm 5. This optimized vector-based version of the recurrence relation must be able to accept the two previous orthonormalized terms, $\breve{P}_{l-1}$ and $\breve{P}_{l-2}$, then return the next orthonormal term $\breve{P}_l$ in the sequence.

Equation (3.26) is inverted to produce:

$$P_l(x) = \frac{2}{2l+1}\breve{P}_l(x) \tag{4.9}$$

This result is substituted in while algebraically manipulating eqs. (3.22) and (3.26):

$$\begin{aligned}
\breve{P}_l(x) &= \frac{2l+1}{2} P_l(x) \\
&= \frac{2l+1}{2} \frac{(2l-1)\, x P_{l-1}(x) - (l-1)\, P_{l-2}(x)}{l} \\
&= \frac{2l+1}{2} \frac{(2l-1)\, x \left(\frac{2}{2(l-1)+1} \breve{P}_{l-1}(x)\right) - (l-1) \left(\frac{2}{2(l-2)+1} \breve{P}_{l-2}(x)\right)}{l} \\
&= \frac{2l+1}{2} \frac{(2l-1)\, x \frac{2}{2l-1} \breve{P}_{l-1}(x) - (l-1)\, \frac{2}{2l-3} \breve{P}_{l-2}(x)}{l} \\
&= \frac{2l+1}{2l} \left((2l-1)\, x \frac{2}{2l-1} \breve{P}_{l-1}(x) - (l-1)\, \frac{2}{2l-3} \breve{P}_{l-2}(x)\right) \\
\breve{P}_l(x) &= \frac{2l+1}{l} \left(x \breve{P}_{l-1}(x) - \frac{l-1}{2l-3} \breve{P}_{l-2}(x)\right)
\end{aligned} \tag{4.10}$$

Care must be taken with each substitution of eq. (4.9) to ensure that the proper polynomial order is used, especially as shown in step 3.

4.3.2.2 Zernike Polynomials

The direct calculation forms of the first 10 orthonormalized Zernike polynomials are collected in table A.2.

Similar to the orthonormalization of the Legendre recurrence relation in eq. (4.10), eqs. (3.42) and (3.43) need alteration to produce an orthonormal result in a recursive manner. For convenience, both the radial and angular orthonormalization constants were incorporated in the radial component:

$$\breve{R}_n^m(r) = \begin{cases} \dfrac{n+1}{\pi} R_n(r,\phi) & m = 0 \\ \dfrac{2(n+1)}{\pi} R_n(r,\phi) & \text{otherwise} \end{cases} \tag{4.11}$$

while the angular component remains as shown in eq. (3.35) such that:

$$\breve{Z}_n^m(r,\phi) = \breve{R}_n^m(r) \Phi^m(\phi) \tag{4.12}$$

As stated in sections 3.2.4.2 and 3.2.4.3, Chong's method was selected as the preferred implementation. In summary, 1) the performance was on-par with, or better than, the other best-available methods, 2) it was formulated to calculate by traversing the 2D rank-order set

by orders, 3) the recurrence relations in eq. (4.11) are also a function of the order, and 4) the best-available indexing scheme is easily adapted to order-based relations.

All three stages of the recurrence relations—eqs. (3.37), (3.42) and (3.43)—will be considered twice, once for each of the cases in eq. (4.11). Explicit definitions as pure functions of the radius r can be found in some cases through simplification and substitution of previously established relations.

First Stage The first stage is used when the rank and order values are identical. The standardized recurrence relation for the first stage is eq. (3.37).

Zero Rank: ($m = 0 \therefore p = 0$) This is always calculated directly (see table A.2), so a recurrence relation is not needed. However, for completeness:

$$\begin{aligned}
\breve{R}^p_p(r) &= \frac{(p+1)}{\pi} R^p_p(r) \\
\breve{R}^0_0(r) &= \frac{1}{\pi} R^0_0(r) \\
&= \frac{r^0}{\pi} \\
\breve{R}^0_0(r) &= \frac{1}{\pi}
\end{aligned} \tag{4.13}$$

Other Ranks: ($m \neq 0 \therefore p > 0$)

$$\begin{aligned}
\breve{R}^p_p(r) &= \frac{2(p+1)}{\pi} R^p_p(r) \\
\breve{R}^p_p(r) &= \frac{2(p+1)}{\pi} r^p
\end{aligned} \tag{4.14}$$

Second Stage The second stage is used when the rank value is two less than the order value. The standardized recurrence relation for the second stage is eq. (3.42).

Zero Rank: ($m = 0 \therefore p = 2$) This can be calculated directly (see table A.2), so a recurrence relation is not always needed for this term.

$$\begin{aligned}
\breve{R}_p^{p-2}(r) &= \frac{(p+1)}{\pi}\left(pR_p^p(r) - (p-1)R_{p-2}^{p-2}(r)\right) \\
\breve{R}_2^0(r) &= \frac{3}{\pi}\left(2R_2^2(r) - R_0^0(r)\right) \\
\breve{R}_2^0(r) &= \frac{3}{\pi}\left(2r^2 - 1\right)
\end{aligned} \tag{4.15}$$

Other Ranks: ($m \neq 0 \therefore p > 2$)

$$\begin{aligned}
\breve{R}_p^{p-2}(r) &= \frac{2(p+1)}{\pi}\left(pR_p^p(r) - (p-1)R_{p-2}^{p-2}(r)\right) \\
&= \frac{2(p+1)}{\pi}\left(p\frac{\pi}{2(p+1)}\breve{R}_p^p - (p-1)\frac{\pi}{2(p-1)}\breve{R}_{p-2}^{p-2}\right) \\
&= (p+1)\left(\frac{p}{p+1}\breve{R}_p^p - \breve{R}_{p-2}^{p-2}\right) \\
\breve{R}_p^{p-2}(r) &= p\breve{R}_p^p - (p+1)\breve{R}_{p-2}^{p-2}
\end{aligned} \tag{4.16}$$

Conceptually, eq. (4.16) could be reduced to a direct calculation by substituting in the result of eq. (4.14). However, it was discovered that this reduction actually introduced more complexity. Consequently, the recurrence relation is retained here.

Remaining Stages The remaining stages are used whenever the first and second stages do not apply, i.e., $p - q \geq 4$. The standardized recurrence relation for all other stages is eq. (3.43).

Zero Rank: ($m = 0 \therefore q = 4$)

$$\begin{aligned}
\breve{R}_p^{q-4}(r) &= \frac{(p+1)}{\pi}\left(H_1R_p^q(r) + \left(H_2 + \frac{H_3}{r^2}\right)R_p^{q-2}(r)\right) \\
\breve{R}_p^0(r) &= \frac{(p+1)}{\pi}\left(H_1R_p^4(r) + \left(H_2 + \frac{H_3}{r^2}\right)R_p^2(r)\right) \\
&= \frac{(p+1)}{\pi}\left(H_1\frac{\pi}{2(p+1)}\breve{R}_p^4(r) + \left(H_2 + \frac{H_3}{r^2}\right)\frac{\pi}{2(p+1)}\breve{R}_p^2(r)\right) \\
\breve{R}_p^0(r) &= \frac{1}{2}\left(H_1\breve{R}_p^4(r) + \left(H_2 + \frac{H_3}{r^2}\right)\breve{R}_p^2(r)\right)
\end{aligned} \tag{4.17}$$

Other Ranks: ($m \neq 0 \therefore q > 4$)

$$\begin{aligned}
\breve{R}_p^{q-4}(r) &= \frac{2(p+1)}{\pi}\left(H_1 R_p^q(r) + \left(H_2 + \frac{H_3}{r^2}\right) R_p^{q-2}(r)\right) \\
&= \frac{2(p+1)}{\pi}\left(H_1 \frac{\pi}{2(p+1)} \breve{R}_p^q(r) + \left(H_2 + \frac{H_3}{r^2}\right) \frac{\pi}{2(p+1)} \breve{R}_p^{q-2}(r)\right) \\
\breve{R}_p^{q-4}(r) &= H_1 \breve{R}_p^q(r) + \left(H_2 + \frac{H_3}{r^2}\right) \breve{R}_p^{q-2}(r) \qquad (4.18)
\end{aligned}$$

Because of the use of Chong's method, the final orthonormal recurrence relations are actually remarkably similar to their parent standardized relations. This is due to the order-based approach of Chong's method. Equation (4.15) is just eq. (3.43) scaled by a factor of one-half. Further, the orthonormalization when $m \neq 0$ is a function of order n only, thus identical for all same-order terms in Chong's relations. As a result, eq. (4.16) is operationally identical to eq. (3.43). Thus the implementation in code required only minor modification when adapting already-implemented standardized algorithms to generate orthonormalized results.

4.4 Benchmarking

The benchmarking was performed on an 8-core processor. The host operating system was an Arch Linux base customized for development and testing. Core frequencies were locked at 3.7 GHz, and the turbo features were disabled, to ensure consistency throughout the testing.[2] The actual core frequencies were monitored throughout the testing. No deviation greater than 0.49 MHz (0.013 %) was observed, and such deviations were uncommon.

All benchmarking binaries were compiled with the flag `-O3` for aggressive optimization. Link-time optimization was also enabled.

4.4.1 Standalone Performance Benchmarking

An FE calculation methodology can be realized as the combination of two component algorithms. The first component, termed evaluation in this section, is the algorithm used

[2]The maximum core frequency was 4.2 GHz. Using a lower value ensured that the system cooling was sufficient to prevent hardware-imposed limitations from setting in.

to calculate the functional terms in the FE basis set. This component encompasses the polynomial recurrence relations and direct calculation methods. The second component, hereafter termed convolution, is the algorithm used to combine the individual function terms with the to create the FE coefficients. Convolution comprises the looping algorithms described in section 4.3.1.

Both single-function performances and a cylindrical convolution were benchmarked. This generated performance data for the 1D Legendre polynomials, 2D Zernike polynomials, and a 3D convolved cylindrical functional basis. All the benchmarks were run once each with the four established methodologies: 1) original, 2) hybrid, 3) standard vector, and 4) orthonormal vector.

The 1D Legendre benchmark was performed up to polynomial order 25; the monodimensional nature provides a one-to-one ratio between the order number and the number of terms evaluated. This results in a linear computational load with increasing order. The 2D Zernike benchmark was performed up to polynomial order 21; the multidimensional nature of the Zernike polynomials means that, for any given order, $(n+1)(n+2)/2$ terms must be evaluated. This results in an exponentially growing computational load with increasing order. Finally, the 3D cylindrical was performed up to polynomial order 21 for both the Legendre and Zernike components, stepped up equally in order. The cylindrical case results in an exponential computational load growth proportional to the product of the individual Legendre and Zernike growth rates.

The selection of the maximum orders in no way implies a limitation of the underlying code. Rather, it is highly unlikely that any FE/FET application will use higher orders. Thus, these are deemed sufficient to span the use cases while establishing behavior trends for thereafter.

The raw FE benchmarking data has been published via Mendeley Data [107].

4.4.1.1 Configuration

Evaluating the performance of a single evaluation is unrealistic on modern computing systems, as most expressions can be evaluated faster than most internally-available timers.

Thus, a dedicated loop-based testing program was created. It employed thousands of calculations to evaluate the average performance of each methodology. This produced a realistically measurable time proportional to the underlying algorithm's performance. The per-evaluation time was calculated by dividing the total time by the number of evaluations performed.

The `C` builtin function `clock()` was used to measure the duration of each set. The test data were written to a `CSV` file outside of the timing block. This ensured that disk I/O—a completely inconsistent activity time-wise—was not factored into the measured durations.

It has been observed that a compiler optimizer can and will eliminate entire blocks of code. This is done only if the code is found to have no side-effects, such as wait loops designed to consume processor cycles or a computation that returns an unused result. The benchmarking was protected against these unwanted 'optimizations' by adding the result of each FE evaluation to a dummy variable. The dummy variable was then recorded in the comma-separated values (csv) file as the 'answer'.

Conveniently, this answer also carried a tertiary benefit. Different methodologies were checked against each other using this value—much like using an `MD5` checksum to verify a file's integrity—since each should be mathematically equivalent and generate identical sums. The answer was recorded at an order-level granularity, and all algorithms were checked against the output of the original implementation code.

4.4.1.2 Methodologies

Four different methodologies were applied to each benchmark test. The first methodology is a reproduction of the original implementation by Tumulak in the preliminary studies [108]. The second is the optimized hybrid evaluation approach employed in subsequent research [109]. The last two are variations on vector-based computations.

Original This evaluation methodology mirrors the original implementation in Serpent. The evaluation component uses a recurrence relation for the Legendre polynomials and the

naive formulation for the Zernike polynomials. The convolution component is likewise a naive looping algorithm as shown by algorithm 1.

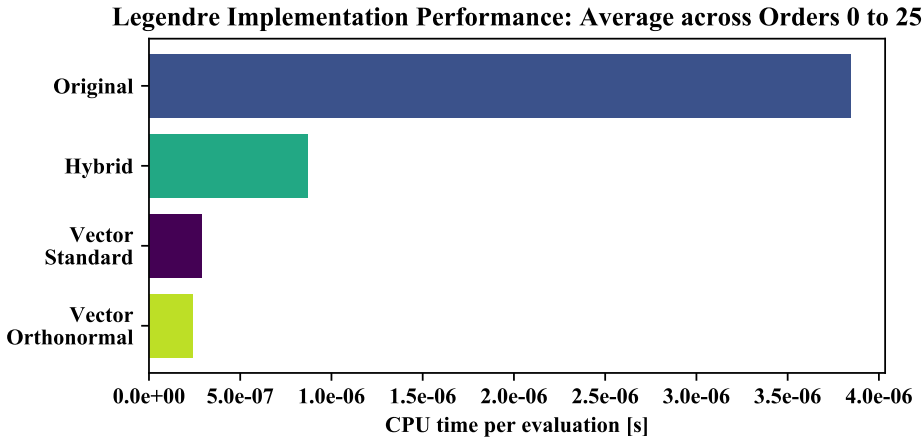
Hybrid The hybrid methodology represents the foundational work that was presented in the parent paper. The evaluation component uses a hybrid methodology for both the Legendre and Zernike polynomials. Chong's method is used as the recurrence relation for the Zernike polynomials. The convolution method used is presented in algorithm 2.

Vector The vector methodology is a completely new development. Because of the vector approach the evaluation and convolution components are tightly integrated; the evaluation component must generate vector data that can be used by the convolution component. The hybrid approach is embedded in the evaluation component. But, instead of returning a single value, an array is filled with the term evaluations. The convolution is performed as outlined by algorithm 4. Both relational forms presented in section 4.3.2—standard and orthonormal—were explored for efficacy in vector-based calculations.

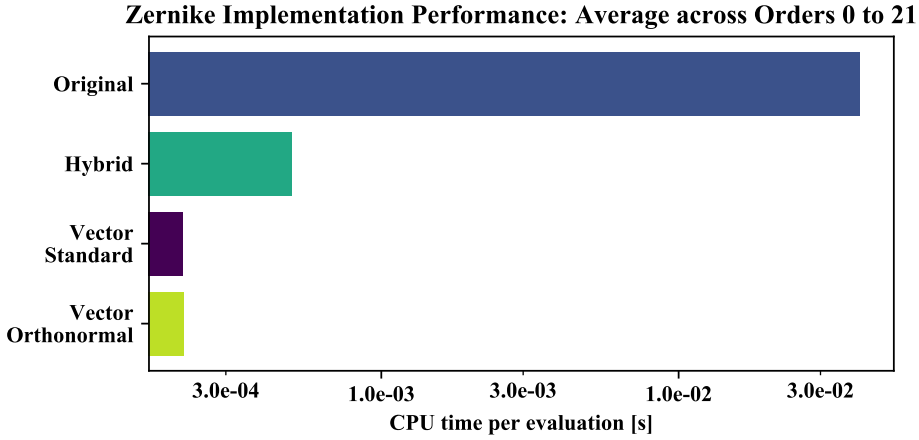
4.4.1.3 Results and Analysis

Overall Figure 4.1 shows the average time required to perform a full FE calculation over all benchmarked orders. This provides a rough guide to the performance that is expected from each methodology for any given order. The behavior trends are the same across all methodologies. As expected, the vector evaluation and convolution algorithms significantly reduced the computation time. The vector-based approaches are the fastest overall. Next, the hybrid method times are within an order of magnitude to the vector methodologies. Finally, the original implementation lags far behind, especially for the cylindrical convolution case in which the average time per evaluation was longer than *eight seconds*.

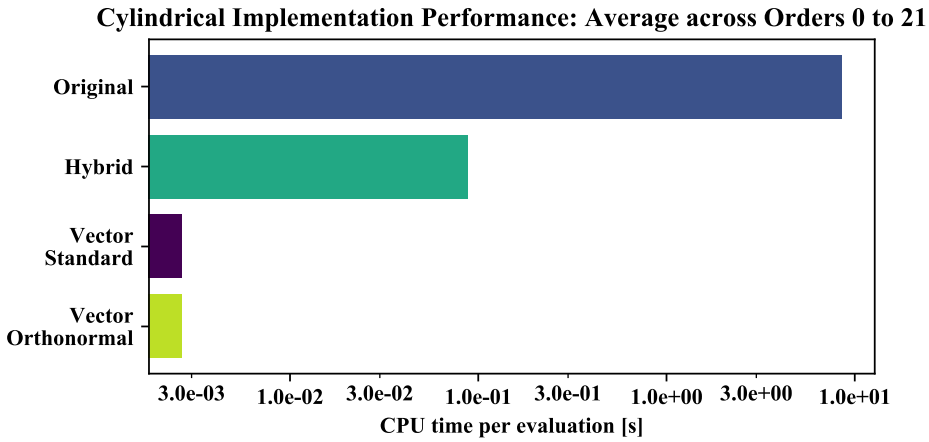
Breakdown The computational cost of the new methodologies, relative to the original implementation, are shown in table 4.2. Any time increases with respect to the original implementation are colored in red. These increases are present only in the lower orders, and



(a) Legendre polynomials



(b) Zernike polynomials (log scale)



(c) Cylindrical polynomial basis (log scale)

Figure 4.1 – The average computational times of various standalone FE methods, calculated from the individual times of each order's evaluation expense. The colors represent the actual value, ranging from blue (low) to gold (high).

Table 4.2 – Relative computational cost of each standalone methodology compared to the original implementation.

			Order				
Basis	**Methodology**	**Average**	**0**	**6**	**12**	**18**	**25**
Legendre	*Hybrid*	22.6 %	103 %	31.0 %	12.7 %	19.3 %	28.6 %
	Vector Standard	7.52 %	101 %	20.6 %	7.69 %	6.25 %	6.90 %
	Vector Orthonormal	6.15 %	101 %	21.2 %	8.67 %	5.35 %	4.45 %
		Average	**0**	**5**	**10**	**15**	**21**
Zernike	*Hybrid*	1.23 %	87.6 %	53.8 %	39.4 %	7.56 %	0.367 %
	Vector Standard	0.528 %	3540 %	62.7 %	19.9 %	3.08 %	0.133 %
	Vector Orthonormal	0.530 %	3360 %	63.8 %	20.0 %	3.12 %	0.133 %
		Average	**0**	**5**	**10**	**15**	**21**
Cylindrical	*Hybrid*	1.03 %	61.3 %	51.9 %	34.7 %	7.49 %	0.367 %
	Vector Standard	0.0312 %	5230 %	15.7 %	2.00 %	0.224 %	0.007 06 %
	Vector Orthonormal	0.0314 %	5210 %	15.7 %	2.04 %	0.229 %	0.007 09 %

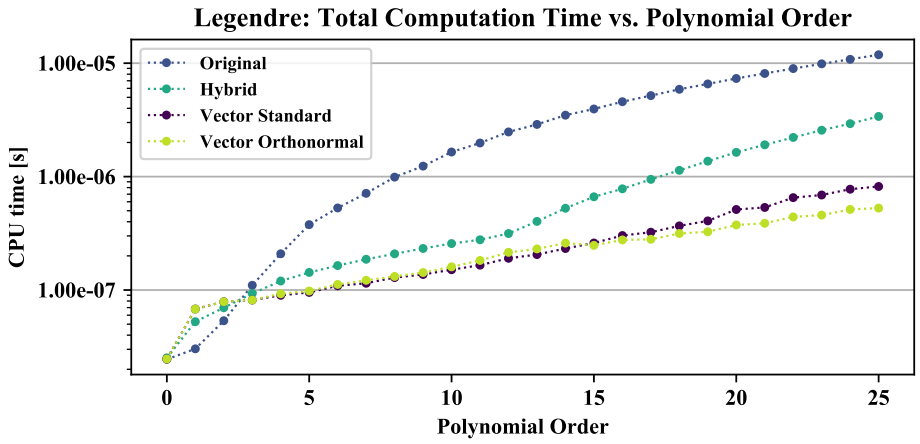
predominantly in the vector forms. The cause of this is the initialization overhead associated with the vector operations, such as zeroing arrays and additional condition checks.

Fortunately, these fixed costs do not invalidate the use case for vector FE evaluations. First, note that the actual computation costs are very low. Although a relative speed cost of 5.23×10^3 % is extremely large (cylindrical vector standard, order 0), the actual costs are small (1.33×10^{-5} s vs. 6.97×10^{-4} s). Second, the benefit of using vector-based methodologies quickly overcomes these fixed costs at low orders. Figure 4.2 illustrates these treads for all implementations. Thus, vector-based approaches should be adopted whenever possible; the most common use cases for FEs and FETs will be orders 5 or higher. Even if the underlying evaluation algorithm is unoptimized or resource-demanding, the utilization of a vector-based convolution algorithm will significantly reduce the overall computational time of any implementation.

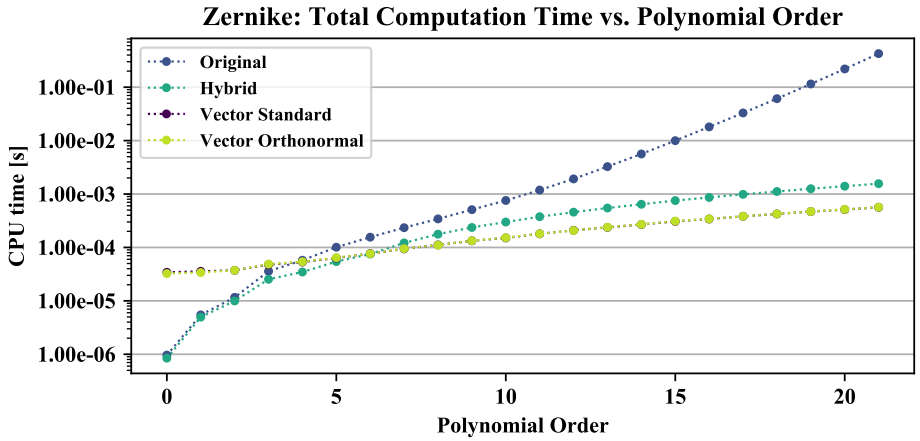
Figures 4.2 and 4.3 plot the benchmark data as a function of the order for all tested methodologies. Figure 4.2 shows the total time required by a particular methodology to completely evaluate all terms within the requested polynomial order. Figure 4.3 charts the incremental duration added to a full evaluation by increasing the requested order. In other words, the plots in fig. 4.2 can be reproduced by summing the order-wise results of fig. 4.3.

Interpretation There are a few interesting features worth highlighting among the data presented in table 4.2, fig. 4.2, and fig. 4.3. First, the vector methodologies exhibit the least growth in computation time with increasing order, i.e., a majority of the computation times are within an order of magnitude to each other. Figure 4.2 demonstrates this characteristic.

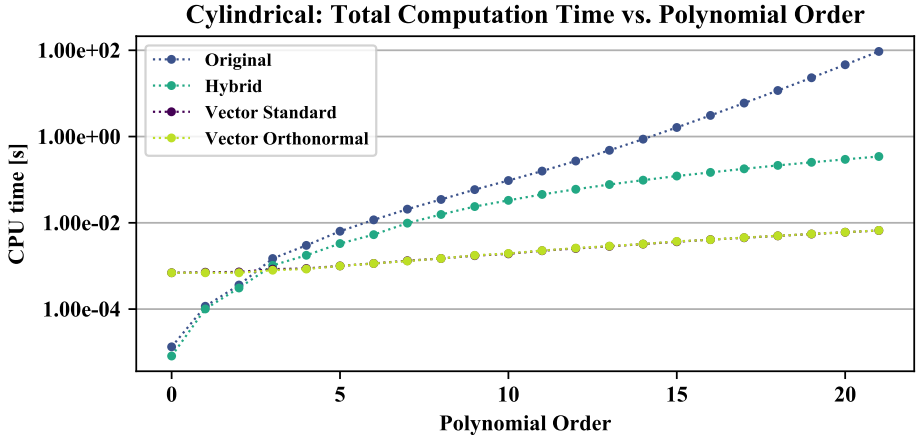
Second, both the standard and orthonormal vector methodologies have remarkably similar performance. The only exception is in the Legendre benchmarks, in which the standard form outperforms the orthonormal variant until order 15 which thereafter the orthonormal variant has better performance. This is due to the more efficient form of the orthonormal Legendre recurrence relation that assumes control after order 12. Overall, the comparison of the standard and orthonormal FE forms does not warrant a recommendation of one method as superior to the other. Instead, each situation should be considered



(a) Legendre polynomials

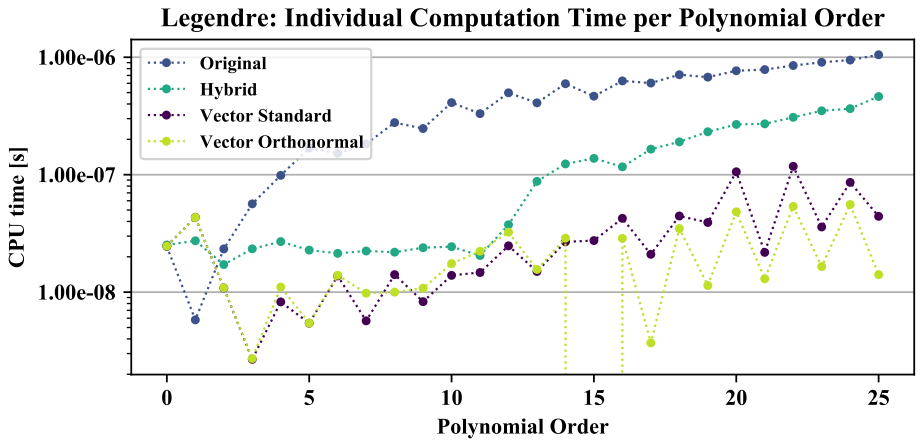


(b) Zernike polynomials

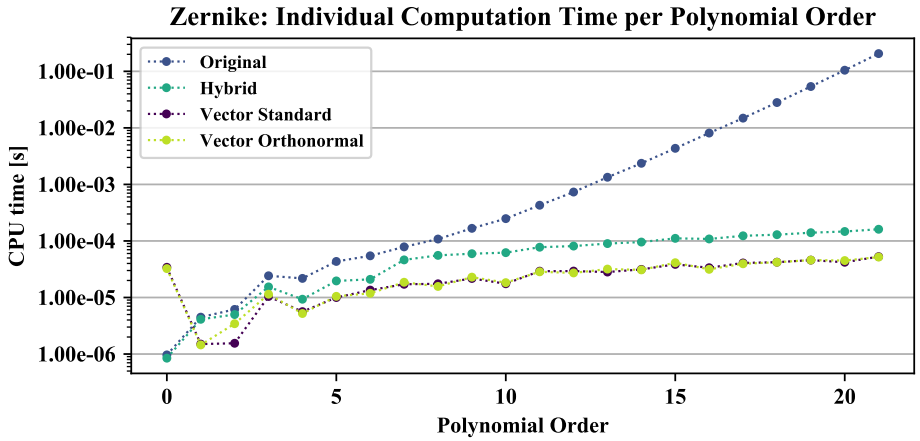


(c) Cylindrical polynomial basis

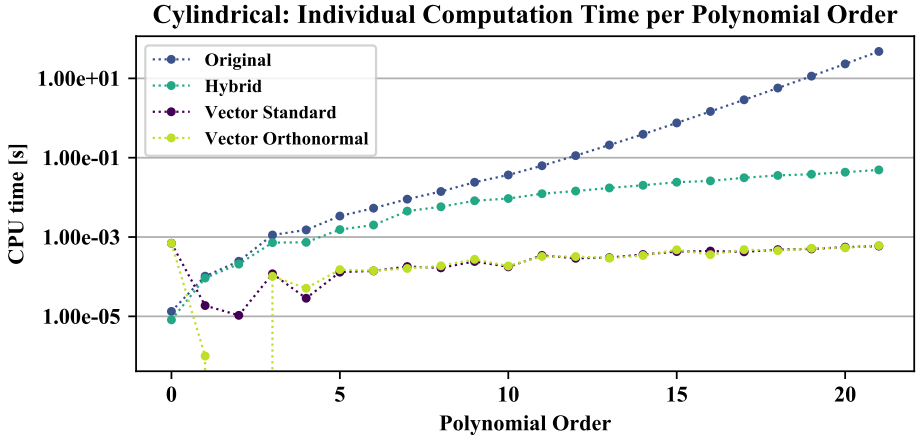
Figure 4.2 – The cumulative computational expense for each algorithm as a function of the polynomial order, evaluated in a standalone context.



(a) Legendre polynomials



(b) Zernike polynomials



(c) Cylindrical polynomial basis

Figure 4.3 – The term-wise computational expense for each algorithm type as a function of polynomial order, evaluated in a standalone context.

specifically with the decision resting on the application. The standard forms may be most useful if the corresponding function evaluations will be needed elsewhere in the same code. For example, this may include setting an initial or boundary condition using the functional series' shapes. Conversely, the orthonormal variants may simplify implementation if FES are the only use case for the functional evaluations.

Third, a couple orthonormal vector test orders were actually evaluated faster then the preceding orders. These are represented by the infinite drops in fig. 4.3a (orthonormal vector order 15) and fig. 4.3c (orthonormal vector order 2). These are not actual features of the methodology but artifacts of the compiler optimization and unique to each data run. The affected orders were inconsistent over multiple data runs, and could mostly be eliminated by changing the compiler flag to `-O2`.

Lastly, the relative performance of the vector Legendre polynomials decreases for higher orders. This is because all the evaluations use the recurrence relation for higher orders. Therefore, from the hybrid switch-over order between direct calculation to recurrence relation, all evaluations converge to the same rate of increase. However, these results still demonstrate that there are advantages to using the hybrid approach since only the lower-to-mid-order terms will be used a majority of the time.

4.4.2 Serpent Benchmark

In addition to being the focus of this work, Serpent provided a legitimate MC-based testing platform for assessing the performance of the presented methodologies.

The vector FET implementation was benchmarked to provide a quantitative characterization of the computational demands. Both mesh-based tallies and the original FET algorithms were benchmarked for comparison. All benchmarks were performed using two OpenMP (OMP) threads. Computational time was the only metric used, and the reported values were the duration average over five trials. Baseline performance benchmarks—created by running Serpent with all tallies off—provided the standard of performance for comparison.

Additionally, considerations must be made for FE-based approaches utilized in massively parallel simulations. Thus, multi-process performance was evaluated with various OMP and

message passing interface (MPI) options. The quality and quantity of the mesh and FE tally data were discussed. These also have an impact on distributive computing performance—considering at least both speed and fidelity contributions.

4.4.2.1 Benchmark Settings

All data were generated by running Serpent in criticality source simulation mode. Reproducibility was not enabled: unlike many other MC codes, Serpent does not guarantee reproducibility for MPI-parallelized contexts by default. This is a result of emphasizing scalability for large computing platforms. Conversely, OMP processes are reproducible as long as the random number seed is explicitly defined. As a result of this paradigm, trials without reproducibility should run much faster. Nevertheless, multiple non-reproducible trials will yield similar results to within a small statistical deviation (as would be expected with an MC code). The starting seed was still set identically in each trial.

The run parameters were set with 40 inactive generations to populate the fission source, then 500 active generations to collect results.[3] The neutron population was set to 2×10^4 per generation. Consequently, each tally was generated with approximately 10×10^6 neutron histories.

Tally methods were tested at four different grades: coarse, medium, fine, and ultrafine. The corresponding tally orders are defined in table 4.4. The chosen orders for each grade were empirically chosen to be representative of typical parameters that might be used for the desired tally resolution.[4]

4.4.2.2 Model Descriptions

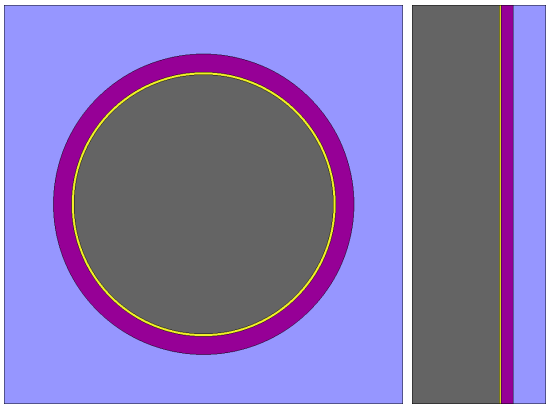
Two models were used for the Serpent benchmarking, one for each of the currently supported FET geometries: cylindrical and square-prismatic. These were based on the AP-1000 design specifications [110]. The relevant parameters are provided in table 4.3. Tallying was

[3] Inspecting of the Shannon entropy results indicated that at least 380 (of 400) inactive cycles were required to converge the axial fission distribution of the fission source. The outcomes of both 40 and 400 inactive cycles were similar enough that the smaller batch-size was selected for time-savings.

[4] These gradations are not previously established definitions; rather, they are qualitatively used in this context to distinguish the different levels of tally fineness.

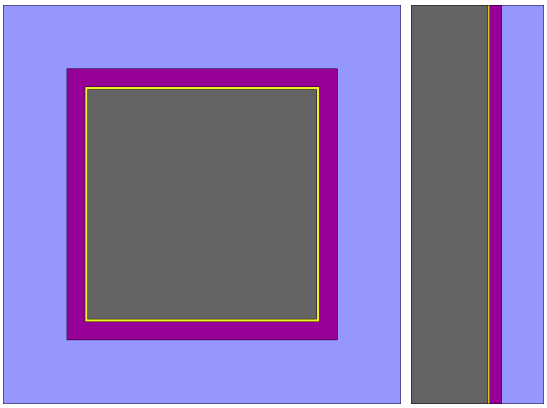
Table 4.3 – The parameters shared by both the cylindrical and square-prismatic models.

Parameter	Value
Fuel Height	366 cm
Fuel Density	10.424 g cm^{-3}
Fuel Enrichment	4.45 wt-% U^{235}
Fuel-Clad Gap	0.008 255 cm
Clad Thickness	0.057 15 cm
Cladding Material	ZIRLO™
Lattice Pitch	1.259 84 cm



X-Y Radial

(a) Cross section of the cylindrical fuel pin model.



X-Y Y-Z

(b) Cross section of the square prismatic fuel model.

Figure 4.4 – Representations of the simulated fuel model geometries in Serpent. Colors identify the different materials. From the center out, fuel is gray: ■, helium is yellow: ■, cladding is purple: ■, and water is blue: ■.

performed only in the fuel regions. A significant portion of the models used herein are based on those created by Tumulak for the original Serpent studies.[108]

Both models were simulated in an infinite square lattice. The fuel-clad gap was filled with helium, and the volume between the cladding and lattice edge was filled with light water. Neutron moderation in the coolant used data evaluated at 600 K.

Cross sections of the models, generated directly in Serpent, are plotted in figs. 4.4a and 4.4b. The left sides, section 4.4.2.2, are cuts in the X-Y plane looking in the axial direction. The right sides, section 4.4.2.2 (vertical not to scale), are cuts in the Y-Z plane from the axial center to the outside edge of the cell.

The cylindrical fuel pin is shown in fig. 4.4a. The fuel radius was 0.409 575 cm (0.161 25 in). This geometry tested the multivariate 3D cylindrical Legendre-Zernike FE basis convolution.

The square-prismatic model is shown in fig. 4.4b. The half-width was set to 0.362 975 cm (0.142 90 in) so that the total fuel volume was equal to the cylindrical fuel pin. This model was designed to test the multivariate 3D Cartesian FET assembled from three 1D Legendre polynomials.

4.4.2.3 Raw Computational Speed

The durations of different tally types and grades were benchmarked. Serpent's self-reported transport cycle times were used as the measurement value; simply, a faster methodology will produce a shorter trial duration. The results are shown in table 4.5.

Before continuing the discussion, it is imperative to mention that the effectiveness of FETs cannot be determined solely by comparing the computational time. The results of the benchmarking are a solid demonstration of the FET speed improvements, but do not represent the convergence efficiency. An actual quantitative comparison of the tally convergence will be presented in chapter 5. Regardless, it can be reconfirmed qualitatively that, when comparing methodologies based on the number of histories, FETs outperform mesh tallies. These implications of this are discussed further in section 4.4.2.5.

Also, these mesh tallies use collision-based estimators. Using track-length estimators would likely improve on the mesh tallies' qualities presented herein; unfortunately, track length estimators cannot be used in Serpent because of the underlying delta-tracking methodology.[5] Further, it is very difficult to implement track length estimators for the continuously-varying properties that are allowed in Serpent [111, 112]. This inherent support for variable properties in Serpent is a critical feature for enabling the use of FE-based methods in Serpent's multiphysics interface; these developments are presented and discussed in chapter 7.

Mesh vs. Vector FE The results of the coarse, medium and fine cylindrical mesh tallies are so similar that it is likely the actual performance cost is within the noise of the benchmarking;

[5]There are a few edge cases that employ limited use of track length estimators in Serpent, particularly the `dtl`-type detector for use within a single surface. Serpent does have the capability to perform surface-tracking, but it is not the primary method.

Table 4.4 – The test tally orders.

		Cylindrical			Cartesian
Type	**Grade**	ρ	ϕ	z	x, y, and z
Mesh Bins	*Coarse*	12	12	20	20
	Medium	16	16	50	50
	Fine	24	24	100	100
	Ultrafine	36	36	200	200
		ρ & ϕ (Zernike)		z (Legendre)	x, y, and z (Legendre ×3)
FET Order	*Coarse*	3		4	4
	Medium	5		6	8
	Fine	7		8	12
	Ultrafine	9		18	20

Table 4.5 – Serpent benchmark results: total average trial duration, percent increase with respect to the baseline, and qualitative results for the Cartesian tallies.

		Cylindrical		Cartesian			
Type	**Grade**	**Duration [min]**	**Change**	**Duration [min]**	**Change**	**Figure**	**Quality**
Baseline		7.83		8.01			
Mesh	*Coarse*	8.39	7.13 %	8.32	3.89 %	4.5c	Fair
	Medium	8.45	7.89 %	8.36	4.39 %	4.5d	Fair
	Fine	8.43	7.64 %	11.5	43.3 %	4.5e	Poor
	Ultrafine	8.70	11.1 %	13.4	67.0 %	4.5f	Unusable*
Vector FET	*Coarse*	9.70	23.9 %	9.76	21.8 %	4.5b	Excellent
	Medium	12.4	58.9 %	15.3	90.5 %	4.5	Excellent
	Fine	15.6	98.8 %	28.5	266 %	4.5b	Good
	Ultrafine	26.4	237 %	95.9	1100 %	4.5b	Good
Original FET	*Coarse*	9.88	23.3 %	9.79	22.2 %	Same as Vector FETs	
	Medium	17.16	114 %	29.7	270 %		

*The benchmark ultrafine mesh, run with only 10×10^6 histories, did not converge well (fig. 4.5f). The 'truth' ultrafine mesh, run with 4.35×10^9 histories, was mostly converged (see fig. 4.5a), albeit with a considerably longer calculation runtime that maxed out the available resources of the computing cluster.

most of the additional computational cost is likely due to the overhead when any tallying mechanism is enabled. The largest increase is 67.0 % for the ultrafine Cartesian mesh tally.

The performance of the vector FETs is as anticipated: utilization of the vector-based methodology drastically improved the computational performance of the FETs. The computational costs of the coarse and medium grades are actually very close to the computational cost of the mesh tallies, while the higher-order FET took longer as expected.

It is recognized that, based on the qualitative convergence results shown in fig. 4.5, the fine and ultrafine tests use a much higher order specification than is required to reproduce the distribution. However, it was important to include them on principle of demonstrating the results of the algorithmic modifications.

Original FE Original methodology trials were also performed, but not with the full suite of grades. Only the coarse and medium FET were benchmarked. The higher grades were actually started, but observed to have a much longer run time than was desired for carrying through to completion. Additional data points would be redundant, and the data from the coarse and medium grades were enough to demonstrate that these trends were also present with the Serpent implementation: the vector-based approach is better.

4.4.2.4 Parallelized Performance

Parallel benchmarking was performed with the options shown in table 4.6. These benchmarks were all performed on Plexi, a Beowulf cluster at Idaho State University (ISU). The objective was to provide an actual distributed computing environment. All tests were run with the equivalent of six processes. Additionally, just one trial was performed (instead of five) for each configuration. Only the ultrafine grade cylindrical geometry model was evaluated. This testing also used 10×10^6 neutron histories.

The different trials seem fairly consistent among the different parallel options. The only exception is the `-omp 1 -mpi 6` option that splits the tasks up among six different nodes. The 'sweet spot' of distributive computing contains a mix of OMP threads and MPI tasks; both of the mixed option trials (`-omp 3 -mpi 2` and `-omp2 -mpi 3`) have the fastest times overall.

Table 4.6 – Cylindrical FET distributive computing benchmarking.

	Baseline	Ultrafine Mesh		Ultrafine FET	
Parallel Options	**Time [min]**	**Time [min]**	**Change**	**Time [min]**	**Change**
`-omp 6` ⋆	20.9	23.1	10.5 %	48.9	134 %
`-omp 3 -mpi 2`	20.6	22.8	10.7 %	49.3	139 %
`-omp 2 -mpi 3`	20.2	22.1	9.41 %	49.6	146 %
`-omp 1 -mpi 6`	25.1	27.3	8.76 %	59.8	138 %

⋆Serpent was unable to run with the option '`-mpi 1`' so it was omitted. This was tried for consistency even though it is contextually meaningless.

No direct comparisons can be made between the local benchmarks reported in section 4.4.2.3. Not only are the cluster hardware components several years older than the local benchmarking platform, but the computing environment and architectures are completely different. An estimation of the slowdown can be inferred from the durations. The baseline test run with `-omp 6` is the closest analog to the local benchmarks and required 125 CPU-minutes to run, compared to the local two-threaded baseline that required 15.7 CPU-minutes. This demonstrates that the slowdown factor is around 8 times; conservatively, a value of 20 can be used.

Conveniently, relative performance changes between tests on the two platforms can be used to compare the performance of different methodologies in various environments. The duration of the mesh tally was roughly the same: 11.1 % local vs. 9.8 % average distributive, an overall slight improvement of 11.7 %. Surprisingly, the FET was comparatively much faster in a distributive environment: 237 % local vs. 139 % average distributive, an improvement of 41.4 %. This comparison provides encouragement that FETs are may be more suitable for a parallelized environment due to more efficient computation costs.[6]

Finally, it is noted that these duration results are not purely quantitative due to the fluctuating nature of the test environment. Specifically, controlling the CPU clock speed of each node in the cluster was simply not possible. They were manually checked throughout the benchmarking, and the range of operation observed was 2010 ± 1 MHz. Fluctuating

[6]These are preliminary results based on a data set that requires additional research. It is possible that, at these sparse task/thread and lower history levels, process overhead dominates the computational demands. Using more histories and larger parallelization options will shift the balance toward the actual sampling process, thereby revealing more information about the scalability.

Table 4.7 – The sizes of the detector data files.

	File Size [kB]			
	Cylindrical		Cartesian	
Grade	**Mesh**	**FE**	**Mesh**	**FE**
Coarse	208	4	573	10
Medium	916	11	8942	53
Fine	4112	24	72 180	157
Ultrafine	18 645	75	584 887	661

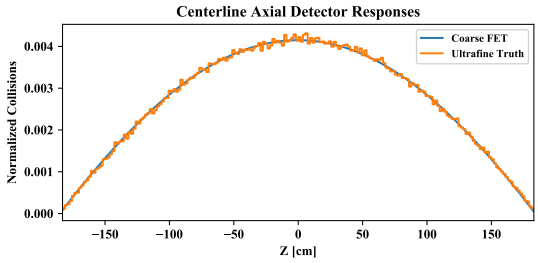
heat loads and network communications (from other users' concurrent jobs) also likely influenced the localized performance of each distributed task. Nevertheless, the qualitative results indicate that the distributive computing performance of both mesh and FE tallies are: 1) comparable or better than the local benchmarks performed, and 2) consistent over multiple parallelized configurations.

4.4.2.5 Data Quantity and Representation Quality

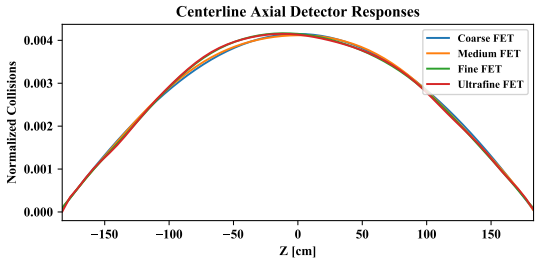
In a slight divergence from the core topic of raw computing performance, the file sizes of the meshes and FES detector outputs were compared. Serpent detector output files are set in the m-file format, a text-based script file that can be used to load the arrays into MATLAB® memory. The convergence of each tally type was also compared using visually-established qualitative statements about how closely each represented the underlying distribution.

One of the key features of FE is the ability to represent a distribution with a smaller data set. This plays an important role in large-scale computing where network bandwidth and latency are crucial factors of multi-task performance. Smaller data sets are communicated much faster between tasks/nodes than large data sets. The sizes of these files, shown in table 4.7, qualitatively represent the amount of data that will need to be exchanged in multi-task and/or coupled multiphysics simulations.

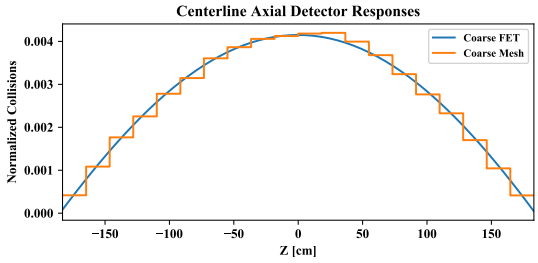
Data size becomes even more relevant when weighing the maximum transmission unit (MTU) of the interlink between tasks/nodes, e.g. ≈ 1.4 kB for Ethernet and 256 B–4 kB for InfiniBand™. Larger data sets implies more network transmissions required to transfer data, which means an increased potential for network saturation and/or communication



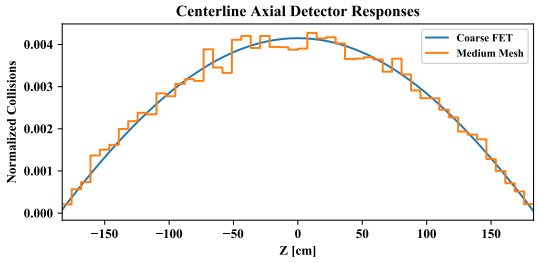
(a) Coarse FET (19.5 min) vs. Ultrafine 'truth' mesh using 4.35×10^9 histories (66 400 min).



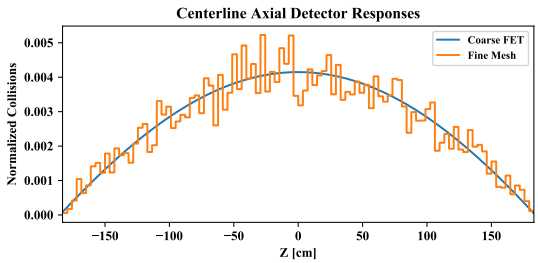
(b) Coarse, medium, fine, and ultrafine FETs (respectively 19.5 min, 30.6 min, 57.0 min, and 192 min). Note that the fine and ultrafine FETs exhibit over-sampling error.



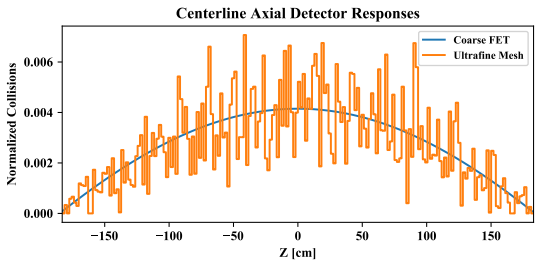
(c) Coarse FET (19.5 min) vs. coarse mesh (16.6 min).



(d) Coarse FET (19.5 min) vs. medium mesh (16.7 min).



(e) Coarse FET (19.5 min) vs. fine mesh (23.0 min).



(f) Coarse FET (19.5 min) vs. ultrafine mesh (26.8 min).

Figure 4.5 – Qualitative comparison between the coarse FET and various mesh tallies. The plots show a 1D slice of 3D Serpent detector datasets. All tallies are collected over 10×10^6 histories unless otherwise noted. The values in parenthesis are the CPU minutes required to generate each tally, calculated as the product of thread count and the runtime.

bottlenecks. This scenario is aggravated exponentially with an increasing number of individual tasks/nodes.

The most dramatic entry is the ultrafine Cartesian mesh detector—the detector output file is over 500 MB. Yet comparatively, the mesh tally has no additional resolution to offer over any of the FETs. This is poignantly illustrated in fig. 4.5, in which multiple tallies are plotted against a coarse-grade FET. Table 4.5 also provides comparative qualitative statements about the Cartesian-based tallies.

The 'truth' plot, fig. 4.5a shows this FET overlayed with an ultrafine mesh generated from 4.35×10^9 histories, the maximum that could be simulated on Plexi. This run required 13 nodes, each with 16 cores and 32 GB of memory. This much memory was needed because the simulation was run in source-term mode; any attempt to increase the number of histories resulted in memory allocation errors. The total runtime was 5.32 h. In fig. 4.5c the coarse mesh actually does a fairly decent job of capturing the general distribution, however there is a slight positive skew in the tally. The higher-grade meshes (figs. 4.5d to 4.5f) exhibit larger variances in the bin values; some of the bins in fig. 4.5f have no score at all.

Conversely, fig. 4.5b plots the medium, fine, and ultrafine FET grades against the coarse FET. These are all very similar, which indicates that lower-order FEs are still highly-suitable to most applications. In the case of the ultra-fine Cartesian tallies, the FETs produce a sampled distribution that is just as good or better than the 'truth' result with an order of hundreds to thousands fewer histories. Higher-order FEs become more relevant as the underlying distributions become asymmetric or more complex.

These results reinforce the proven characteristics of FETs—now augmented with vector-based algorithms—and equip them to expand their role in data storage and transmission systems.

4.5 Conclusions

FE are promising and mathematically rigorous tools that can improve the fidelity of multiphysics M&S. Significant strides have been made to improve the performance of FE-based

methods. Different functional bases, such as the Legendre or Zernike polynomial series, can be used to construct composite series to represent a variety of geometrical shapes.

Previous work performed demonstrated that FE can be used in a multiphysics coupling context [106, 113, 114]. Additional work was performed herein to reduce the computational burden of using FE-based methods. This was effected through the applications of: 1) a hybrid evaluation approach that efficiently melds the immediate benefits of direct calculations and the far-reaching impact of recurrence relations for functional term evaluations, and 2) a vector-based process that greatly improves the computation efficiency of evaluating all the terms in a functional series.

A known recurrence relation for the Legendre polynomials was already adapted in the Serpent FET customization. Chong's method, a Zernike polynomial recurrence relation developed for efficiency, was implemented in place of the naive expression [5, 100]. Two forms of the vector-based recurrence relations—standard and orthonormal—were also created.

An application was created for the sole purpose of benchmarking various FE evaluation methodologies. It was designed to independently test each existing approach at evaluating an FE. Benchmarking demonstrated that the vector methodologies were ideal with a slight preference toward the orthonormal forms.

The ability to use vector FE-based tallies was added to Serpent in the version `2.1.29` release. These tally mechanisms are available through the detector `det` card as the `dfet` detector type. This first release contained only a Legendre-based Cartesian detector type, with additional geometries and options planned for later releases. A second type, the composite Legendre-Zernike cylindrical FET, has been subsequently developed.

Next, the performance of the vector FETS, both Cartesian and cylindrical, were benchmarked in Serpent. The baseline of performance was an identical simulation model with all tallying disabled. Both mesh tallies and the original FET algorithms were similarly benchmarked for comparison. It was concluded that the vector algorithms are significantly faster than the original implementation. The qualitative advantages of FETS over mesh tallies was also reconfirmed.

Finally, FES lay the groundwork for easily transferring high-fidelity data between individual codes with different geometrical constructions. With these performance improvements in place, FE-based methods are poised for coupling use in the next generation of multiphysics reactor core modeling. Furthermore, these improvements expand the potential roles FE-based methods can fill within large scale distributive multiphysics M&S endeavors.

Chapter 5

FET Figure of Merit

We build too many walls and not enough bridges.

— Isaac Newton

One frequently-voiced concern of using FETs arises from the expensive computational cost caused by evaluating the entire functional basis set for each score to the tally. Do the computational expenses of FETs outweigh the benefits of a reduced sample set? The purpose of this chapter is to study and quantify the time-to-convergence of FETs. Recently optimized FETs in the MC reactor physics code Serpent will be used (see chapter 4). Specifically, it will be shown that FETs have a significant advantage over mesh tallies using computational runtime vs. convergence as a metric. These results will also inherently reconfirm prior convergence studies, in which the convergence of FETs was discussed as a function of sample population.

5.1 Convergence

Like all MC-based tallying methods, FETs are subject to both truncation and statistical errors. Another important realization is that some terms may be orthogonal—or nearly orthogonal—to the sampled distribution F. Accordingly, the variance estimators for these terms can be statistically large even though the coefficient estimators may be small or zero.

This chapter is adapted from Wendt and Kerby [115].

The challenge is to accurately represent the total uncertainty without an unreasonable bias toward less-meaningful terms.

A cost-to-benefit ratio has been proposed to evaluate the worth of each coefficient (note the use of $\widehat{a}_n$ instead of $\widehat{b}_n$) [76]:

$$R_n = \frac{\widehat{\sigma}^2_{\widehat{a}_n} c_n}{\widehat{a}_n^2} \tag{5.1}$$

The accompanying recommendation was to accept coefficients with $R_n \ll 1$ and reject those with $R_n \gg 1$. Unfortunately, the fate of coefficients with $R_n \approx 1$ is unsettled. One solution is to set an arbitrary limit, e.g. $R_n = 2.0$, as a cutoff. Without detailed insight into the sampled distribution F itself, though, this cutoff approach can be unsatisfactory as it does not optimize the balance between information and uncertainty contributions. Prior searches for an optimal cutoff are situation-specific [76, 83], implying that prescribing a generalized cutoff value is highly speculative. Other possible heuristics are still being investigated that may provide more generalized filtering recommendations than a manually-fixed cutoff value.

The need for an unbiased uncertainty measure—a quantitative metric of the data quality vs. uncertainty significance—induced development of a new approach to managing the statistical uncertainty analysis of an FET. First, it was decided that all evaluated values should be included in the analysis. Next, an equivalent measure must exist for mesh tallies. Finally, it must be evaluated in ℓ^2 space to capture the effects of non-zeroth (higher order) terms in orthogonal functional bases.

The proposed metric is a total relative variance U, which measures the uncertainty contribution $\|\sigma_i\|^2$ of each term with respect to the aggregated information $\sum\|v_i\|^2$ provided by the values themselves:

$$U_i = \frac{\|\sigma_i\|^2}{\sum\|v_i\|^2} \tag{5.2}$$

$$U = \sum U_i = \frac{\sum\|\sigma_i\|^2}{\sum\|v_i\|^2} \tag{5.3}$$

This metric is not a "true" uncertainty, but a sensible measure found to be very useful for this time-to-convergence analysis.

U_{FET} Derivation The uncertainty and value contributions, with the resulting total relative variance, are:

$$\|\sigma_i\|^2 = \|\widehat{\sigma}_{\widehat{b}_i}\|^2 = \int_\Gamma \left(\widehat{\sigma}_{\widehat{b}_i}\psi_i\right)^2 \omega_i = \frac{\widehat{\sigma}^2_{\widehat{b}_i}}{c_i} \tag{5.4}$$

$$\|v_i\|^2 = \|\widehat{b}_i\|^2 = \int_\Gamma \left(\widehat{b}_i\psi_i\right)^2 \omega_i = \frac{\widehat{b}_i^2}{c_i} \tag{5.5}$$

$$U_{\text{FET}} = \frac{\sum \frac{\widehat{\sigma}^2_{\widehat{b}_i}}{c_i}}{\sum \frac{\widehat{b}_i^2}{c_i}} \tag{5.6}$$

Note that the bias of $\sigma \approx \sqrt{\sigma^2}$, a variance to standard deviation relation used in eq. (5.4), is less than 0.03 % when the sample size is greater than 1000 (a condition satisfied for nearly all MC-derived distributions). For clarity, the coefficient value $\widehat{b}_i$ is collected during the sampling process using eq. (3.65), and the corresponding statistical variance $\widehat{\sigma}_{\widehat{b}_i}$ using eq. (3.74).

U_{mesh} Derivation The equivalent mesh tally form is:

$$\|\sigma_i\|^2 = \int_b \sigma_b^2 dV = \sigma_b^2 V_b \tag{5.7}$$

$$\|v_i\|^2 = \int_b v_b^2 dV = v_b^2 V_b \tag{5.8}$$

$$U_{\text{mesh}} = \frac{\sum \sigma_b^2 V_b}{\sum v_b^2 V_b} \xRightarrow{\text{uniform } V_b} U_{\text{mesh}} = \frac{\sum \sigma_b^2}{\sum v_b^2} \tag{5.9}$$

for bin b and volume V.

Figure of Merit A figure-of-merit (FOM) is appropriate for this context, in which the effectiveness of an approach is measured against the runtime. Using the total relative variance U as the uncertainty measure, the time-to-convergence FOM is:

$$\text{FOM}_x = \frac{1}{U_x t} \tag{5.10}$$

where t is the trial's runtime. In this context, a higher FOM means better performance.

5.2 Methodology

From the International Criticality Safety Benchmark Evaluation Project (ICSBEP), benchmarks PU-COMP-MIXED-001 case 5 (unreflected) and PU-COMP-MIXED-002 case 23 (reflected) were selected as the base models for testing [116]. These were selected due to: 1) uncomplicated models, 2) reflected and unreflected configurations, and 3) common dimensions.

5.2.1 Testing

The benchmark trials were run on a Beowulf cluster, constructed of nodes containing eight CPU cores clocked at 2.0 GHz and 16 GB RAM. Each trial was run on a separate node from the others to isolated each trial's performance and ensure the highest level of computational environment similarity. Serpent was run with 8 OMP threads to fully utilize each node's CPU resources, and its self-reported transport cycle duration was collected as the runtime metric.

All tallies were generated using Serpent's detector functionality. Each trial contained 1100 generations with the first 100 discarded; the Shannon entropy initially verified to ensure source term convergence before the active cycles were begun. The FETs were based on a convolution of three Legendre polynomial series (one for each dimension) to create a fully multivariate 3D functional basis. The mesh tallies were uniformly divided so that the bin volumes were equal. Well-resolved high-resolution "truth" mesh tallies were also generated to provide a sensible and accurate approximation of the underlying sampled distribution S. These "truth" tallies were utilized to provide an additional comparative measurement of the trial tallies' overall statistical and truncation error.

The "truth" tallies were recorded with 1×10^{10} total particles during the active cycle. Coarse and fine trial tally granularities were defined as shown in table 5.1; the "truth" tally specifications are also provided. Since the objective was to research the time-based convergence of each tally type, short, medium, and long trial runtimes were performed with respective total particle sources of 1×10^{6}, 1×10^{7}, and 1×10^{8}.

Table 5.1 – The granularities used, and the total number of values tracked, by each tally type.

Type		Specification	Total Values
FET		Order	
	Coarse	$4 \times 4 \times 4$	125
	Fine	$8 \times 8 \times 8$	729
Mesh Tally		Bins	
	Coarse	$8 \times 8 \times 8$	512
	Fine	$32 \times 32 \times 32$	32 768
	"Truth"	$100 \times 100 \times 100$	1 000 000

Note: the Cartesian domain correlation of the tally specification is: $x \times y \times z$

5.2.2 Comparisons and Analysis

All tallies were normalized to a count density of 1 over the detector volume. For the FETs this required a transformation from the functional domain to the physical model. The total relative uncertainties were calculated using eqs. (5.6) and (5.9).

The trial tally results were then compared to the "truth" tally to individual accuracies. A full 3D comparison was not performed; instead, 2D data slices were evaluated using perpendicular planes passing though 29.5 %, 45.5 %, 79.5 %, and 95.5 % of each dimension's length, resulting in twelve slices per trial mesh comparison. These positions were chosen from the nearest common bin centroids shared among the three mesh tally granularities, enabling the most accurate comparison between the mesh tallies. Each FET's error was found by numerically evaluating the expansion using eq. (3.75) over a fine quadrature, calculating average values over "truth" tally bin ranges, then accumulating the differences with the corresponding "truth" tally bin values. Each mesh tally's errors were evaluated at a resolution constructed as the union of the bin-edge sets from itself and the "truth" tally. The bin values were projected onto this common mesh and the errors accumulated as the volume-weighted difference with the collocated "truth" bin. The standard deviation of these accumulations represented the degree to which a trial tally deviated from the "truth"; therefore, it was used as the measure of overall relative error.

Table 5.2 – Raw data for the reflected benchmarks; plots are presented on the right sides of figs. 5.2 and 5.3.

Tally	Time [h]	U	Error [%]
"Truth"	285.4	—	—
Coarse FET	0.041 94	6.23×10^{-5}	2.13
	0.3842	6.21×10^{-6}	1.65
	3.812	6.20×10^{-7}	1.50
Fine FET	0.089 17	2.37×10^{-4}	2.39
	0.8647	2.37×10^{-5}	1.02
	8.589	2.36×10^{-6}	0.58
Coarse Mesh	0.030 83	2.50×10^{-4}	7.52
	0.2844	2.51×10^{-5}	7.25
	2.749	2.48×10^{-6}	7.19
Fine Mesh	0.038 88	6.47×10^{-3}	9.21
	0.2917	6.49×10^{-4}	3.41
	2.771	6.48×10^{-5}	2.04

5.3 Results

The eigenvalues reported by Serpent were in agreement with the benchmarks, indicating that the models were correctly defined in the input files. The analysis results of PU-COMP-MIXED-002 Case 23 are shown in table 5.2. Also, fig. 5.1 compares two fine tally trials—the FET medium-runtime and mesh tally long-runtime—to the "truth" tally. The full analysis for both benchmarks plotted in figs. 5.2 and 5.3; the optimal performance region is in the lower-left corner of each graph.

First, and foundationally requisite to these results and analyses, fig. 5.2 and table 5.2 exhibit a near-perfect $U \propto \frac{1}{t}$ relationship in log-log space. This indicates that U is a sensible measure for evaluating the statistical convergence of both FETs and mesh tallies. This also means that U is an appropriate uncertainty measure for use in calculating the FOM according to eq. (5.10). The final FOM for each tally type, shown in table 5.3, was calculated as the average of the individual FOMs from each trial duration. The FET FOMs are consistently higher than the mesh FOMs, sometimes by more than a factor of ten.

Second, fig. 5.2 reconfirms that the time-based FET performance is consistently better than the mesh tallies as indicated by the FOMs. This is apparent when comparing the

Table 5.3 – Final FOMs of the tally types for each benchmark model, organized by granularity.

Granularity	Type	Unreflected	Reflected
Coarse	FET	$3.57 \times 10^{+6}$	$9.80 \times 10^{+6}$
	Mesh	$6.70 \times 10^{+5}$	$3.33 \times 10^{+6}$
Fine	FET	$6.92 \times 10^{+5}$	$1.16 \times 10^{+6}$
	Mesh	$2.99 \times 10^{+4}$	$1.19 \times 10^{+5}$

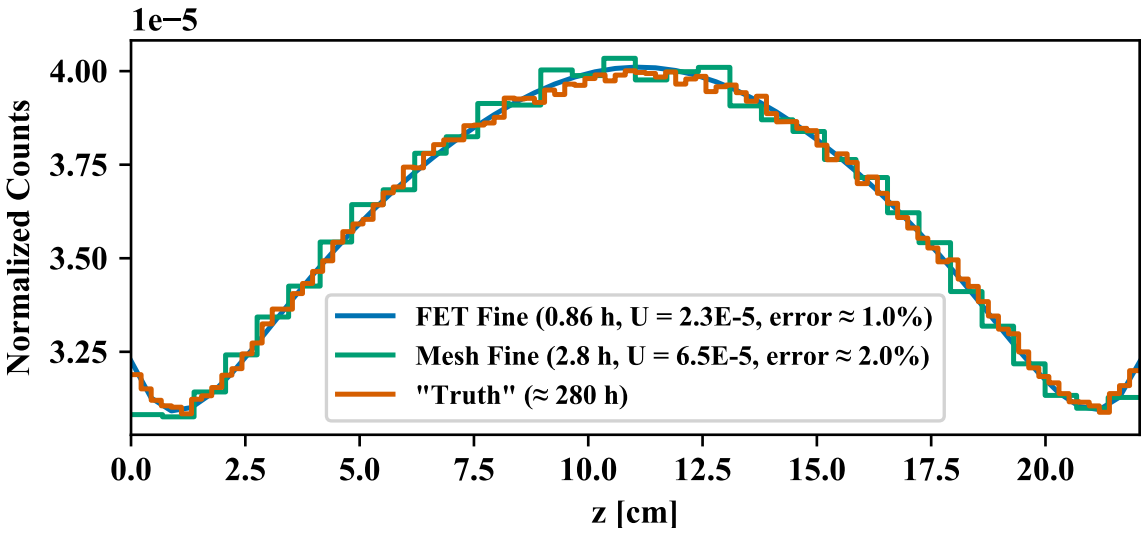


Figure 5.1 – Comparison of fine tally methods for the reflected model, with the 3D data sliced at 45.5% of the X and Y axes to produce a 1D plot in Z. The numbers in parenthesis provide information about each tally's results. The values for U are taken from table 5.2, and the reported error values are the average relative error compared to the "truth" tally.

short runtime FETs to the long runtime mesh tallies: the short FETs require only minutes of runtime to surpass the quality of data provided by the mesh tallies running for over two hours. Figure 5.1 is an especially poignant demonstration, in which the fine FET tally shows remarkable similarity to the "truth" tally. Further, it requires only a fraction of the fine mesh tally's longer runtime and produces a more accurate result. This is a strong confirmation of the hypothesis: despite a heavier per-sample computational footprint, FETs have a faster time-to-convergence rate than mesh tallies.

Third, inspection of fig. 5.3 reveals that the coarser tallies have a lower U but a higher overall relative error due to truncation effects. Interestingly enough, fig. 5.2 shows that, for the unreflected case, the coarse mesh tally and fine FET have exactly the same convergence rate; however, when checking the overall relative errors in fig. 5.3 it is apparent that the

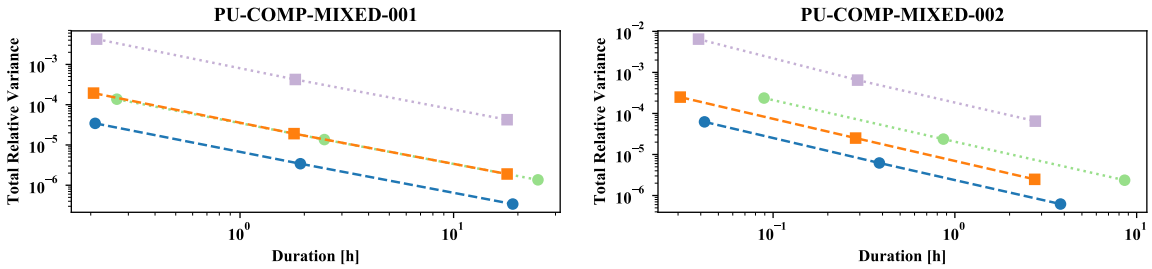


Figure 5.2 – Comparison of the total relative variances as a function of trial runtime (both axes are logarithmically scaled).

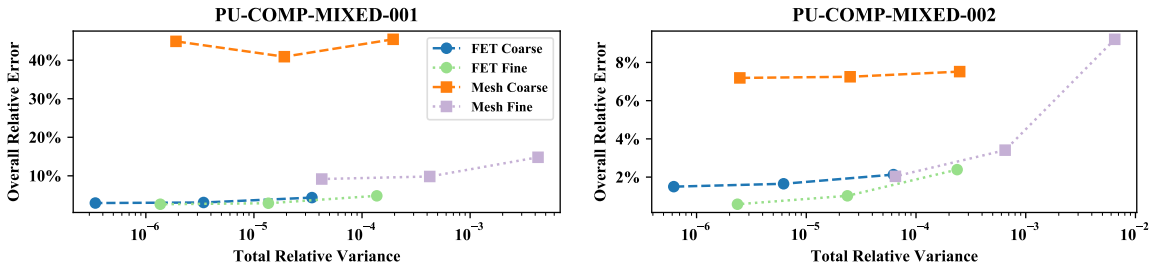


Figure 5.3 – Relationship between the total relative variances and the overall relative errors for the different time-to-convergence trials (the horizontal axis is logarithmically scaled; the legend is common for figs. 5.2 and 5.3).

FET provides a much better representation. The fine FET is tracking 729 coefficient values, compared to the coarse mesh tally's 512 bins. This places the FET at a statistical disadvantage, making the outcome in favor of the FET even more impressive.

Fourth, the reflected trials have better overall results. This is primarily because the reflection causes the neutron profile to be less varying throughout the detector volumes. Paying particular attention to the y-axis scale, comparing the left and right sides of fig. 5.3 demonstrates how both tally methods benefit from this effect.

Finally, the trial tallies appear to asymptotically approach a minimum error-to-"truth" value for longer computational times, especially for the unreflected benchmark (PU-COMP-MIXED-001). It is inferred via fig. 5.3 that all the mesh tallies—with the exception of the reflected fine trial—are unlikely to reach the minimal error values of the FETs; finer resolutions (needing a significantly longer runtime) would be required.

5.4 Conclusions

FETs have been verified be computationally advantageous due to superior convergence qualities, despite an algorithmically heavier tallying methodology. They showed a performance speedup of several times over a fine mesh—and hundreds over a "truth" tally (see table 5.2)—while consistently providing a higher-quality representation than a coarse mesh. Low-order FETs are suitable to capture high-fidelity information of simple distributions with very short runtimes. Higher-order FETs are required when capturing more complex distributions, yet can still provide suitable results with medium-length runtimes.

In high-performance computing contexts, gains of 10 % to 20 % are significant; the performance advantages of FETs are monumental: often several times faster (or more) than mesh tallies providing a comparable quality. Consequently, FETs have an impressive case for consideration as the primary MC tally mechanism. Finally, these results firmly support the role of FET methods in MC-coupled multiphysics simulations seeking reduced computational time and statistical error while insuring or improving fidelity.

Chapter 6

Functional Expansions in MOOSE

> *All the mathematical sciences are founded on relations between physical laws and laws of numbers, so that the aim of exact science is to reduce the problems of nature to the determination of quantities by operations with numbers.*
>
> — James C. Maxwell

Finite element solutions—used for numerically solving differential equations over an analytically-challenging geometry—may contain a large number of elements for high-fidelity multiphysics simulations. Coupling between decomposed (split) domains, and/or separate physics modules, can involve substantial communication as variable values are transferred for each quadrature point or element average. Conversely, the desired use of FE data transfers is motivated primarily by the fact that a complex spatial distribution can be reduced to a handful of expansion coefficients. Only these coefficients need to be transferred for communicating the underlying data.

MOOSE is an open source finite element-based code, and thus does not stochastically sample data as a series of events like the Serpent MC code. Instead, a solve is performed by numerically evaluating and minimizing a solution field over a set of quadrature points based on differential equations supplied by the user [118]. A MOOSE simulation is constructed

This chapter is adapted from Wendt *et al.* [117].

from action parameters that are specified in an input file. For example, the `Mesh` action is used to create or import the mesh structure used to solve the simulation. Another, `Variables`, defines the variables that are used in the solution. The `Kernels` actions define the physics equations that will be solved in the simulation. The associated `ICs` and `BCs` actions specify initial conditions and boundary conditions, respectively.

Non-MOOSE codes can also be coupled to MOOSE applications due to the flexibility of the MOOSE MultiApp system, available via the `MultiApp` action. Transfers between different MultiApps are performed by `Transfers` action items. For data transfers with external codes, there is no guarantee of a simple quadrature point-to-quadrature point, or element-to-element correspondence. MOOSE makes no assumptions and establishes no constraints on the model space used in external apps; the information exchange is entirely the responsibility of the researcher. In this case, the use of FE transfers is particularly useful for exchanging data between the MOOSE application and other codes. Describing solution fields in terms of an FE greatly simplifies the conversion and mapping process between codes with distinct model constructions. In this research, a generalized framework will be provided for capturing, transferring, and expanding data into/from variable fields using functional expansions.

6.1 Implementation

It is extremely desirable to implement FE-based coupling that is flexible, simple-to-use, and widely available. A module to the MOOSE framework was identified as the best fit to these criteria, as it would make the methodology accessible to all users without causing bloat in the base framework. This module was created with the name `functional_expansion_tools`, and became an official part of the MOOSE open source codebase on February 15, 2018.

An underlying architecture to simplify future additions was required. Support for different types of FEs was also necessary, as multiphysics simulations depend on both volumetric- and surface-based coupling methodologies. These requirements were fulfilled by leveraging inheritance and polymorphism. Interfaces were used to specify behaviors for

inter-class interactions. Concrete classes were provided, which implemented these interfaces to perform the FE-related operations. A high-level organizational structure is shown in fig. 6.1.

However, it is important to highlight a slight shift in nomenclature from the convention adopted elsewhere in this dissertation. The initialism 'FE' is often understood within the MOOSE ecosystem to mean 'finite element'. To avoid confusion, the developers of MOOSE recommended that the initialism 'FX' be used instead for functional expansions. Thus, many class names described throughout this chapter use 'FX' to maintain consistency with this suggested nomenclature.

6.1.1 Function Series

`FunctionSeries` is the core of the FE-related capabilities in the `functional_expansion_tools` module. Because `FunctionSeries` is a child class of `Function` (one of the standard MOOSE class types), it can be used wherever a MOOSE `Function` would be used. Some applications include setting variable `ICs`, contributing to `Kernels` calculations, or defining `BCs` in non-FE contexts. `FunctionSeries` also contains a capability to memoize evaluations for reuse in repetitive evaluations. This memoization stores the results of an FE evaluation in a map using a hash value of the quadrature point as the key. As long as the coefficients remain unchanged, these evaluations can be quickly retrieved without re-computing the entire FE. This is particularly useful when a certain evaluation needs to be performed repeatedly in a calculation, such as when used directly by a physics solver object like a `Kernel` or `BC`. The memoization capability is disabled by default—a majority of the use cases require only a single evaluation for each quadrature point after each Picard iteration and coefficient transfer.

The design of the `FunctionSeries` and associated classes is based on the principle that a multivariate function (a "composite" series) can be constructed by convolving one or more individually-defined functions (a "single" series). Examples of composite series are the `Cartesian` and `CylindricalDuo` classes, which are concrete implementations of the `CompositeSeriesBasisInterface` (`CSBI`) interface. Each `CSBI` object contains a vector of references to one or more `SingleSeriesBasisInterface` (`SSBI`)-derived single series, from

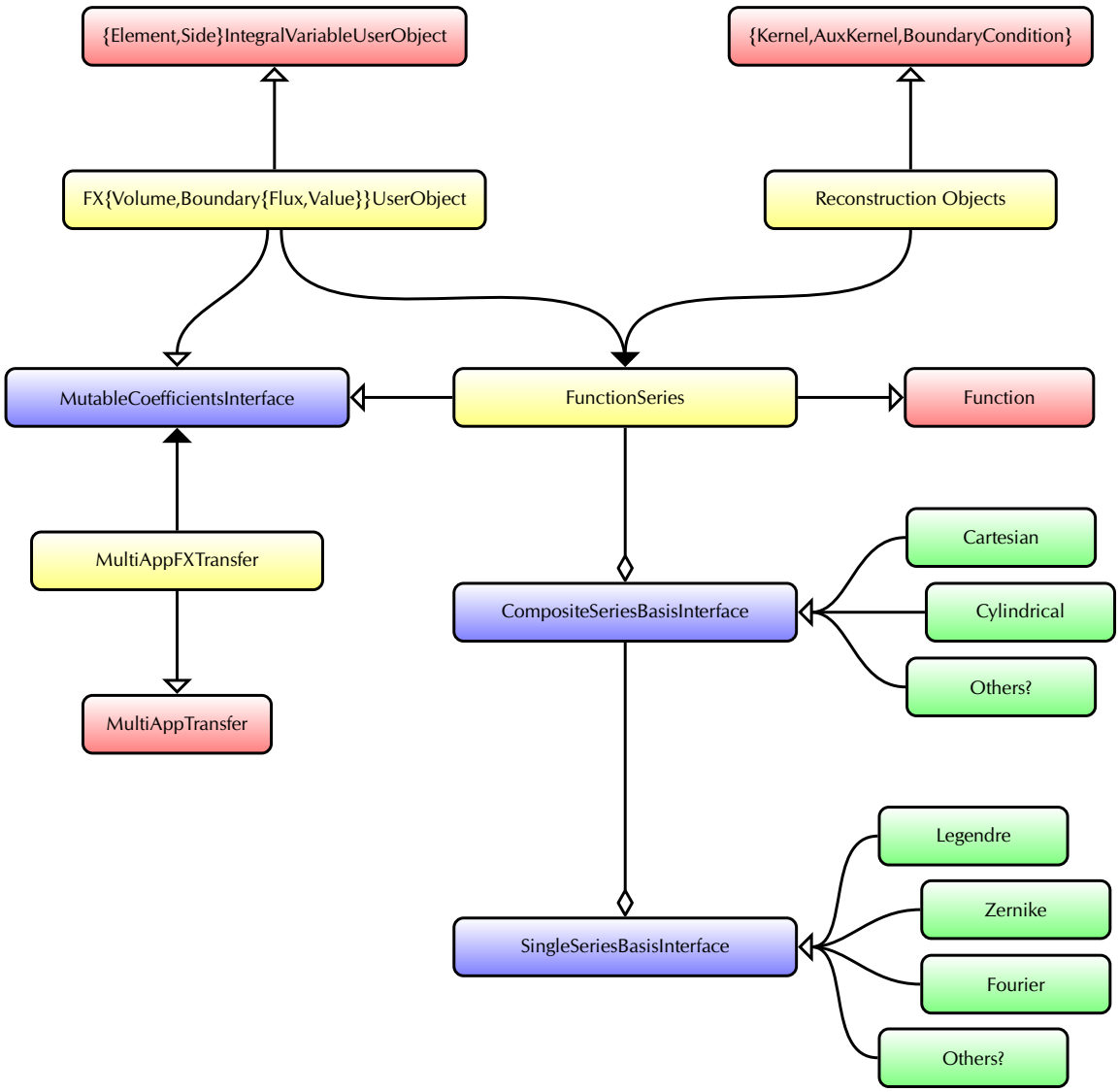


Figure 6.1 – The architecture of the FE module in MOOSE. The boxes are colored based on class type: red = existing MOOSE classes; blue = new interface definitions; yellow = external concrete classes available to users; green = internal concrete classes. The tips represent the relationships: filled arrow = a requires/uses association, open arrow = inheritance; diamond = composition.

which the composite series is constructed. The dimensionality of a single series is not limited to 1D; any series described by a single expansion order, such as the 2D Zernike polynomials or 3D spherical harmonics, can be fully implemented through the `SSBI` interface. Each `CSBI` implementation must provide a full-3D functional basis; however, lower-dimensional composite bases can still be accessed by setting the corresponding order(s) of the unneeded dimension(s) to zero. Both Legendre and Zernike polynomials are currently implemented as single `SSBI` series. From these can be constructed both Cartesian and cylindrical composite series.

In the MOOSE input file, the user specifies the type of composite series to be constructed using the `FunctionSeries` parameter '`series_type`'. The underlying single series are specified using the parameters '`disc`', '`x`', '`y`', and '`z`'. The choice of composite series dictates the allowable single series to use. For example, it is not possible to use the Zernike polynomials when creating a Cartesian composite series FE; error checking is performed to ensure that the series' specifications are sane. The '`orders`' parameter is then used to specify the orders for each of the single series. Finally, '`physical_bounds`' is used to specify the FE domain in the MOOSE model space. Multiple `FunctionSeries` can be defined in a single input file, each with a unique name and providing a distinct FE instance. This may be useful if multiple FEs are needed to: 1) communicate with more than one MultiApp, 2) evaluate the same variable field with a different series or order basis, and/or 3) collect data from multiple variable fields.

6.1.2 User Objects

UserObject-based classes provide the infrastructure for generating the FE coefficients from a variable field. A policy-based class, `FXIntegralBaseUserObject`, was created to facilitate future implementations. The polymorphic policy-based design also enables FE coefficient generation from the numerical integration routines provided by most UserObjects. The only requirements are that the policy class: 1) be a child of the MOOSE UserObject class, 2) provide the numerical integration methods `computeIntegral()` and `computeQpIntegral()`,

and 3) provide information about the finite element objects being integrated over via the methods `getCentroid()` and `getVolume()`.

Primarily, the UserObjects `ElementIntegralUserObject` and `SideIntegralUserObject` intrinsically satisfy the first and second requirements. `ElementIntegralUserObject` also meets the third requirement by default; however, more coding is needed for `SideIntegral`-based policies. To provide assistance to any future `SideIntegral`-based implementations, these methods were completed and added to the module as the `FXBoundaryBaseUserObject` policy class. Any new `SideIntegral`-based functional expansion classes are encouraged to derive from this policy.

As long as these three requirements are met, `FXIntegralBaseUserObject`-derived objects can be applied to render any sort of scalar distribution in a MOOSE problem as an FE. Examples of possible scalar quantities include temperature, density, concentration, energy, power, speed, pressure, dose, and luminosity. It is conceivable for the `functional_expansion_tools` module implementation to be extended for use with vector quantities; however, the current implementation is focused on representing scalar quantities in up to three dimensions.

Further, `FXIntegralBaseUserObject` depends on a `FunctionSeries` instance to provide the functional basis used when generating the expansion coefficients via numerical integration. These numerical integration algorithms are adapted from existing classes in MOOSE. All integrals can be calculated using the quadrature capabilities of libMesh in the finite element structures of MOOSE via its UserObject action classes. Because of the policy-based design centered on MOOSE UserObjects, further specialization for parallel computations is not required. In other words, this design inherently permits multi-subdomain evaluation and aggregation of FE coefficients from field data distributed across many nodes and processes in MPI and/or OMP contexts.

Three basic policies were provided with the first `functional_expansion_module` commit. First, the class `FXVolumeUserObject` inherits from `FXIntegralBaseUserObject` using `ElementIntegralUserObject` as the policy. It is used to generate a volumetric FE from one variable field. Second, `FXBoundaryValueUserObject` inherits from `FXBoundaryBaseUserObject` without any further specialization. It generates an FE using a variable field's values at a named

boundary. Third, FXBoundaryFluxUserObject inherits from FXBoundaryBaseUserObject with additional specialization to adapt the integration to a flux calculation, particularly by overriding the existing code for computeQpIntegral(). It generates an FE from a variable field gradient at a boundary to calculate the flux across the boundary; this flux calculation also requires a MOOSE-Material with a diffusion coefficient to be provided in the MOOSE input file.

6.1.3 Reconstruction Objects

The reconstruction objects use a FunctionSeries instance to evaluate an FE, then insert the result into the problem solution. Explicitly supported MOOSE actions currently include Kernels, AuxKernels, and BCs; other components may be added in the future as needed.

FunctionSeriesToAux (AuxKernel) Using FunctionSeriesToAux, derived from the standard MOOSE class AuxKernel, an FE can be expanded into an AuxVariable field. This automatically expands the FE at the appropriate times throughout the solve, and can be useful for secondary measures that are evaluated in the auxiliary variable space. Further, an AuxVariable solution field can be inserted into a Variable's solution using the existing CoupledForce Kernel class in MOOSE. However, AuxVariables are numerically simplified over regular Variables within the MOOSE finite element framework, so using this approach may incur some information loss.

BodyForce (Kernel) FEs can be incorporated directly into a Variable's solution using BodyForce, which is the recommended approach over the FunctionSeriesToAux → AuxVariable → CoupledForce method. BodyForce itself is already part of the MOOSE framework, and is used to insert the output of a Function into a Variable's solution. Accordingly, it is mentioned here for completeness although it is not part of the functional_expansion_tools module. As such, it is strongly encouraged to enable the memoization feature of the associated FunctionSeries when coupling directly via BodyForce. Otherwise, each solve step

will be slower since the corresponding FunctionSeries will be required to re-evaluate the entire FE during each linear and nonlinear iteration.

Boundary conditions (BCs) Direct use of a FunctionSeries is required for FE-based boundary conditions. Thus, all the BC reconstruction objects automatically enable the memoization feature of the associated FunctionSeries. Three classes have been created for this purpose.

FXValueBC This class is derived from FunctionDirichletBC. It enforces a value boundary condition on a sideset of the solution model by using the reconstructed FE distribution to set the a Variable's value at the finite element faces.

FXValuePenaltyBC This class is derived from FunctionPenaltyDirichletBC. Rather than enforcing a boundary condition, it 'encourages' matching values by penalizing differences between the Variable field and to the FE reconstruction.

FXFluxBC This class is derived from FunctionNeumannBC. Similar to FEValuePenaltyBC, it likewise 'encourages' a flux boundary condition by penalizing differences in the solution gradient compared to the FE reconstruction.

6.1.4 Mutable Coefficients Interface

MutableCoefficientsInterface was created to provide a universally-applicable interface for working with vectors of non-constant values, but especially FE coefficients. It contains methods for getting, setting, and transferring vectors of coefficients. It also provides functionality for a compatibility check with another MutableCoefficientsInterface instance. This checking is performed by comparing: 1) the sizes of the coefficient vectors, and 2) the contents of a characteristics vector. Currently, the characteristics array is a simple list of the orders in each dimension. Both the FXIntegralBaseUserObject and FunctionSeries classes inherit from this interface, enabling them and any derived classes to be used with MultiAppFXTransfer.

6.1.5 Coefficients Transfer

The FE coefficients transfer class, `MultiAppFXTransfer`, provides a universal mechanism to copy a vector of coefficients between compatible objects. `MutableCoefficientsInterface`-based classes are inherently compatible with this transfer method. It is not necessary to specify the object type in the MOOSE input file, only the object name. At runtime each `MultiAppFXTransfer` instance automatically searches the problem description of its parent and the child MultiApps for the named objects. Matching characteristics are also ensured once the objects are located. Finally, the transfer ensures that all MPI processes and OMP threads receive the required information among the parallelized subdomains.

6.2 Results

Test scenarios were created to demonstrate the fidelity and accuracy of multiphysics FE coupling methods within MOOSE. Both volumetric and boundary FEs were tested by comparing the results with existing mesh-based MOOSE data transfer methods.

6.2.1 Volumetric FEs

A simulation was contrived that emulated the transient thermal feedback behavior of a simplified graphite fuel element from the TREAT at the INL [92]. The simulation included a solution of two coupled physics fields, temperature and heat generation. The transient duration was 20 s. No boundary conditions for heat loss were applied due to the simulation's short timeframe, a reasonable assumption given TREAT's air-cooled design.

The temperature field was solved using the already available `HeatConduction` and `HeatConductionTimeDerivative` classes. Both are `Kernels` provided by the `heat_conduction` module to the MOOSE framework. Thermal material properties, required for the temperature solution, were inserted using the `GenericConstantMaterial` action.

The heat generation calculations were provided by `TREATHeat`, a MOOSE `Kernel` written specifically for this test. An equivalent `TREATHeatAux` was created for use with MOOSE's `AuxVariable-AuxKernel` system. These used an empirical calculation for heat generation that

was inversely related to temperature for an approximation of actual feedback effects in TREAT. Further, the heat generation profile was shaped by: 1) decreasing the power at the edges, and 2) radially shifting the peak power toward the $-y$ and $-z$ edge. These two adjustments applied a simplified estimation of the expected neutron flux profile in a fuel element: the first due to leakage out of the top and bottom of the core, while the second due the element's position in a radial-shaped core that has a higher flux toward the center. Another important reason for not excluding these effects—especially the non-symmetrical second adjustment—was to provide some spatially-varying features that high-fidelity multiphysics simulations are expected to capture.

Temporal behaviors were also part of the heat generation solution. This includes a linear ramp-up to full power during the first 3.3 s, then a power quench at 10.0 s effected by multiplying the empirical calculation result by 1×10^{-4}. Both the reaction ramp-up and quenching behaviors were intended to represent the effects of control rods being respectively withdrawn from and inserted into the core. Between 3.3 s to 10.0 s the temperature-driven feedback was allowed to independently drive the simulation. Finally, after the quench, the temperatures were allowed to equilibrate for the remaining 10 s of the transient simulation.

This simulation setup provided an excellent setup for testing the FE-based coupling methodology. First, it contained both symmetric and asymmetric features. Second, it required representation of field values over a wide range. Third, it demonstrated the efficacy of using FE methodologies in a tightly-coupled simulation. Finally, it provided a scenario for straightforward comparison to other coupling methodologies.

6.2.1.1 Cartesian

The model was 120 cm × 10 cm × 10 cm, representative of a basic fuel element used in the TREAT facility at the INL. The standard mesh was respectively divided into 100, 25, and 25 elements, for a total of 62 500 elements; the default 3D hexahedral finite element shape was used, each containing 8 quadrature points.[1] This model was created using the

[1]The `GeneratedMesh` action does have the ability generate meshes with different element shapes via the `elem_type` parameter. However, for much of this research this parameter was unspecified, thus MOOSE defaulted to the linear shape which best corresponded to the problem dimensionality.

Table 6.1 – Graphite thermal material properties used by the temperature solution for the Cartesian volumetric coupling tests.

Thermal Conductivity	Specific Heat	Density
$2.475\,\mathrm{W\,cm^{-1}\,K^{-1}}$	$0.800\,\mathrm{J\,g^{-1}\,K^{-1}}$	$1.8\,\mathrm{g\,cm^{-3}}$

`GeneratedMesh` action of the MOOSE framework. The thermal material properties used in the simulation are shown in table 6.1.

First, an ideal solution was generated by a fully-coupled simulation that solved both temperature and heat generation in the same MOOSE MultiApp. The `TREATHeat Kernel` included the heat generation as part of the temperature field calculations. This provided the best-possible solution and represents the 'gold-standard' when comparing the accuracy of the remaining tightly-coupled dual-MultiApp simulations (described later). A plot of the average and peak temperatures from the fully-coupled simulation are shown in fig. 6.2. The effects of the linear ramp-up are especially noticeable by the corresponding linear increase in temperatures during 0.5 s to 3.0 s, as well as the quench at 10 s. The feedback between temperature and heat generation is also shown by the concave portion of the curves between 4 s to 10 s. Finally, the average temperature is constant throughout the equilibration phase while the peak temperature asymptotically approaches the average temperature.

Four different tight coupling methodologies were compared to the ideal fully-coupled case: Direct, Interpolation, Layered, and FE. Each was tested in a simulation built from two MultiApps. Picard iterations were used to converge these dual-MultiApp solutions. The first MultiApp solved the temperature field problem, and was also the driver for the coupled simulation. The second MultiApp calculated the heat generation profile using `TREATHeatAux`. Transitioning to the `AuxVariable-AuxKernel` system in the second MultiApp was required because the empirical heat generation calculation was not a differential equation, thus could be performed in one pass. The main `Variable-Kernel` system is designed to solve differential and/or complex relations, and will actually fail the convergence check if the field is solved in the first nonlinear step and remains constant throughout the nested linear passes. The fully-coupled MultiApp was able to use the `TREATHeat Kernel` because it included other `Kernels` that solved the differential heat transfer equations for the temperature field, whereas

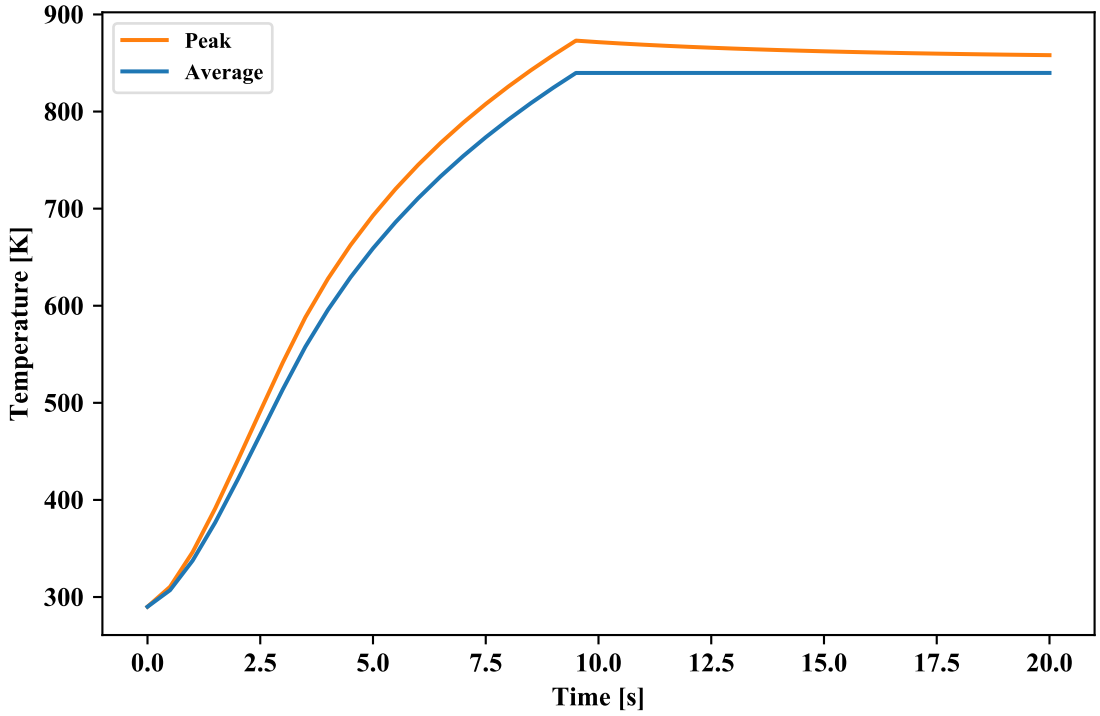


Figure 6.2 – Average and peak temperature data for the ideally-coupled simulation of the Cartesian volume test.

the tightly-coupled dual-MultiApp simulations calculated only the heat generation in the second MultiApp.

The Direct methodology used `MultiAppCopyTransfer` to copy the variable field values between the meshes in each MultiApp, the closest possible equivalent to the fully-coupled solution in a tightly-coupled simulation. Data was passed at the quadrature-point resolution, for a total of 500 000 floating point values (62 500 elements with 8 quadrature points each) per transfer. This methodology also served as a test platform to ensure that `TREATHeatAux` was functioning equivalently to `TREATHeat`.

The Interpolation methodology used the `MultiAppInterpolationTransfer` mechanism. It behaves like the Direct methodology if both MultiApp meshes are the same. Otherwise, it interpolates the required local field value from the closes quadrature points in the other MultiApp's mesh.

The Layered methodology is based on the `LayeredAverage` `UserObject` in MOOSE. It is one of the standard coupling approaches used in many multiphysics simulations, in which the solution is averaged over a user-specified number of discrete blocks or layers along a chosen axial orientation. Ten layers in the x direction were used to represent both the temperature and heat generation solutions. It utilized the transfer `MultiAppUserObjectTransfer` to send the field representations between the temperature and heat generation apps. Although this approach has the capability of providing excellent axial resolution, it also unfortunately eliminates any information in the axially-orthogonal directions and is non-optimal for many high-fidelity simulations.

The final methodology utilizes the FE classes and methods described in section 6.1, and is the actual focus of this testing suite. An `FXVolumeUserObject` object was used to generate an FE from the temperature field solution in the first MultiApp. This FE was transfered to the second MultiApp using `MultiAppFXTransfer`, and then expanded into the heat generation solution using a `FunctionSeriesToAux` object. The heat generation field was evaluated, converted into an FE, transferred back to the primary MultiApp, and then inserted into the solution using the recommended `BodyForce-FunctionSeries` approach. An identically-defined `FunctionSeries` object was used in each MultiApp for both generation and expansion calculations. Further, two different FE grades were tested—coarse and fine—to provide a small parametric study on the benefits of using higher-order FE series to reduce truncation error. The coarse FE was built using a 5×3×3 ordered Cartesian functional series, while the fine FE was built from an 11×5×5 ordered series.

Initially, these tests were performed with uniform element sizes and identical meshes in the coupled MultiApps. Additional simulations were then begun to test the adaptability of FEs and compare them with the other transfer methods already available in MOOSE. To accomplish this, the mesh in the heat-generation MultiApp was changed in the all dimensions, meaning that few (if any) of the elements had a matching element in the temperature-solver MultiApp. Again, the simulation outcomes were compared against the fully-coupled simulation; very minor differences were expected due to the change in the numerical solution of the heat-generation MultiApp. Both test types were designed to

simultaneously: 1) check the volumetric FE implementation, and 2) demonstrate their utility and efficacy in multiphysics coupling simulations.

The comparison results are compiled in table 6.2, highlighting the maximum relative temperature differences with respect to the idealized fully-coupled case. Both the identical and the nonidentical mesh coupling results are shown together. The 'Max. Overall' column shows the maximum temperature difference for all quadrature points over the entire transient simulation. The other two columns represent integral quantities. The 'Averaged' column uses data computed by the `ElementAverageValue` UserObject, which can be used in MOOSE to calculate the average value of a variable field at each timestep. The 'Peak' column uses data computed by the `ElementExtremeValue` UserObject, which can be used in MOOSE to identify the highest value of a variable field at each timestep. All the relative difference data was reported by the `exodiff` comparison utility that is compiled alongside MOOSE.

Identical Meshes These test are performed with identical meshes in both the temperature and heat generation solver MultiApps. All are compared against the ideal fully-coupled case. The relative difference data are shown in table 6.2 as the first row of each methodology. For the most part, each methodology was quite accurate. As anticipated, the Direct and Interpolation results were identical, and generally had the smallest relative differences.

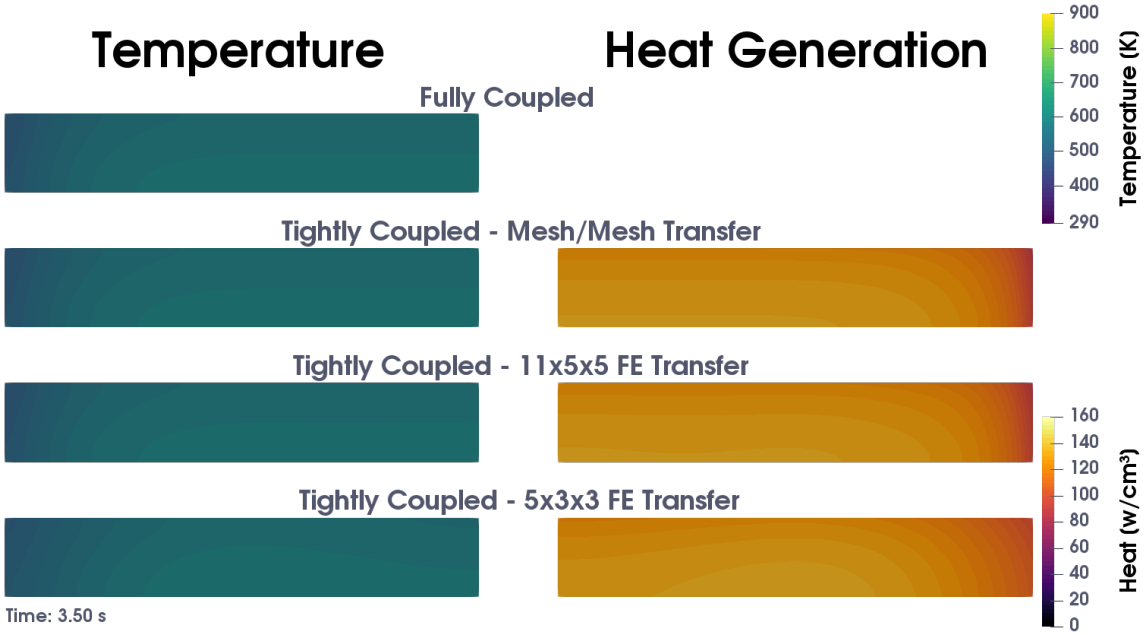
The fine FE results were in very close agreement to the fully-coupled case, still at or below 0.1 % differences. As expected, the truncation error of coarse FE is more noticeable compared to the fine FET. Yet, the results are still quite good; the highest relative difference is the maximum overall temperature at a 2.26 % difference, but the other two values are within acceptable levels. Interestingly, both FE methodologies have average temperature differences that are smaller than all the other methodologies. This is likely a direct result of the FE ability to dampen noise, a by-product of truncation error from capturing a finite number of corresponding moments from the variable fields. Lastly, the commonly-used Layered methodology, while still boasting an impressive average temperature result, has 9.4 % and 3.3 % differences for the maximum overall and peak temperature results, respectively. As

Table 6.2 – Relative temperature differences between the various Cartesian MultiApp volumetric coupling methodologies and the fully-coupled simulation.

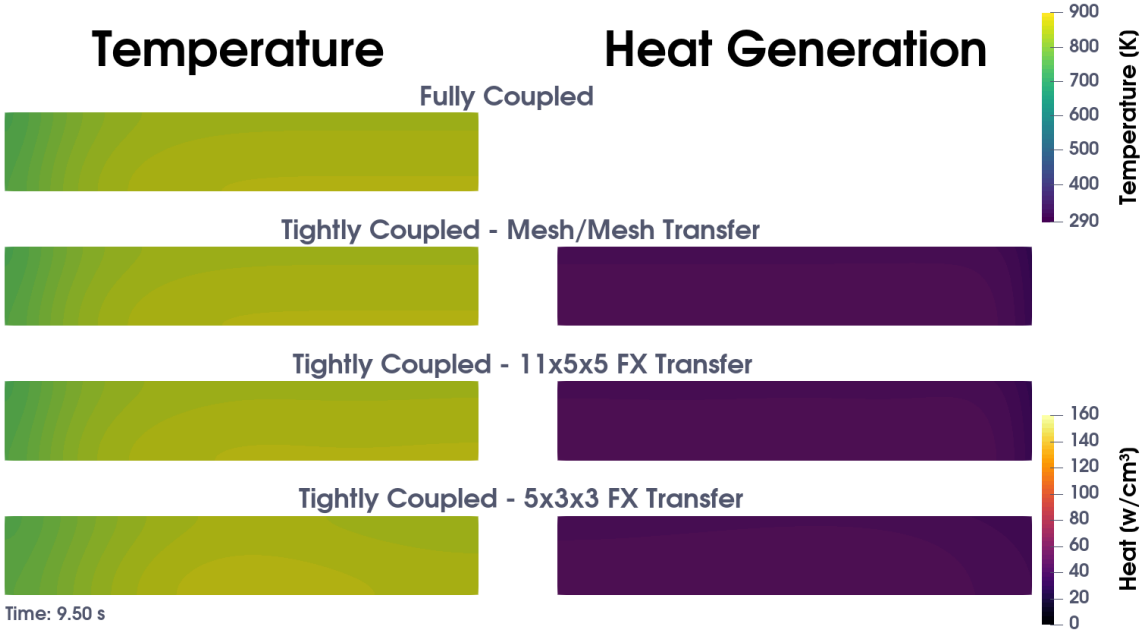
	Mesh Variations				
Methodology	**More**	**Skew**	**Max. Overall**	**Averaged**	**Peak**
Direct			4.426×10^{-4}	7.082×10^{-5}	6.388×10^{-5}
	✓			Failed	
		✓	8.076×10^{-2}	1.971×10^{-3}	1.150×10^{-2}
	✓	✓		Failed	
Interpolation			4.426×10^{-4}	7.082×10^{-5}	6.388×10^{-5}
	✓		4.306×10^{-3}	4.304×10^{-4}	2.630×10^{-5}
		✓	1.070×10^{-2}	2.620×10^{-4}	4.107×10^{-5}
	✓	✓	1.108×10^{-2}	3.965×10^{-4}	6.628×10^{-5}
Layered			9.420×10^{-2}	9.311×10^{-4}	3.335×10^{-2}
	✓		9.450×10^{-2}	9.834×10^{-4}	3.334×10^{-2}
		✓	9.314×10^{-2}	6.001×10^{-4}	3.333×10^{-2}
	✓	✓	9.285×10^{-2}	3.939×10^{-4}	3.333×10^{-2}
5×3×3 FE			2.261×10^{-2}	2.462×10^{-5}	7.337×10^{-3}
	✓		2.266×10^{-2}	1.780×10^{-5}	7.340×10^{-3}
		✓	2.300×10^{-2}	1.229×10^{-4}	7.433×10^{-3}
	✓	✓	2.301×10^{-2}	1.147×10^{-4}	7.433×10^{-3}
11×5×5 FE			1.016×10^{-3}	5.849×10^{-5}	4.718×10^{-4}
	✓		9.641×10^{-4}	4.851×10^{-5}	4.775×10^{-4}
		✓	3.031×10^{-3}	1.837×10^{-4}	4.695×10^{-4}
	✓	✓	2.959×10^{-3}	1.742×10^{-4}	4.727×10^{-4}

a side note, this is an indication of the risk that is involved when using a Layered-type approach, and the 10 layers used herein is more that the typical 3 to 5 layers.

Some snapshots of the simulation states between a selected coupling methodologies is also shown in fig. 6.3 at 3.5 s and 9.5 s. The results are spatially separated to visualize the variable profiles of temperature and heat generation. No heat generation profile is available for the fully-coupled test because, as stated previously, the physics were already integral to the temperature solution. All the results are remarkably similar. Slight undulations are detectable in the coarse FE, a result of the truncation errors mentioned previously; the characteristic is significantly reduced in the fine FE representation. Nevertheless, the results of the FE-based coupling methodologies are very satisfactory, while simultaneously demonstrating that the algorithms have been correctly implemented.



(a) Comparison at 3.5 s: peak heat generation



(b) Comparison at 9.5 s: near-peak temperature before quenching

Figure 6.3 – Comparison of volumetric coupling test results between differing methodologies. Shown are 2D axial cross sections along the fuel element center, divided symmetrically to illustrate the relationship between temperature and heat generation.

Nonidentical Meshes The temperature mesh was left identical to the previous set of simulations so that it could still provide the basis for comparison with the original full coupling simulation. The differences were achieved by perturbing the mesh generation in the heat generation solver MultiApp. Two types of mesh division variations were evaluated, simply termed 'Skew' and 'More' herein. The 'Skew' tests have a non-uniformity in the mesh elements effected via setting values for `x_bias`, `y_bias`, and `z_bias`—all parameters of the `GeneratedMesh` action—to 1.025, 1.05, and 1.1, respectively. The 'More' tests were generated by increasing the number of elements in the x, y, and z directions to 107, 29, and 31, respectively. The number of mesh elements were increased, rather than decreased, to preserve the numerical fidelity of the heat generation solver. Care was taken to not increase the number of elements by large steps because too fine a mesh has the potential to destabilize or impede the numerical solution.

First, the Direct tests with the 'More' variation actually failed to run because the number of elements in each MultiApp was different. Even further, although the 'Skew' variation ran, it produced very skewed results. The number of elements and element identifications (IDs) were the same, but the skewness of the meshes caused the equivalent elements of each mesh to occupy separate physical spaces. This provides a cautionary message against using this type of transfer when the two involved meshes are not explicitly identical.

Second, the Interpolation tests results were most impacted by the 'Skew'-type variation. Differences of 1 % were evident in the maximum overall temperature analysis. Conversely, the average and peak temperature results were very similar to the identical mesh test case.

Third, the Layered test results were essentially unchanged. This is expected since the layering works at a much coarser granularity than the mesh elements in both MultiApps. However, it is still the most undesirable performer of the tests.

Fourth, the FE-based coupling results for the maximum overall and peak temperature differences are all remarkably similar to the identical mesh results. Strangely, the average temperature differences did increase; however, they were still better than any of the other methodologies' average temperature difference results. This is a strong confirmation of the adaptability of FE-based methodologies to non-identical mesh assemblies.

Table 6.3 – UO_2 thermal material properties used by the temperature solution for the cylindrical volumetric coupling tests.

Thermal Conductivity	Specific Heat	Density
$0.053\,W\,cm^{-1}\,K^{-1}$	$0.233\,J\,g^{-1}\,K^{-1}$	$10.5\,g\,cm^{-3}$

6.2.1.2 Cylindrical

The cylindrical test model was not constructed to represent any physical system, but rather used to demonstrate: 1) the finite element-type agnosticity of FES, 2) the correctness of the Zernike implementation, and 3) the adaptability of FES when mesh-refinement methods are used. The actual model already exists within the MOOSE framework examples as `cyl-tet.e`. This mesh file stores a cylinder composed of tetrahedral elements, each containing 4 quadrature points, in an unstructured grid. The base model contained 4193 elements, was 5 cm tall, and had a radius of 1 cm.

A level 1 uniform mesh refinement was used for most of the cylindrical tests This increased the total element count to 33 544. Using this existing mesh provides the foundation needed to test the finite element-type agnosticity of FE methodologies: the tetrahedral element shape is very different from the hexahedral elements used in section 6.2.1.1. A successful demonstration will occur if the existing algorithms tested in section 6.2.1.1 can work properly, without further modification or need to specify the element shape, on a mesh using a different element shape.

Despite being very different dimensionally, some analogues to a fuel pin were used since the model is cylindrical. The thermal material properties used in the simulation are shown in table 6.3.

Again, an ideal solution was generated using a fully-coupled simulation that solved both temperature and heat generation in the same MOOSE MultiApp. A level 2 uniform mesh refinement was used for this ideal simulation, which increased the element count to 2 146 816. The physics of this simulation were similar to those used in the Cartesian test case: multiphysics feedback between temperature and heat generation. The corresponding heat generation classes created were `FuelPinHeat` and `FuelPinHeatAux`. Similar shaping was used, but the edge-oriented center-of-power offset was replaced by a radially-centered distribution

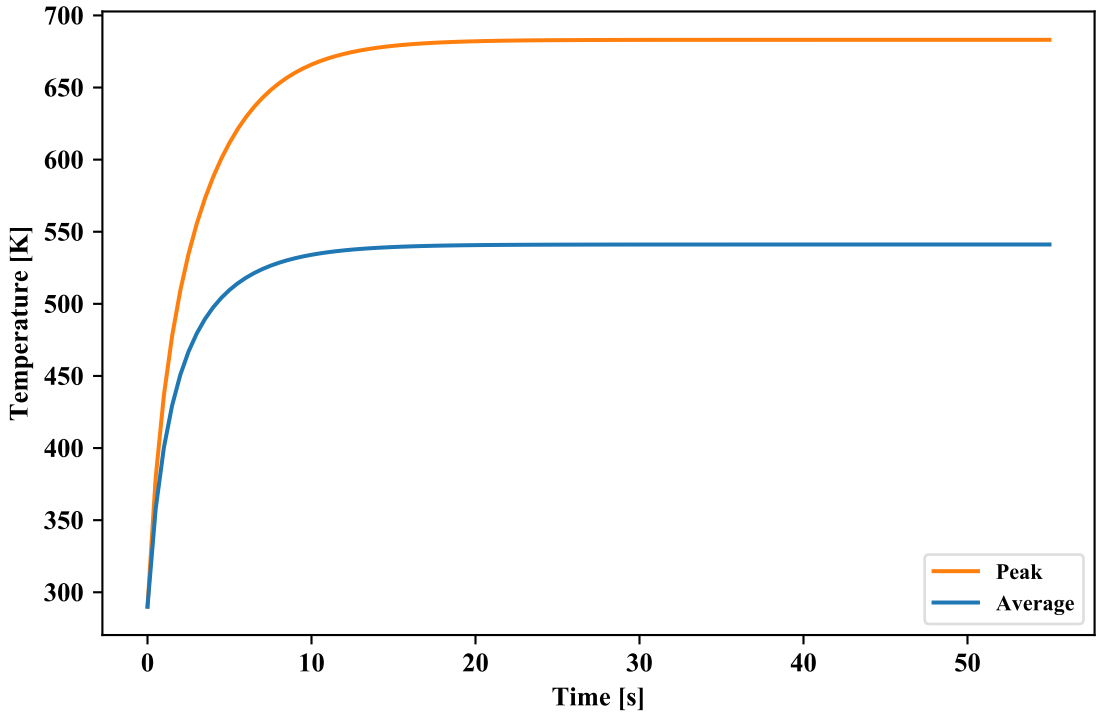


Figure 6.4 – Average and peak temperature data for the ideally-coupled simulation of the cylindrical volume test.

that could be shifted anywhere throughout the volume; for these tests the center of power was located at $(0.15, 0.15, -0.2)$ cm. The radial shift in the center-of-power provides a good asymmetric test of the Zernike implementation, as well as the `CylindricalDuo` composite series class.

No power adjustments were enforced; instead, a boundary condition was applied to the outside to represent a heat flux across the 'pin' edges. The `CoupledConvectiveFlux` class from the `heat_conduction` module was used, with the far fluid temperature $T_\infty = 290\,\mathrm{K}$ and heat transfer coefficient $h = 0.8\,\mathrm{W\,cm^{-2}\,K^{-1}}$. No boundary conditions were applied at the top or bottom edges. These boundary conditions also provide a great test of the Zernike polynomials by introducing a strong radially-varying effect. Finally, steady-state detection was enabled in the `Executioner` action, which caused the simulation to stop once the correct conditions were met. Figure 6.4 shows the profiles of the average and peak temperature data as a function of time.

Additionally, a different way of representing the field data was prototyped and tested for use during coefficient generation and expansion. In particular, it was observed in running the standard tests (described in section 6.2.1.2) that the physics of this experiment created a very steep gradient at the top and circumference of the model. This was caused by both the flux boundary condition and the center of the heat generation. As mentioned before, large gradients can be a challenge to FE-based methods. Although in some cases this can be managed through careful selection of the underlying functional bases, another approach to mitigate this deficiency is to transform the data using exponential powers. Given an exponential value f, the modified form of eq. (3.14) to use when generating the coefficients is:

$$b_i = \int_{x_1}^{x_2} \widetilde{\psi}_i(x)\rho_\psi(x)\left(F(x)\right)^f \, dx \tag{6.1}$$

and the modified form of eq. (3.15) to use when expanding the distribution is:

$$F(x) = \left(\sum_{i=0}^{\infty} b_i\psi_i(x)\right)^{\frac{1}{f}} \tag{6.2}$$

This approach was tested and demonstrated in section 6.2.1.2. One main caveat exists: this approach is ill-suited for distributions that contain negative numbers unless f or its inverse is an odd integer.

Standard Coupling A number of tightly-coupled tests were evaluated. Steady-state detection was replaced by a transient duration of 55 s, the actual time at which the ideal fully-coupled simulation reached steady-state conditions and stopped. The only non-FE type coupling performed was a single Direct coupling simulation to verify the proper functionality of `FuelPinHeatAux`. Four FE coupling order granularities were tested (using the format Legendre×Zernike): 5×3, 7×5, 7×7, and 11×7. Additional simulations were also performed using MOOSE's internal mesh-refinement capability for the temperature MultiApp's mesh. This allowed MOOSE to dynamically modify the mesh after each time step based on the comparative magnitude of the field's gradient change within each element; a high value results in mesh refinement by splitting the element evenly into smaller elements,

Table 6.4 – Relative temperature differences between the cylindrical MultiApp volumetric coupling methodology and the fully-coupled simulation.

Methodology	Mesh Refinement	Averaged	Peak
Direct		1.616×10^{-3}	2.565×10^{-3}
5×3 FE		3.135×10^{-3}	3.516×10^{-2}
	✓	1.500×10^{-3}	3.931×10^{-2}
7×5 FE		5.290×10^{-3}	1.136×10^{-2}
	✓	2.101×10^{-3}	1.296×10^{-2}
7×7 FE		1.010×10^{-2}	5.658×10^{-3}
	✓	2.670×10^{-3}	5.974×10^{-3}
11×7 FE		1.011×10^{-2}	5.734×10^{-3}
	✓	2.655×10^{-3}	6.752×10^{-3}

while a low proportionality may result in a group of elements being merged together to create a single larger element. Such a refinement methodology is incompatible with many coupling methodologies that depend on identical or near-identical meshes. The heat generation MultiApp's mesh was not dynamically refined, but both the temperature and heat generation meshes started with a uniform refinement of 2.

The comparison results are compiled in table 6.4. The `exodiff` tool could not be used since the various refinement levels caused too great a difference between the different tests' meshes. Instead, the average and peak temperature calculations were dumped into a csv file and analyzed using a Python script for comparison to the fully-coupled results.

The results of the Direct test, with both values less than 1 %, are small enough to demonstrate that the actions of `FuelPinHeat` and `FuelPinHeatAux` are indistinguishable within the tolerance of a coupled multiphysics simulation. Again, like in section 6.2.1.1, the average temperature results of the lower-ordered fes are slightly better than the higher-ordered tests. The strong radially-varying behavior, introduced by the boundary conditions, is demonstrated by the reduced difference of the peak temperature scores with the higher-ordered Zernike polynomial tests. This is demonstrated even further by the essentially identical results for the 7×7 and 11×7 tests, in which no added benefit was obtained when increasing the order of the axial Legendre series. Mesh refinement did serve to improve the

simulation fidelity, in some cases bringing the results very close to those of the Direct test case.

Flattened Distribution At first, the relatively high near-1 % results of section 6.2.1.2 were thought to be either a result of numerical integration error when generating the coefficients. This is essentially caused when the quadrature is not fine enough to capture the FE moments of the variable field. Another possibility was that the data required a higher-ordered FE to provide all the information. Tests were performed to check each theory, but increasing neither the mesh refinement level nor the FE order improved the results. Since they were not significantly better, the tests using mesh refinement corroborated the evidence that numerical integration error was not fully to blame. It was then realized, through visual inspection, that the heat field was strongly-varying at the top and circumference of the model. These effects were a natural cause of the physics used in the simulation.

Equations (6.1) and (6.2) were implemented respectively into `FXIntegralBaseUserObject` and `FunctionSeries`. Preliminary testing revealed that an exponential value f of 0.5 was optimal for transforming the data in this simulation, providing a good balance between allowing the FE to capture the highly-varying regions while still retaining fidelity in the slowly-varying areas. The optimal order number for each FE-based series was searched for via parametric testing. A 7×7 functional series was found to be optimal when using temperature as the criterion, and a 13×7 functional series was optimal when considering the heat generation. Both of these ordered sets, along with a 5×3 for completeness, were tested using the exponentially transformed field distribution. Uniform mesh refinements of 1 and 2 were used for the temperature and heat generation MultiApps, respectively.

Figures 6.5 and 6.6 show the results of the test runs compared to the Direct test case from section 6.2.1.2. The Direct case, not the fully-coupled case, was used because the heat generation field was separately available for comparison due to the requirements of the dual-MultiApp simulation. It also provided an analogue because of the shared dual-MultiApp Picard iteration-converged simulation approach. Interestingly, the boundary condition induced a much lower temperatures at the outside edges. This is especially

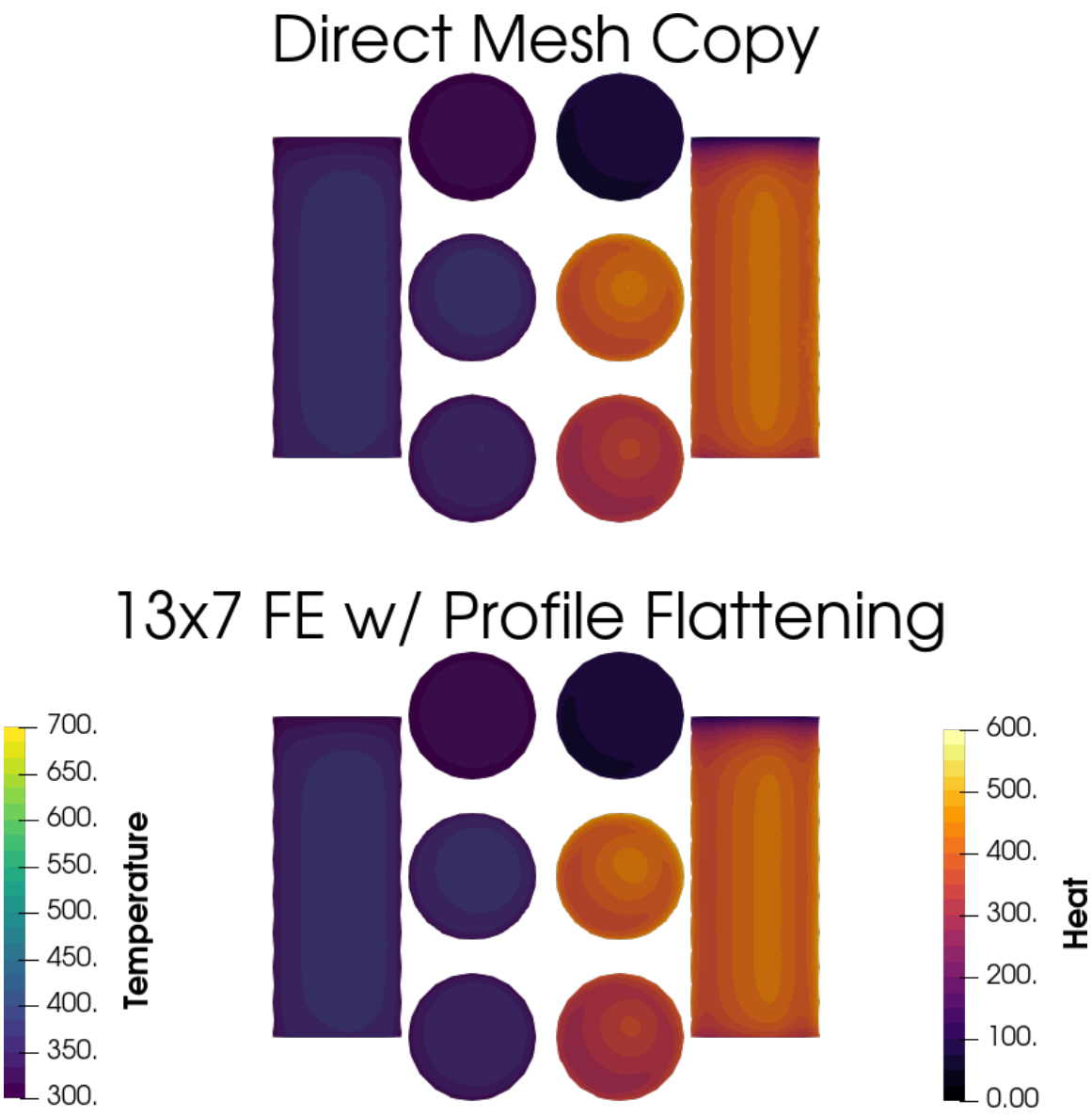


Figure 6.5 – Visual comparison between the Direct and exponentially-transformed-data FE coupling results, at the time of highest heat generation. On the outside edges are 2D axial slices of the field profiles. In the center are 2D radial slices of the field profiles at the bottom, middle, and top of the axial lengths.

noticeable in fig. 6.6, which also caused a relatively high localized heat generation due to the empirical temperature feedback model used. The field profile of the flattened FE-based coupling agrees very well with Direct simulation results.

Table 6.5 provides the calculated results of these tests. The non-flattened tests results from table 6.4 for the Direct, 5×3, and 7×7 are also included for comparison. Both demonstrate that

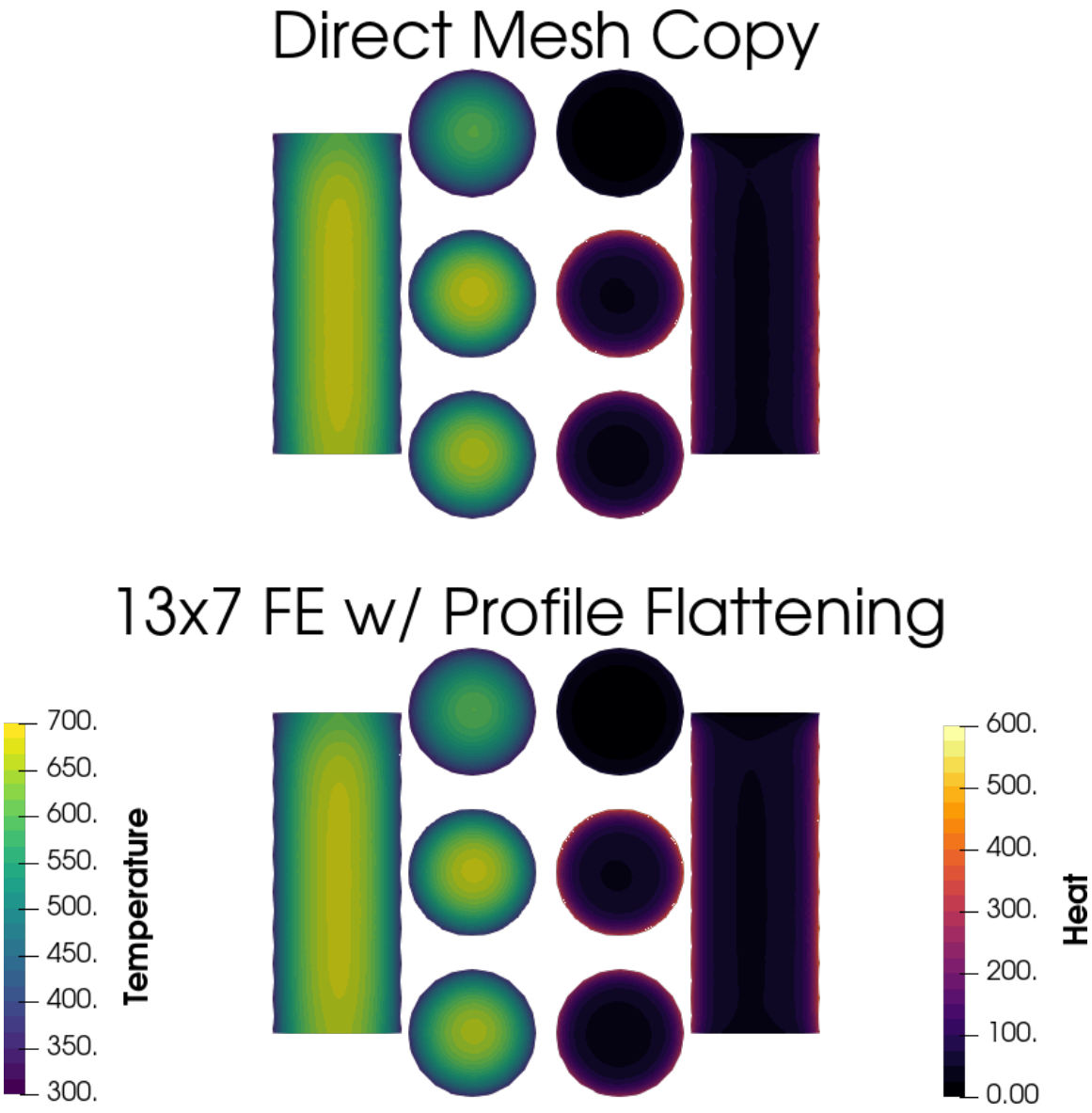


Figure 6.6 – Visual comparison between the Direct and exponentially-transformed-data FE coupling results, at a time when the system is near steady-state. On the outside edges are 2D axial slices of the field profiles. In the center are 2D radial slices of the field profiles at the bottom, middle, and top of the axial lengths.

Table 6.5 – Relative temperature differences between the exponentially-transformed cylindrical MultiApp volumetric coupling methodology and the fully-coupled simulation. The non-exponentially-transformed results are included here from table 6.4 for easier comparison.

Methodology	Exponentially-transformed	Averaged	Peak
Direct		1.616×10^{-3}	2.565×10^{-3}
5×3 FE		3.135×10^{-3}	3.516×10^{-2}
	✓	9.463×10^{-4}	1.800×10^{-2}
7×7 FE		1.010×10^{-2}	5.658×10^{-3}
	✓	1.398×10^{-3}	1.234×10^{-3}
13×7 FE	✓	1.339×10^{-3}	3.452×10^{-3}

the use of an exponential transform to 'flatten' the underlying distribution is one possible solution when using FEs to represent widely-varying data. The improvement between the standard and exponentially-transformed coupling results is especially compelling. In fact, the relative differences are very similar to the Cartesian FE results presented in table 6.2

Meeting of Cylindrical Test Objectives These results confirm the three main objectives of the cylindrical FE methodology test cases.

First, the FE-based methodologies were demonstrated to be agnostic to the finite element shape used, requiring no extra configuration or specialization whatsoever. By inheriting from existing UserObject classes in the MOOSE framework, the numerical integration routines are performed correctly.

Second, the Zernike polynomials were correctly implemented. This was demonstrated by the similitude of the simulation results to the fully-coupled, and by the fact that the FE-coupled simulation converged. It can also be seen in figs. 6.5 and 6.6, in which the temperature and heat generation profiles of the Direct and FE methodologies match; improperly-implemented Zernike polynomial algorithms would exhibit either an angular rotation and/or an unexpected distribution shape.

Third, dynamic mesh adaptivity did not confuse nor compromise the behavior of the FE-coupled MultiApps. Actually, in some cases it served to improve the simulation fidelity through the action of refining the mesh for improved finite element analysis, as well as by refining the numerical integration quadrature for a higher-quality FE coefficient generation.

With all three objects met, the cylindrical volume FE-based coupling tests were determined to be successful.

6.2.2 Boundary FEs

A simulation was contrived to simulate, in a 2D representation, the 1D heat flow from a hot uranium fuel segment into pressurized water in order to test the boundary condition FEs. Cladding was ignored for simplicity; the relatively high thermal conductivity of typical cladding materials renders them effectively invisible to heat transport problems. The model size was 2.4 cm × 10 cm, respectively divided into 36 and 80 hexahedral elements, again constructed using the `GeneratedMesh` MOOSE action. The fuel region was in the left 0.4 cm of the model, and water in the right 2.0 cm.

The entire model was initialized to a uniform temperature of 290 K. Heat was generated in the fuel region with a rate of 600 W cm^{-2} at the center, which was decreased toward the interface, top, and bottom. The heat was then conducted throughout the fuel, across the interface, and into the water. The boundary conditions were set as: 1) a zero flux on the fuel block's left side, 2) a matching temperature and heat flux at the interface, and 3) a temperature that ranged linearly from 290 °C to 390 °C at the water block's far right side. The simulation was run for 100 s to reach a quasi steady-state condition.

MOOSE's built-in `MeshModifier` capability was used to create a fully-coupled model by: 1) splitting the `GeneratedMesh` construction into two separate solution blocks, and 2) fabricating a named interface between the two blocks. A suitable `InterfaceKernel`, designed to couple the fluxes between two variables in opposite blocks at an interface, was found among the example source files in MOOSE and used for this test.

For the tightly-coupled tests, the model was again solved by two separate MultiApps. This time, instead of physically overlapping as in the volumetric tests, one MultiApp was created for each material region with its separate physical location. MOOSE does not have any other straightforward methods for transferring non-scalar boundary conditions between MultiApps. Thus, only FE coupling was compared against the fully-coupled reference solution.

Table 6.6 – Maximum relative differences at the fuel-water interface between the fully-coupled and FE-coupled simulations.

Test	Average Interface Temperature	Heat Transfer Rate
0th order FE	9.647×10^{-3}	5.391×10^{-2}
7th order FE	1.322×10^{-2}	2.096×10^{-2}

Two FE grades were tested, 0th and 7th order Legendre expansions. The water MultiApp acted as the MasterApp for the FE-based tightly-coupled simulations. The heat conduction was solved in the water, and the temperature at the interface was captured as an FE using `FXBoundaryValueUserObject`. The temperature FE was then transferred to the fuel MultiApp and inserted as a boundary condition using `FXValueBC`. Heat conduction and generation were then solved simultaneously within the temperature variable field, then the resulting heat flux at the interface was captured as an FE using `FXBoundaryFluxUserObject`. This FE was then transferred to the water MultiApp using `MultiAppFXTransfer`, where it was converted to a boundary condition using `FXFluxBC`. Picard iterations were again used to converge the solution.

A visual comparison of the fully-coupled and 7th-order FE-coupled simulations is shown in fig. 6.7 at 100 s. The results of the two methods are visually identical. Two integrated values were also generated in the fuel MultiApp: average interface temperature and total heat transfer rate. These values were calculated using the `SideAverageValue` and `SideFluxIntegral` classes in MOOSE, then saved to a csv file. The analysis results are provided in table 6.6. These were generated by comparing the contents of the csv files and searching for the timestep with the largest difference. The 7th-order FE methods perform very well in comparison to the reference solution, with relative errors of 1.3 % and 2.1 % for the average interface temperature and heat transfer rates, respectively. As in previous results, the lower 0th-ordered FE exhibits the lowest difference for the average temperature, but has a larger difference for the heat transfer rate. This is because the heat transfer rate is dependent on the spatially-varying temperatures at the fuel and water interface.

Additional FE-based coupling simulations (not shown here) were run to investigate possible causes of these differences, which are large compared against the volumetric

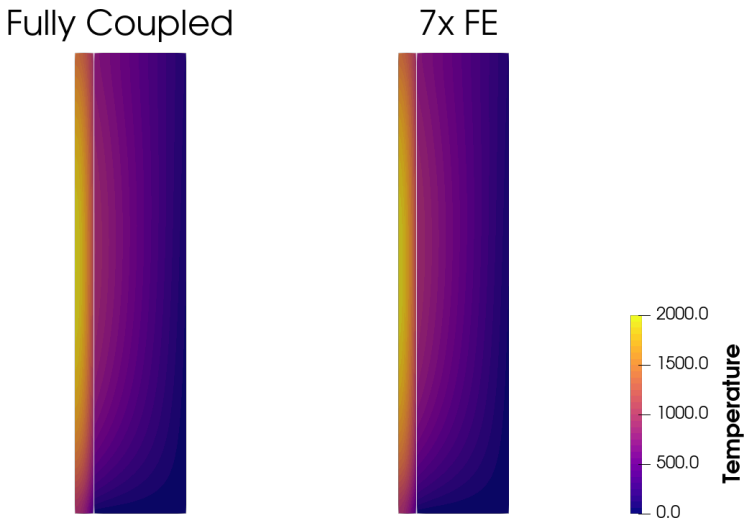


Figure 6.7 – Comparison of boundary coupling test results between differing methodologies. A thin white line marks the MultiApp interface between the fuel (left) and the water (right).

coupling results in section 6.2.1.1. Further investigation was performed by using varying combinations of increased FE order and mesh divisions in the 'y' direction. The FE-based coupling solution remained essentially unchanged, which indicated that the cause was not due to the FE-based data. Instead, it is suspected that these differences were caused by numerical discrepancies in the Picard iteration-based tightly-coupled approach. Unfortunately, lack of an equivalent MultiApp transfer methodology in MOOSE precludes any study to confirm this hypothesis.

However, and even with these differences, the outcome is a resoundingly positive demonstration of FE-based multiphysics boundary coupling. As mentioned in chapter 2, many boundary-coupled multiphysics simulations employ a simple average-based approach in which the spatial fidelity is lost. In such cases, the 0^{th}-order FE-coupled simulation provides a good approximation of the expected errors: 0.96 % and 5.4 % for the average interface temperature and heat transfer rates, respectively. For high-fidelity simulations, however, nonzero-ordered FEs are more desirable because they provide spatially-correlated data.

6.3 Conclusions

Fes are an excellent coupling solution because they are: 1) agnostic to the underlying mesh structures, 2) continuous in space, and 3) promising techniques for achieving low-data-density transfers with high accuracy. Further, they eliminate the need to implement mesh interpolation algorithms. This permits great flexibility in running uniphysicss codes on widely disparate mesh refinement levels. Fe-based coupling methods have been implemented within the MOOSE framework, allowing great flexibility for small-data high-fidelity transfers between MOOSE applications and external codes that support fe representations. Both volumetric- and boundary-type fes were developed, tested, and demonstrated to be effective.

Chapter 7

Coupling with Serpent under MOOSE

> *Any working composer or painter or sculptor will tell you that inspiration comes at the eighth hour of labour rather than as a bolt out of the blue. We have to get our vanities and our preconceptions out of the way and do the work in the time allotted.*
>
> — John Williams

A primary purpose of implementing FEs in MOOSE and FETs in Serpent was to study the possibility of using FE-based methods for multiphysics coupling between separate codes. With due diligence, such an outcome should actually be accomplished and demonstrated.

Previous studies in FE-based couplings have performed full 3D simulations but not utilized FE-based methods for all data transfers [119], or fully-utilized FE-based couplings but only for lower-dimensional simulations [83]. This was the first fully-3D complete FE-based coupling known of in published literature. It was also the first to use fully-convolved, i.e., non-separable or multivariate, functional expansions.

This was a culmination of the methods developed in chapters 4 to 6. Extensions of MOOSE classes, from both the core framework and the `functional_expansion_module`, were developed so that Serpent could be incorporated as a MOOSE-wrapped application.

Additional modifications of the build system were incorporated to simplify the compilation of Serpent as a shared library within MOOSE. These developments utilized MOOSE as a multiphysics driver for tightly-coupled simulations with Serpent.

7.1 Serpent Build System

In the MOOSE ecosystem, a `MooseApp` is used to implement the functionality of the MOOSE framework as a usable simulation application. Under this paradigm a new `MooseApp` was created. Following the convention encouraged by the core MOOSE developers of using animal-related names, this new `MooseApp` was called `chrysalis` to represent the anticipation that something beautiful and beneficial would emerge.

Before proceeding, it is imperative to recognize that Serpent is written in C while MOOSE is developed in C++. Although syntactically similar, this slight difference in languages immediately introduced some complexities into the build process. Thus, some modification of the Serpent source files was required for Serpent to be built as a MOOSE-wrapped application. Namely, there were a few declaration collisions that needed repairing. A couple conflicting compiler directives were also found. These were resolved via: 1) writing an automated `update_serpent.sh` script to refactor the Serpent source, and 2) modification of the standard `Makefile` that accompanied `chrysalis`.

The `Makefile` is what directs the build process when `chrysalis` is compiled. It was modified so that one of the first actions is to call the `update_serpent.sh` script. The script depends on the same environment variable as Serpent: `SERPENT_HOME`, which contains the directory path that houses the Serpent files. This script copies the required files to a local directory in order to prevent any unintended alteration of the original files. The required modification are also performed by the script once it has copied the files.

Conveniently, `serpent_update.sh` only copies/updates those files that are missing or out of date. Such information is returned back to the `Makefile`, which leverages that knowledge to optimize the build process. For example, if any header files are modified then the entire source tree need recompiled. Conversely, if nothing was updated then no further action is

required nor taken. The `Makefile` also specifies some customized building options, such as flagging whether or not link-time optimization (LTO) should be used.[1] Further, MOOSE build options for debugging, MPI, and OMP capabilities are automatically detected and integrated into the build options for Serpent. The Serpent source files are actually compiled within the MOOSE system to ensure full compatibility, then linked into the final `chrysalis` binary executable.

With these developments, compiling and linking Serpent into MOOSE is now a trivial process. Further, this build process will likely be compatible with future releases of both MOOSE and Serpent. Essentially, little or no modifications will be required to maintain compatibility with most future developments.

7.2 Serpent Multiphysics Interface

It may be recalled that the preliminary implementation of FETs into Serpent was performed using the existing multiphysics interface. The ensuing optimization studies and additional development work for inclusion into the globally-released Serpent codebase was performed within the detector framework (see chapter 4). Work on the fully-featured multiphysics interface implementation was begun only after the detector framework was satisfactorily completed and tested.

First, the necessary input options were added into the multiphysics interface parsing routines. For example, integers are used in Serpent files to specify multiphysics input types; 31 and 32 were added as values to represent density and temperature FE inputs, respectively. Additional routines were added for reading the `.ifc` interface input files containing the input functional expansion parameters and coefficients Although technically optional, the parameter specifications for the fission power density data FET were also supported; output routines were added for printing the FET's coefficients and other information to a text file.

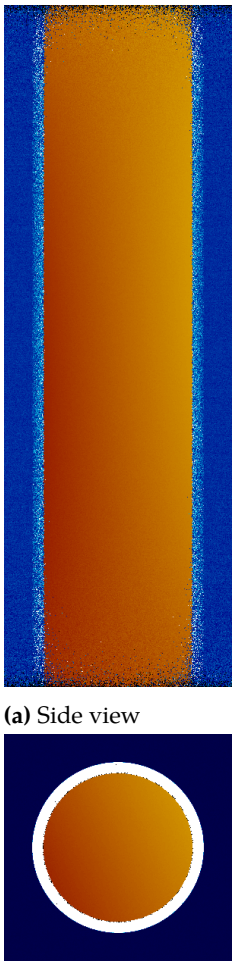
[1] LTO is a compilation process that reduces redundant instructions by checking was is not used during linking and stripping it out. It has been observed that using LTO has improved the computational efficiency of Serpent by 20 % to 40 % beyond standard compiler optimization options.

Second, the existing FE-oriented routines in the detector framework were updated and/or redesigned to be cross-compatible for use by the multiphysics routines. For example, a common data structure was defined that was capable of storing the description and parameters unique to each FE utilized in Serpent. This common structure supports both detector or multiphysics interface instances, regardless of whether the intended use is for input or output data. The scoring routine `ScoreFET()` was also generalized so that it could be used by both the FET detector and the multiphysics interface coefficient tally estimators. The underlying routines for the Legendre and Zernike polynomials were likewise reused.

Finally, these routines were tested and demonstrated to be functional. Although still under private development, it is anticipated that the code will be integrated into the mainstream release of Serpent in the near future.

Serpent's capability to plot multiphysics feedback data was used to test the implementation validity; an example of the output is shown in fig. 7.1. The plot was generated throughout a multiphysics simulation with a continuously-varying temperature profile provided via FES. Each pixel represents a collision event, and the color indicates the temperature that sampled from the FE. No output FET was requested, for the sole purpose was to visually inspect the output and compare the results to the expected shape of the distribution defined by the input FE coefficients. The influence of the FE-based temperature sampling is immediately evident, as demonstrated by the orange color variations going from dark to light while traversing from right to left.

Figure 7.1 is just one example, but in all the trials performed each plotted temperature profile matched the expected shape described by the input FE coefficients. Conveniently, this testing was simplified since the input coefficients were expected to be in orthonormalized form. In other words, using manually-set coefficients to test the implementation was straightforward because Serpent expects the orthonormalization constant to be already included; each coefficient value corresponds directly to its magnitude of influence relative to the zeroth-order coefficient value. This is a reconfirmation of the justification provided in section 3.2.2 for the distinct uses of orthonormalized and standardized variants input and output contexts.



(a) Side view

(b) Top view

Figure 7.1 – Plot of an FE-based continuously-varying temperature field in Serpent for a representative AP-1000 fuel pin. Note that the image scale of fig. 7.1a has been reduced in z.

7.3 MOOSE-Serpent Driver

Establishing MOOSE as a multiphysics driver for Serpent was split into two development activities. The first activity was focused on new local additions to the `functional_expansion_tools` module that will be generally applicable to FE-based methods. These modifications may later be integrated into MOOSE, although currently they exist in a separately forked repository. The second activity was centered on implementing any Serpent-specific requirements.

7.3.1 Modifications to the Functional Expansion Tools Module

As described in chapter 6, the foundational interface for managing coefficients in the `functional_expansion_tools` module is `MutableCoefficientsInterface`. In this context, the relevant aspects are the coefficient storage and transfer mechanisms. Thus, any MOOSE action seeking to leverage FE-based capabilities should inherit from this interface. Extending the existing FE transfer mechanism to work with external applications similarly requires use of this interface.

7.3.1.1 FXExecutioner

Creating a MOOSE-wrapped application required the development of a specialized MultiApp. The specifics of this will be addressed in section 7.3.2; however, for now it sufficient to highlight that the MOOSE `Executioner` action class is responsible for executing solve steps in each MultiApp. A generalized FE-enabled `Executioner` was created by inheriting from both `Transient` (a concrete implementation of the MOOSE `Executioner` action) and `MutableCoefficientsInterface`. The resulting class was called `FXExecutioner`, using 'FX' in keeping with the nomenclature suggested by the MOOSE development team.

Further, the FE-parameter specification code was refactored from `FunctionSeries` into a new base class `FXInputParameters`. Both classes need the ability to parse FE parameters in the MOOSE input file. Using this adaptation, both `FunctionSeries` and `FXExecutioner` could inherit from `FXInputParameters`. This provided the ability to ensure a common FE parameter specification while preventing code duplication.

Next, the transfer class `MultiAppFXTransfer` was expanded to scan MultiApps for `Executioner` instances. As part of enabling this functionality, each `FXExecutioner` is required to be given a unique name by the user; this name is used by `MultiAppFXTransfer` to locate the specified instance. Other than the additional scan functionality, the remainder of `MultiAppFXTransfer`'s behavior for interacting with `FXExecutioner` is identical to the already supported MOOSE action classes.

7.3.1.2 Relaxation

Relaxation approaches were then incorporated into `FXExecutioner`. The methods were assembled within `RelaxationBuffer`, a new data storage class. Three different options were provided in the initial implementation, and can be applied to both coefficients exported to an external code as well as those imported into MOOSE. First, the 'none' option is applied by default. As could be inferred, it leaves the coefficients unaltered. Second, the 'weighted' option uses a user-specified weight ρ to apply a contribution from the updated coefficients as:

$$\widehat{b}_{n,p} = \rho b_{n,p} + \left(1 - \rho\right) \widehat{b}_{n,p-1} \tag{7.1}$$

where $\widehat{b}_{n,p}$ is the relaxed coefficient for iteration p, $b_{n,p}$ is the newly-calculated coefficient, and $\widehat{b}_{n,p-1}$ is the previously-relaxed coefficient. Third, the 'windowed' option takes a user-specified width X and averages that many most recent coefficients together as:

$$\widehat{b}_{n,p} = \frac{\sum_{i=p-x}^{p} b_{n,i}}{x} \tag{7.2}$$

where x is the smallest of the window width X and the current number of iterations P. Additional approaches will be trivial to add in the future, as needed.

7.3.1.3 Fourier Series

A Fourier functional series was added to compliment the existing Legendre and Zernike implementations. As described in chapter 6, it must be a concrete implementation of `SingleSeriesBasisInterface`. It contains two parameterized forms that are applicable to the various use cases. The first variant can be used for a linear single series, much in the same way the Legendre polynomials depend on just a single dimension. The second variant is a radial form that is used to represent only composite angle-dependent data on a plane defined by two dimensions. On other words, the radius is ignored. The actual angle is calculated as the arctan from the coordinates in the plane.

This Fourier series was used to create `AxialShell`, an concrete implementation of `CompositeSeriesBasisInterface`, which is designed to work with 3D boundary data in

cylindrical systems. A radius-agnostic series was desired in order to be able to specify the FE parameters in terms of a mesh sideset.[2] Alternatively, a range could potentially be provided to specify where the FE is valid. Together, these two series additions were coded in a matter of hours, a demonstration of the ease an flexibility provided by the design architecture of the `functional_expansion_toosl` module.

7.3.1.4 Iteration Skipping

Finally, one more addition was performed that was immediately useful for reducing the overall simulation computation time. Namely, MC simulations are notoriously slow compared to numerical methods. One possible technique to employ when coupling a stochastic code (like Serpent) to a numerical solver (like MOOSE) is skipping some of the MC solves while performing Picard iterations to converge the solutions. Thus, a parameter was added to `FXExecutioner` that permits the skipping of solve steps for subtended MOOSE-wrapped applications. The first Picard iteration will always be evaluated by the wrapped application, but then it will sleep for the specified number of iterations. The cycle is repeated throughout the multiphysics convergence process. This allows a much faster convergence for the total problem, as the MC runs are performed less frequently but still can provide the required multiphysics feedback.

This feature may also prove interesting for future characterization and research into convergence studies.

[2] There are two approaches to using a FE-based boundary condition for a curved surface of a mesh. This is necessary because of the challenge of representing non-linear geometries with linear elements.

The first approach is to define a radius range over which the FE is valid. This would be required when the underlying functional basis depends on all three 3D r, θ, and z components. In this implementation, the radius r would be given a minimum and maximum value over which the FE is valid. A radius range would be necessary because the element sides are never created with infinite floating-point precision nor curved such that all quadrature points lie on a the surface of a cylinder. This works for the most part, but may require some *a priori* knowledge of the necessary tolerances in the model curvature; if the ratio of angular mesh divisions to the radius is low, then the tolerances must be larger. Such an approach may also be used when defining the operational parameters for an annular series.

The second approach, and that which is implemented in `AxialShell`, is to define a functional basis that is dependent on θ and z only. In MOOSE contexts, this means that the coordinates normal to the axial direction are converted into an angular value only. This provides infinite tolerance for boundary conditions on a mesh sideset, regardless of whether the functional expansion is being used to generate coefficients or expand the distribution. Conveniently, it also reduces the number of terms in the functional series be eliminating the radial terms—while maintaining the fidelity in θ and z—which acts to reduce the overall error. These errors could be either due to numerical integration discrepancies or statistical uncertainties.

7.3.2 Serpent-Specific Additions to Chrysalis

With `FXExecutioner` in place, the Serpent driver could be developed in `chrysalis` (the Serpent MOOSE-wrapped application described in section 7.1). A Serpent-specific executioner, `SerpentExecutioner`, was created along with a matching `SerpentTimeStepper`. `SerpentExecutioner` extends `FXExecutioner` by providing input requirements for Serpent. `SerpentTimeStepper` extends `TimeStepper` by making the appropriate method calls for the simulation initialization, execution, and finalization.

7.3.2.1 SerpentExecutioner

`SerpentExecutioner` is responsible for providing the high-level interface between MOOSE and Serpent. Serpent's multiphysics interface is file-bound, so `SerpentExecutioner` is responsible for generating the files that Serpent will require to participate in multiphysics simulations. It fulfills three primary roles.

First, it exposes and interprets the FE definition parameters. These are parsed by the inherited `FXInputParameters` class of `FXExecutioner`.

Second, it generates the requisite Serpent multiphysics interface file. To do this, it requires the path to a Serpent input file that defines the MC problem; the use of Serpent's '`include`' directive is still supported. `SerpentExecutioner` duplicates this input file, then appends the required options that engage with Serpent's multiphysics interface. These information fields are filled out by `SerpentExecutioner` using the parameters provided via `FXInputParameters` and coefficients stored in `MutableCoefficientsInterface`. Completing the interface file also requires names for the materials defined in Serpent; these are provided to `SerpentExecutioner` via additional mandatory input parameters.

Third, it provides options for normalizing the fission power density coefficient values. There are two normalization options available: 1) scale, which simply scales all the coefficients equally by a single value, and 2) level, which normalizes the coefficients such that the average of the expanded distribution will equal the user-specified value. Both

of these normalization types can have the value specified as a constant, or as a MOOSE `Function` with more complex behavior.

Subsequently, a simple coefficient filter was enabled based off the R_n factor as referenced by eq. (5.1). This is evaluated as the FET coefficients are being read in from the Serpent output. The filter value is currently hard-coded in with an acceptance relation of $R_n <= 1$; any higher-valued coefficients are zeroed out.

7.3.2.2 SerpentTimeStepper

`SerpentTimeStepper` is responsible for actually running Serpent using the shared library built into `chrysalis` as described in section 7.1. The calls to Serpent methods are split into three operational regimes: initialization, execution, and finalization. These were implemented using virtually overidden methods from the base class, each containing the program flow from Serpent's `main()`. The flow of `main()` was analyzed and split up according to its functional regime. Method calls, conditional checks, and assignment operations not relevant to the multiphysics coupling simulations were excluded.

Due to software export controls, the specifics of the actual method calls cannot be provided. It is hoped that, in the future, the content of `SerpentTimeStepper` will either be cleared for open release or be made available upon request from the Serpent development team. Until then, the restricted content of `SerpentTimeStepper.C` (provided in file E.16) has been redacted.

7.4 Testing

A multiphysics simulation was created that involved three MOOSE MultiApps—water, fuel, and neutronics—to simulate a 365.76 cm long boiling-water reactor (BWR) fuel pin. The fuel radius was 0.605 79 cm. Cladding was neglected for simplicity, and the total thermal transport effects were approximated by a 0.194 21 cm thick sheath of water. It approximation partially represented the combined effect of the fast thermal conductivity of Zircaloy-2 and with the gap heat transfer effects. The outside border of the water MultiApp had a convective

heat flux boundary condition applied using a heat transfer coefficient of 3.5 W cm^{-2} K^{-1} and a T_∞ ranging as 330 °C to 360 °C from bottom to top. The initial conditions of the entire system were at an equilibrium temperature of 300 °C. A transient-type simulation was run for 5 0s with the fuel pin producing heat at a constant rate of 30 kW.

7.4.1 Models

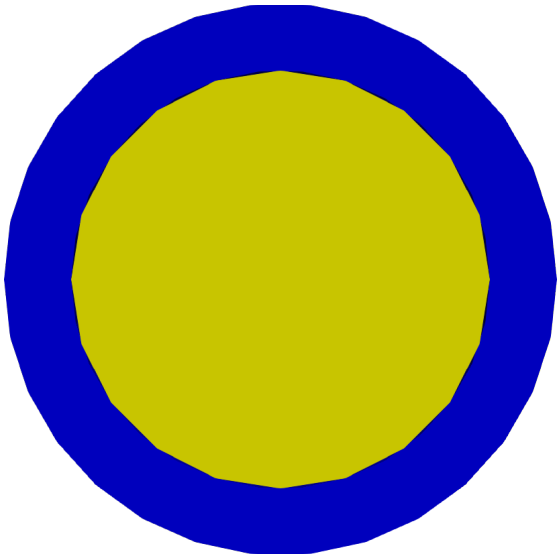
Figure 7.2a shows an XY cross section of the model meshes used in MOOSE; blue identifies the water MultiApp mesh and yellow identifies the fuel MultiApp mesh. Both the water and fuel meshes were defined using the `AnnularMesh` action, combined with the `MeshExtruder` and `Transform` invocations of the `MeshModifier` action set. The Serpent model was constructed using the typical CSG approach for nuclear MC codes; and XY cross section of the model is shown in fig. 7.2b. Note that, in simulation space, the dimensions of the fuel region in the Serpent model are identical to the fuel region in the meshed MOOSE model.

The gas gap between the fuel and cladding, which corresponds to the Serpent model's void region (see fig. 7.2b), has been removed from the MOOSE model for simplicity (see figs. 7.2a and 7.2c). The focus of this test is to effectively demonstrate the FE-based coupling capabilities, not to create a "real-world complete" model. Introducing too many complexities into the model would blur the distinction between the actual FE-based coupling behavior and each field's solution. Using such a simplification aids the demonstration by targeting the testing and evaluation to the coupling method itself, thereby preserving a clear separation from the remaining multiphysics effects.

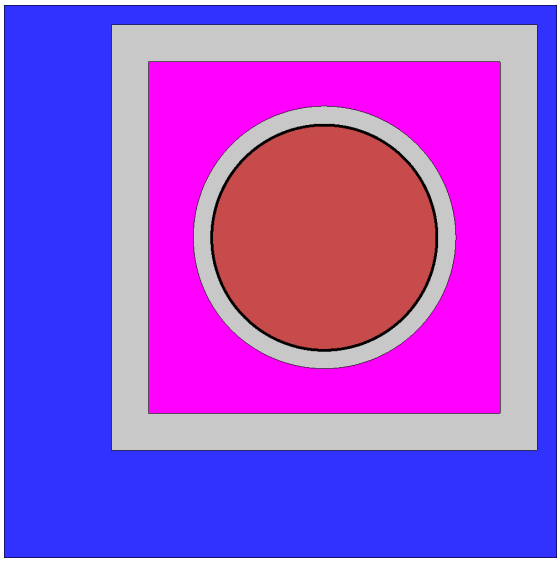
7.4.2 Coupling

This test will leverage the three types of coupling available via FES: boundary flux, boundary value, and volume value. The organization hierarchy is shown in fig. 7.3.

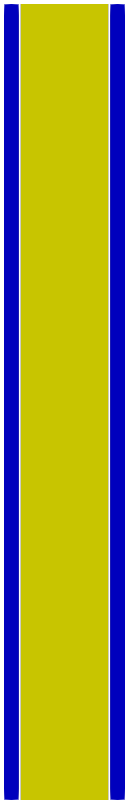
The water MultiApp is the main MasterApp for the simulation, acting as a driver, so it performs a solution step first (1). Next, the heat flux at the boundary is rendered as an FE using `AxialShell` and transferred to the fuel pin MultiApp (2). Heat conduction is performed in the fuel pin (3.a), then the temperature field is rendered as an FE and transferred



(a) XY cross section of the coupled MOOSE test meshes.



(b) XY cross section of the coupled Serpent test model. From inside to outside the material regions are: fuel, void, Zircaloy-2, coolant, coolant channel wall, moderator.



(c) MOOSE fuel pin model at 0.03 axial scale, with the water mesh exploded out slightly for clarity.

Figure 7.2 – Cross sections of the coupled test geometries.

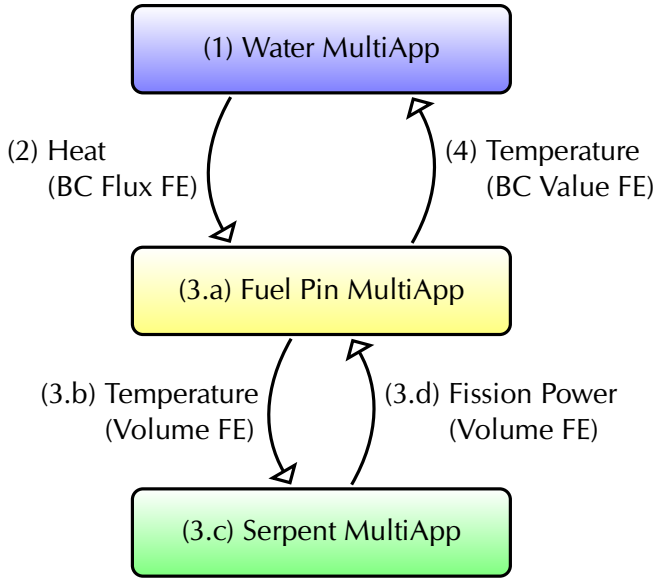


Figure 7.3 – MultiApp hierarchy of the test simulation for the MOOSE-Serpent coupling. Steps 3.a–d represent an inner iteration that was converged using Picard iterations, and then steps 1–4 are part of an outer iteration cycle that was also converged using Picard iterations.

to the Serpent() MultiApp (3.b), which then performs a stochastic simulation (3.c) and sends the fission power distribution FE back to the fuel pin MultiApp (3.d). This fission power distribution from Serpent, directly related to the MC-simulated neutron interaction rate and fission events, is explicitly treated as the heat generation term in MOOSE. Picard iterations are used in step 3 to loop through all the substeps until the convergence criteria are met at substep 3.a. The temperatures on the boundary are then collected and sent back to the water MultiApp for inclusion in its solution (4). Picard iterations are again used to cycle through the upper tier until the convergence criteria are met at step 1. The transfers 3.b and 3.d were relaxed using the 'windowed' option, with a window width of 4 and 60, respectively. Such a large window on the Serpent fission data transfer improves the quality of the FET coefficients. The relatively small window on the MOOSE temperature FES served only to dampen any oscillatory feedback introduced by coefficient estimators.

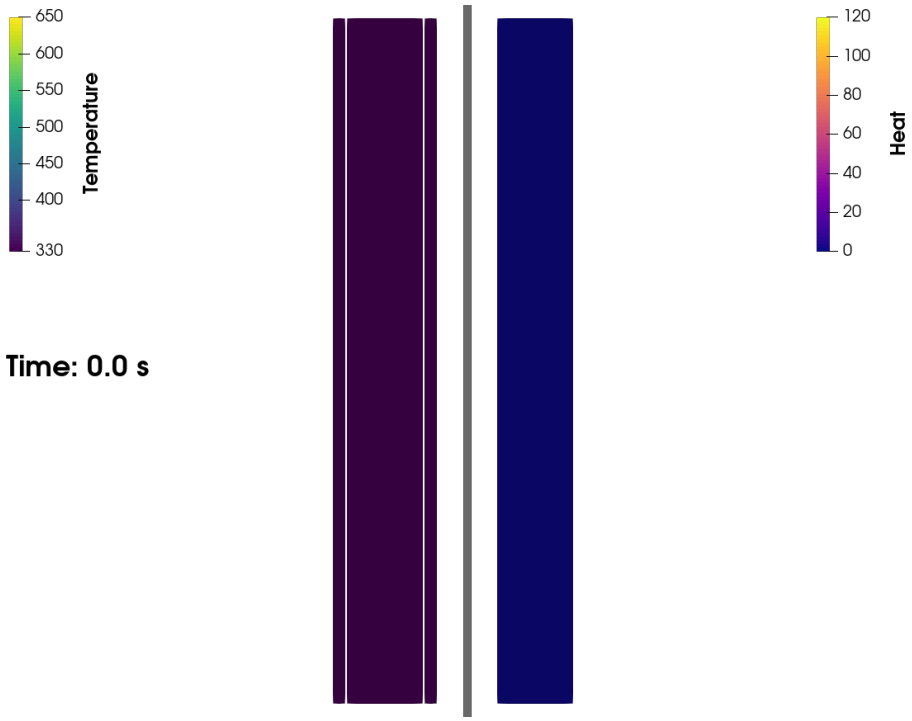
7.5 Results

The fully-coupled simulation required 19.9 h to complete. Visualization of the results are shown in fig. 7.4, which shows a cut of the simulation down the axial length of the model. Temperature is shown on the right, with the water mesh separated slightly for clarity, and the heat generation on the left. From $t = 1.0\,\mathrm{s}$ on, the heat generation and temperature fields changes with respect to the previous timestep are shown on the outside edges for comparison; again, temperature is on the right and heat generation is on the left.

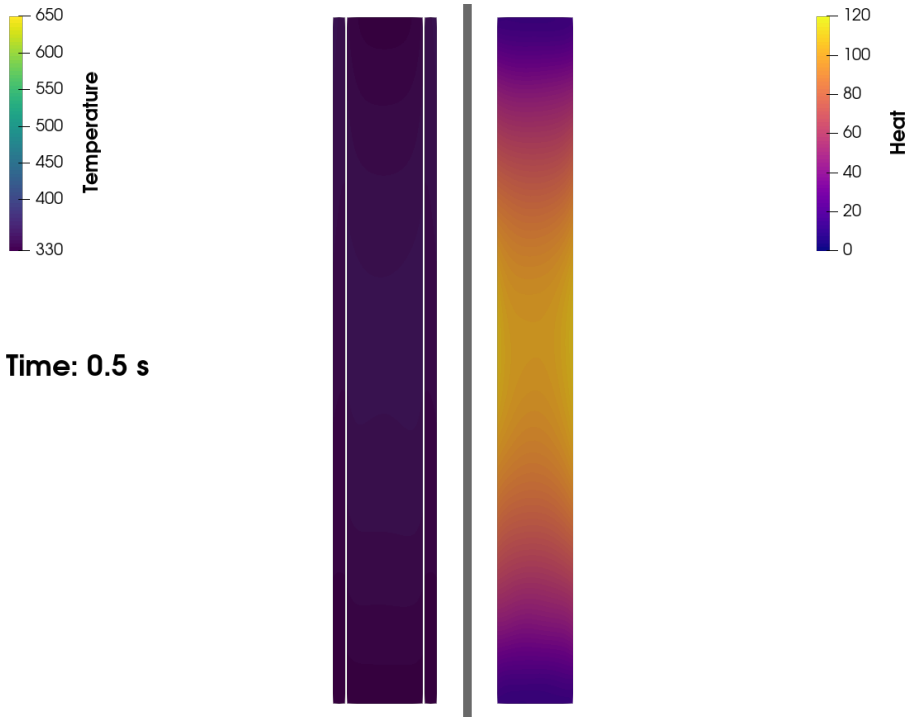
As could be anticipated, the neutronics-driven heat generation is axially focused toward the center. The effect of the moderation and reflection by the water is further visible as the heat generation tends to increase towards the outside edge of the fuel pin. Although this was as expected, it provided a confirmation that the Serpent simulation was configured correctly.

Also, most of the increases in heat generation tended to be away from the center of the fuel pin. This is an expected multiphysics feedback effect; as the temperature in the center of the fuel pin increases, the impact of Doppler broadening in the Serpent simulation pushes the fission reaction away from hot spots. Of all the timesteps, only $t = \{10.0\,\mathrm{s}, 12.0\,\mathrm{s}\}$ (figs. 7.4u and 7.4y) have a concentration of increasing heat generation near the center of the fuel pin; in all the others, any increases at the center are fringe effects from other nearby hot spots.

Further, the localized changes in heat generation tend to be highest where, in the previous timestep, they were lowest. This is yet another effect of a temperature-dependent localized feedback in the neutron interaction rate. This is generally true for the duration of the simulation (figs. 7.4c to 7.4z), but is especially apparent by comparing timesteps $t = \{2.5\,\mathrm{s}, 7.0\,\mathrm{s}, 9.5\,\mathrm{s}, 11.0\,\mathrm{s}\}$ (figs. 7.4f, 7.4o, 7.4t and 7.4w) to the previous timestep. A partial cause of this behavior is admittedly due to forcing the total heat generation to a constant total power for each iteration; because the integral power of the fuel pin is fixed, the heat generation in localized hot spots is depressed only relatively in comparison to the total heat generation. In a more realistic simulation, the total heat generation at each timestep

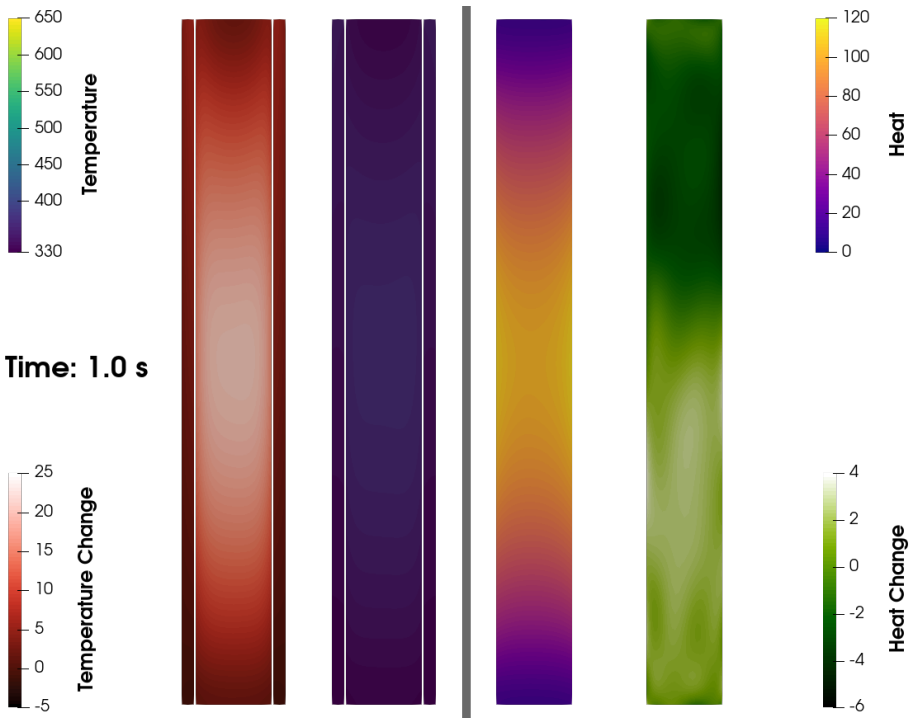


(a) Multiphysics fuel pin simulation at time $t = 0.0\,\mathrm{s}$

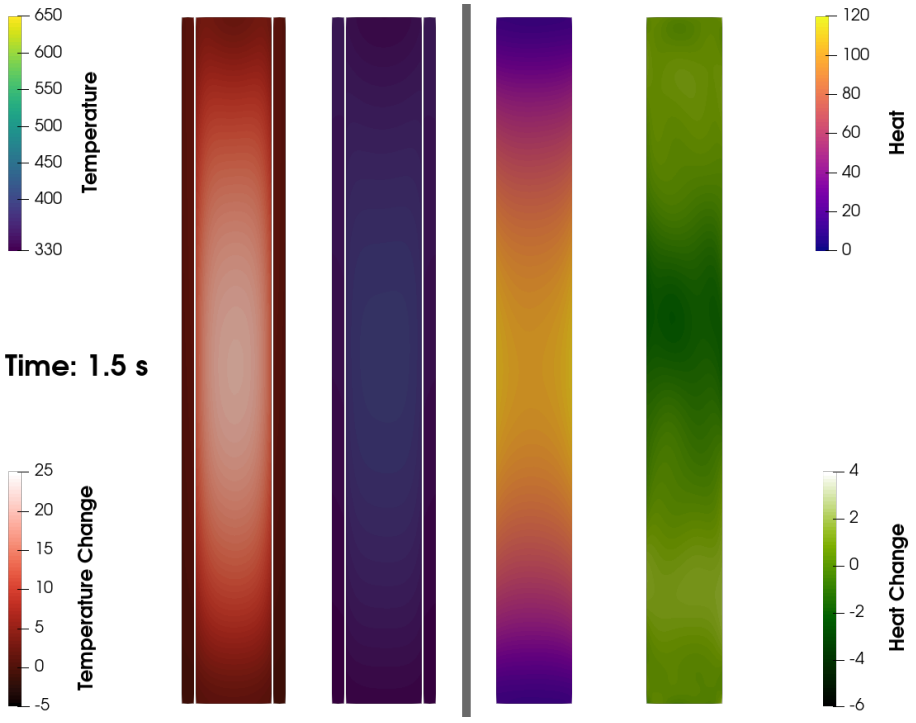


(b) Multiphysics fuel pin simulation at time $t = 0.5\,\mathrm{s}$

Figure 7.4 – Results of the MOOSE-Serpent coupled multiphysics simulation, at 0.03 axial scale. From left to right: the temperature, the heat generation. *cont...*

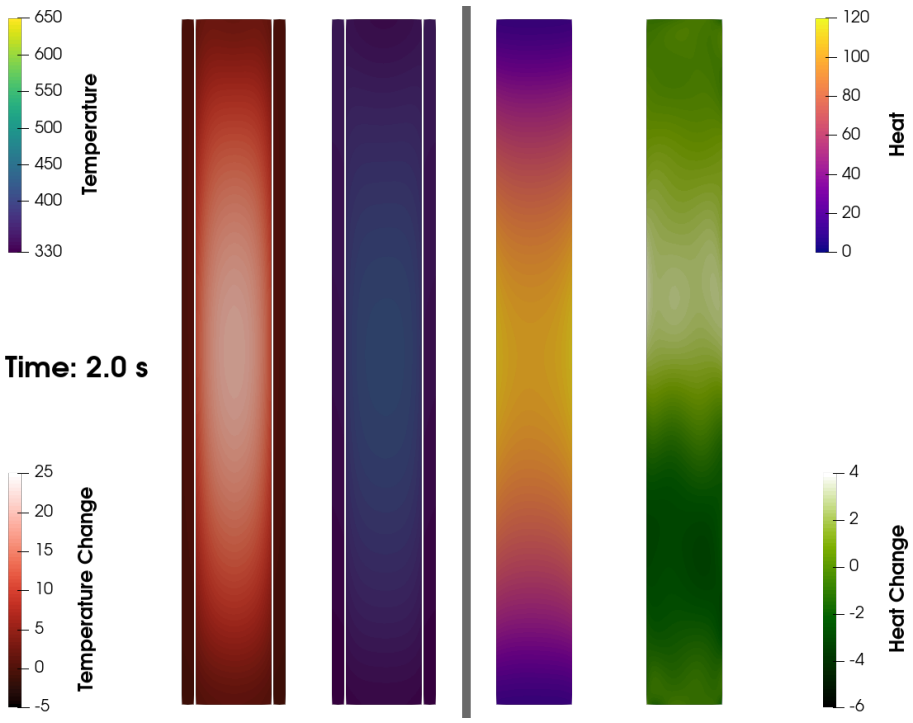


(c) Multiphysics fuel pin simulation at time $t = 1.0\,\mathrm{s}$

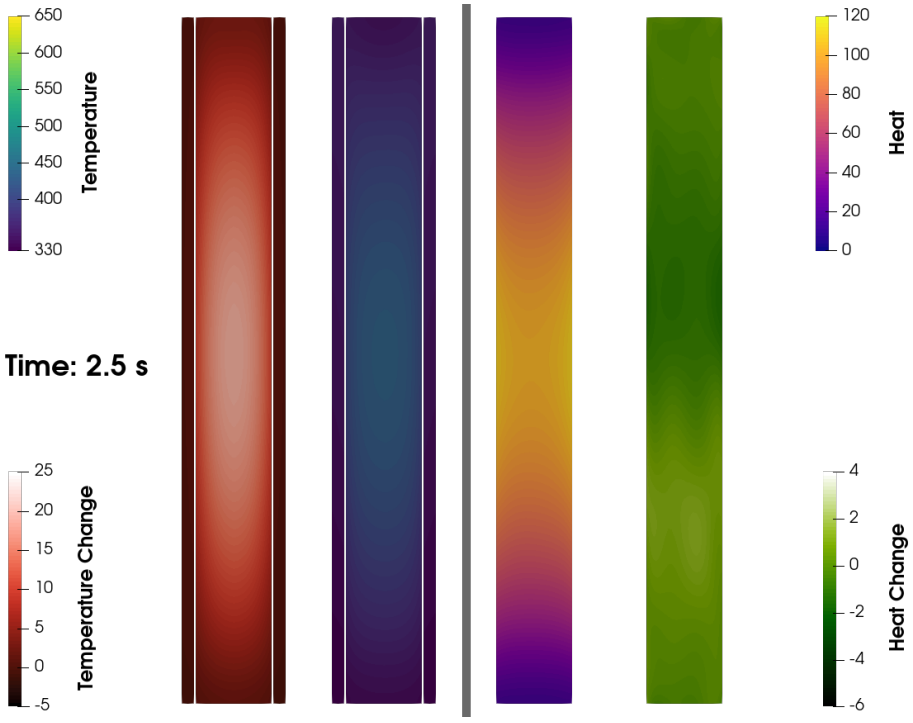


(d) Multiphysics fuel pin simulation at time $t = 1.5\,\mathrm{s}$

fig. 7.4 cont... Results of the MOOSE-Serpent coupled multiphysics simulation, at 0.03 axial scale. From left to right: the change from the previous timestep's temperature, the current temperature, the current heat generation, the change from the prevoious timestep's heat generation.

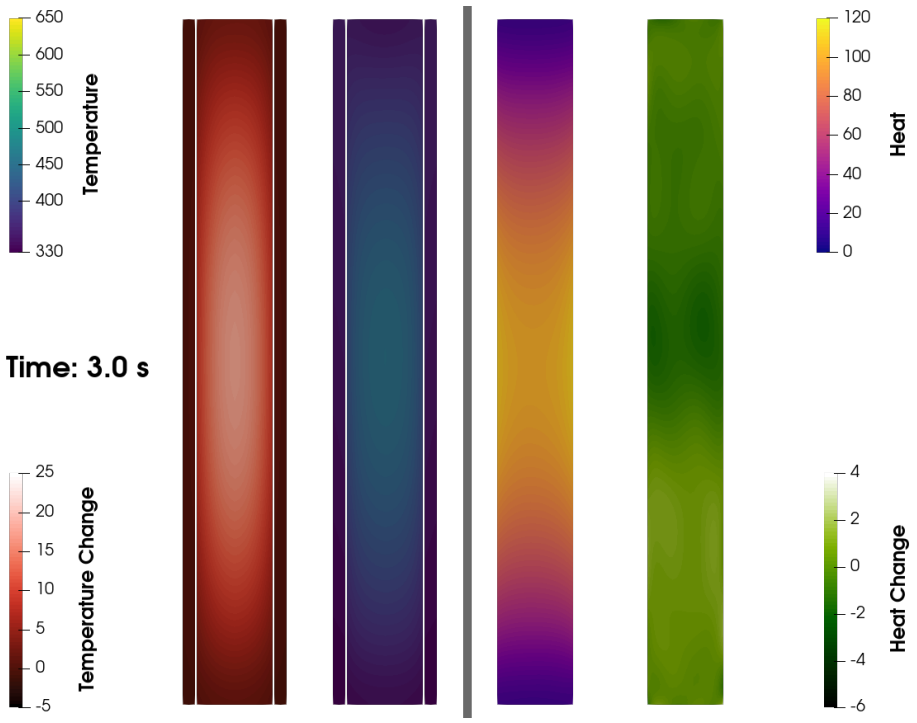


(e) Multiphysics fuel pin simulation at time $t = 2.0\,\mathrm{s}$

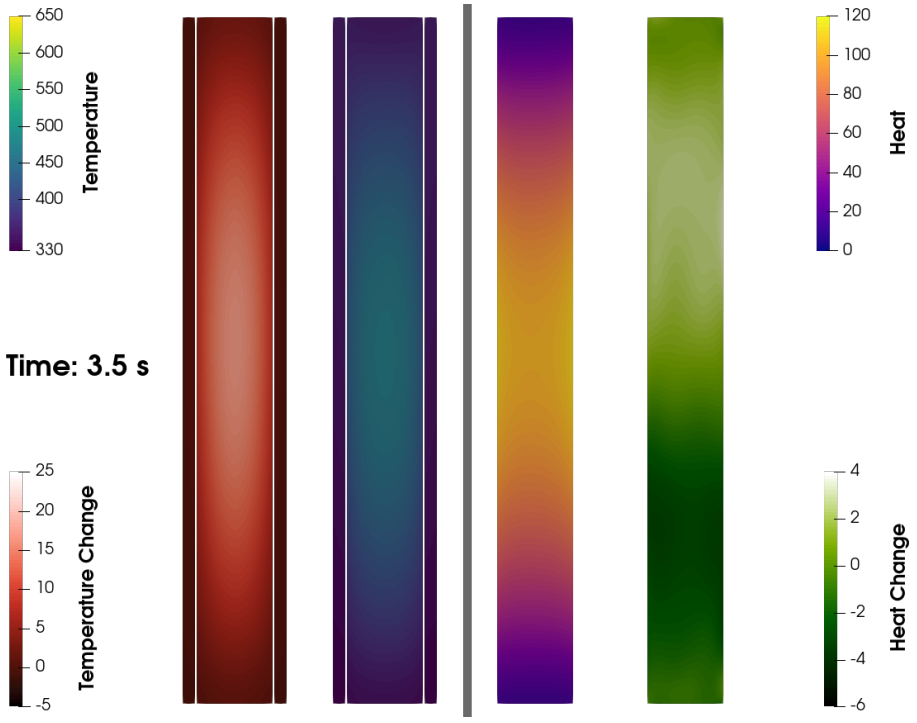


(f) Multiphysics fuel pin simulation at time $t = 2.5\,\mathrm{s}$

fig. 7.4 cont... Results of the MOOSE-Serpent coupled multiphysics simulation, at 0.03 axial scale. From left to right: the change from the previous timestep's temperature, the current temperature, the current heat generation, the change from the prevoious timestep's heat generation.

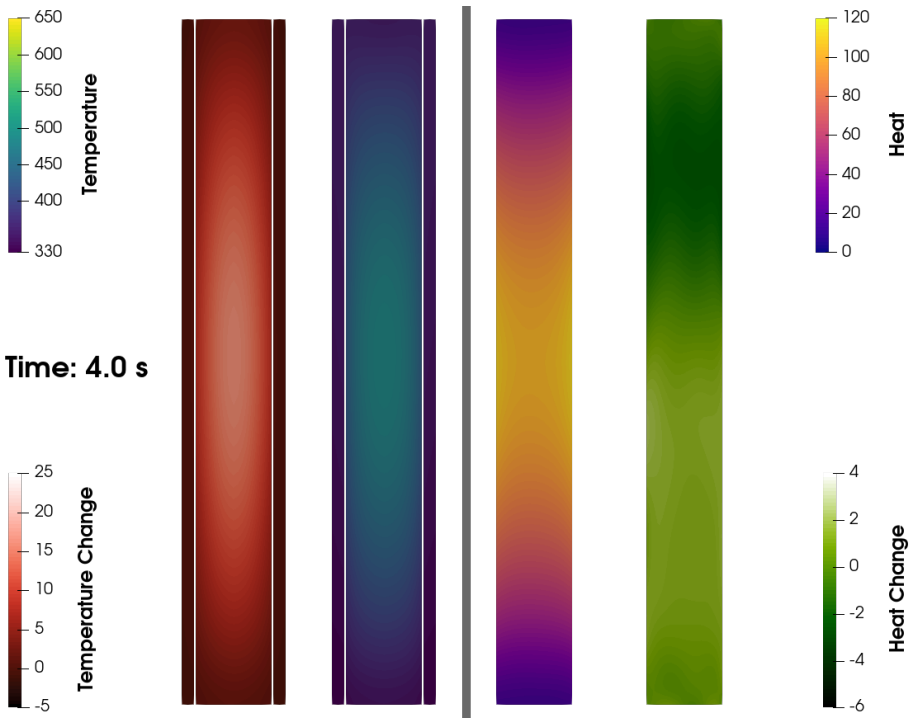


(g) Multiphysics fuel pin simulation at time $t = 3.0\,\mathrm{s}$

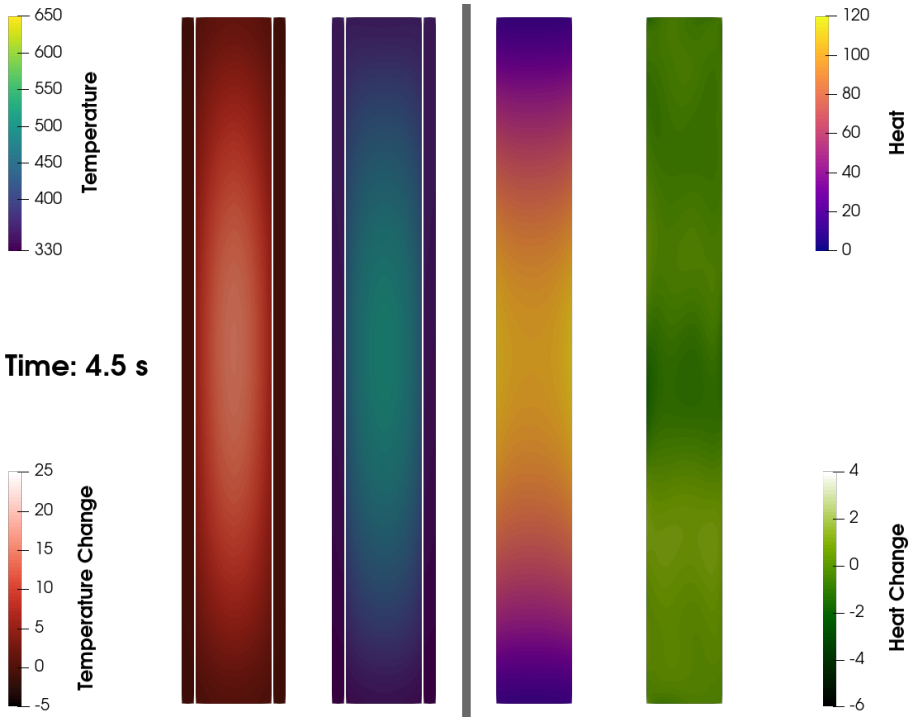


(h) Multiphysics fuel pin simulation at time $t = 3.5\,\mathrm{s}$

fig. 7.4 cont... Results of the MOOSE-Serpent coupled multiphysics simulation, at 0.03 axial scale. From left to right: the change from the previous timestep's temperature, the current temperature, the current heat generation, the change from the prevoious timestep's heat generation.

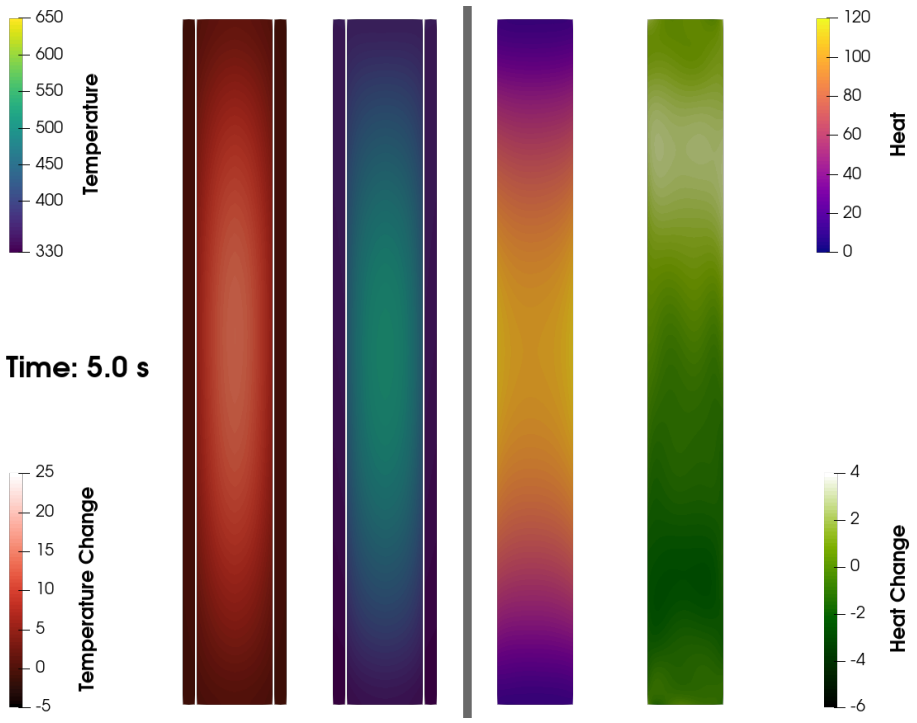


(i) Multiphysics fuel pin simulation at time $t = 4.0\,\mathrm{s}$

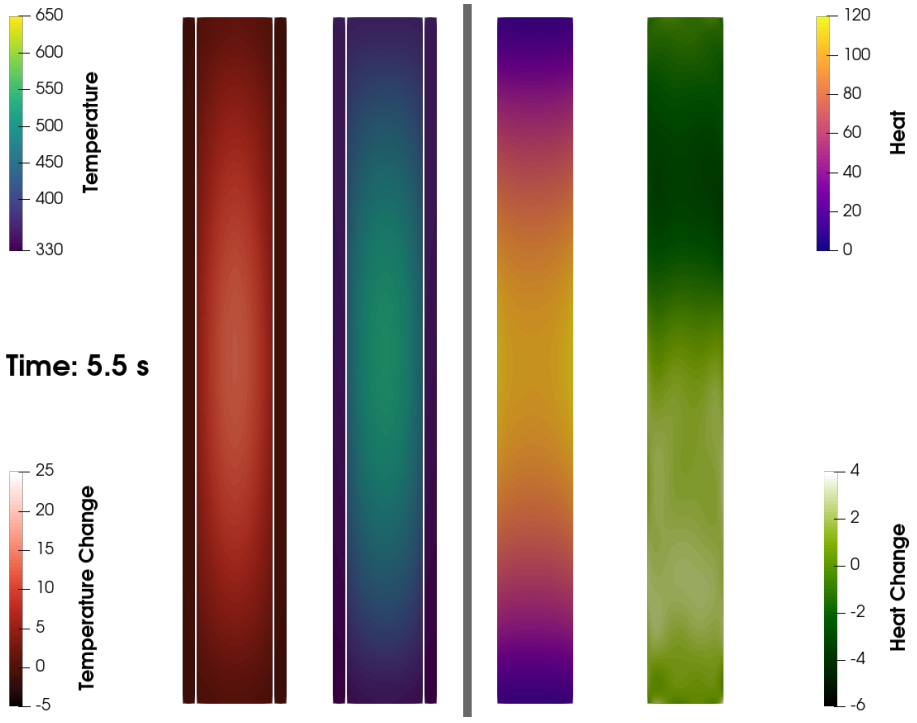


(j) Multiphysics fuel pin simulation at time $t = 4.5\,\mathrm{s}$

fig. 7.4 cont... Results of the MOOSE-Serpent coupled multiphysics simulation, at 0.03 axial scale. From left to right: the change from the previous timestep's temperature, the current temperature, the current heat generation, the change from the prevoious timestep's heat generation.

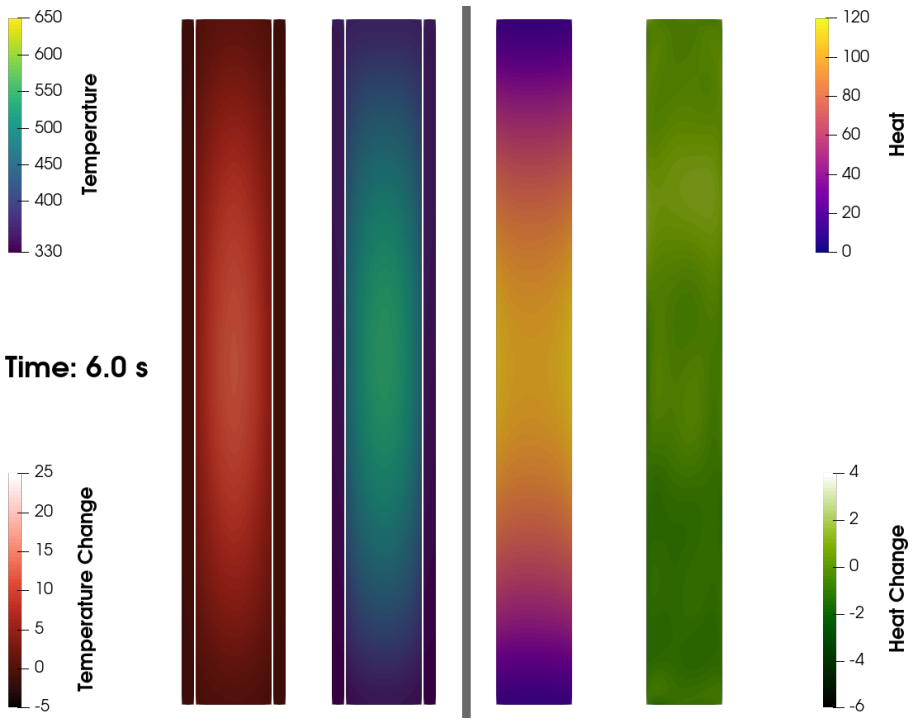


(k) Multiphysics fuel pin simulation at time $t = 5.0\,\mathrm{s}$

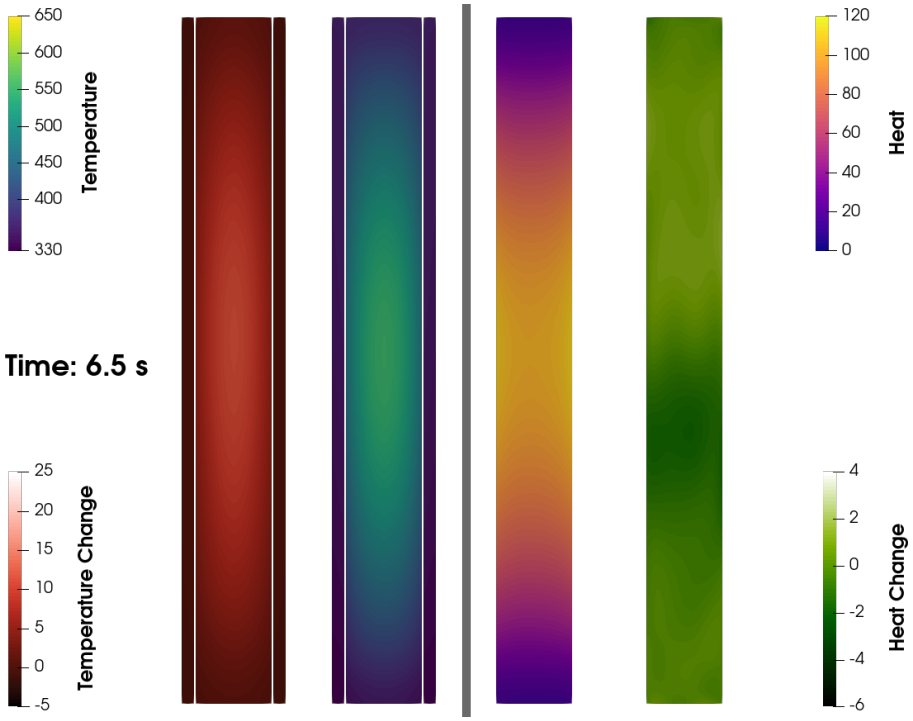


(l) Multiphysics fuel pin simulation at time $t = 5.5\,\mathrm{s}$

fig. 7.4 cont... Results of the MOOSE-Serpent coupled multiphysics simulation, at 0.03 axial scale. From left to right: the change from the previous timestep's temperature, the current temperature, the current heat generation, the change from the prevoious timestep's heat generation.

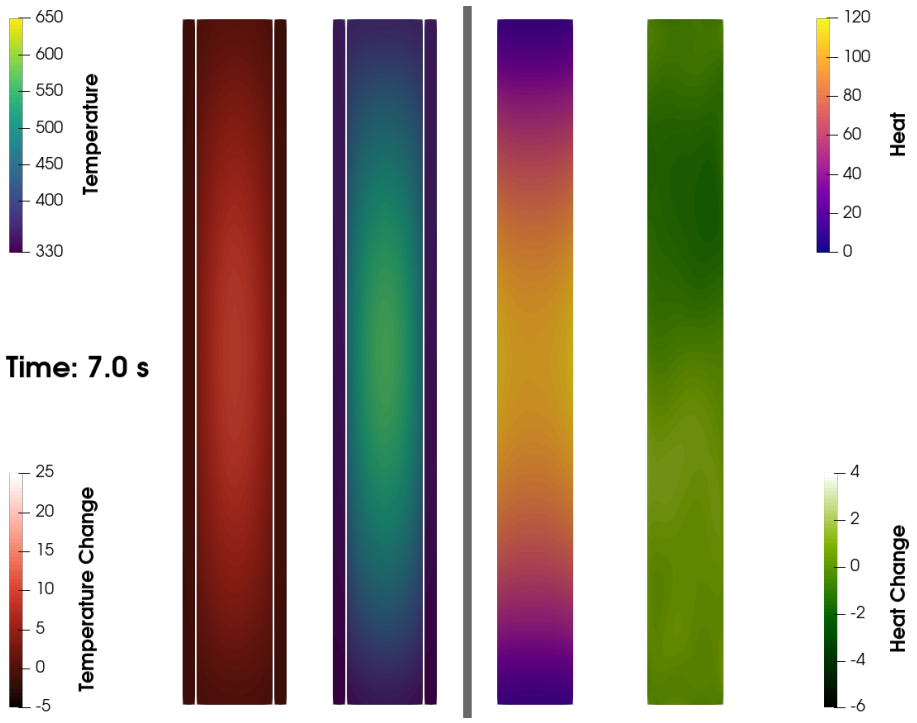


(m) Multiphysics fuel pin simulation at time $t = 6.0\,\mathrm{s}$

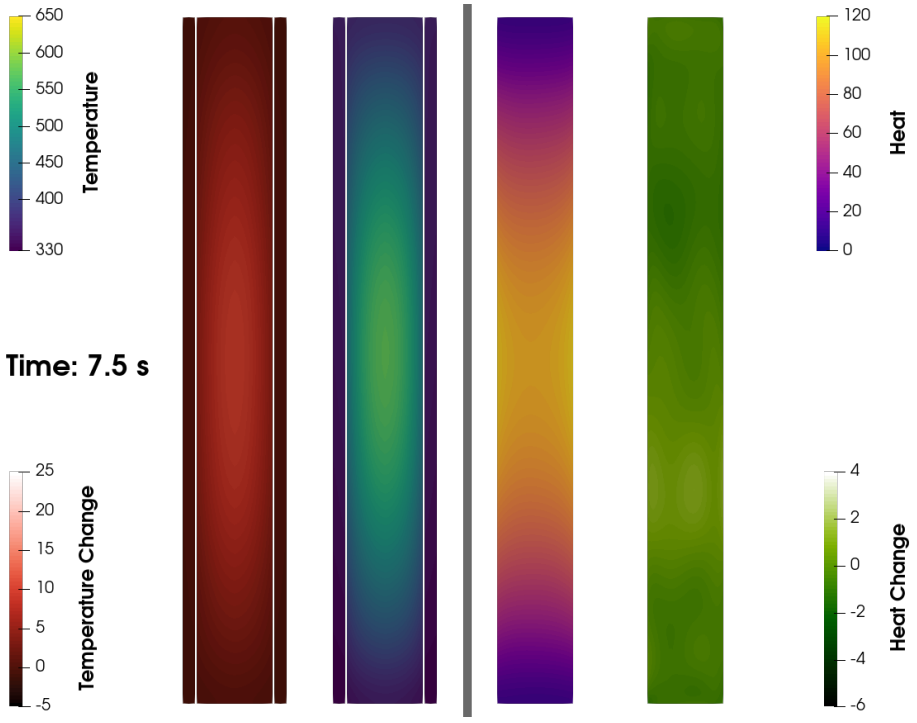


(n) Multiphysics fuel pin simulation at time $t = 6.5\,\mathrm{s}$

fig. 7.4 cont... Results of the MOOSE-Serpent coupled multiphysics simulation, at 0.03 axial scale. From left to right: the change from the previous timestep's temperature, the current temperature, the current heat generation, the change from the prevoious timestep's heat generation.

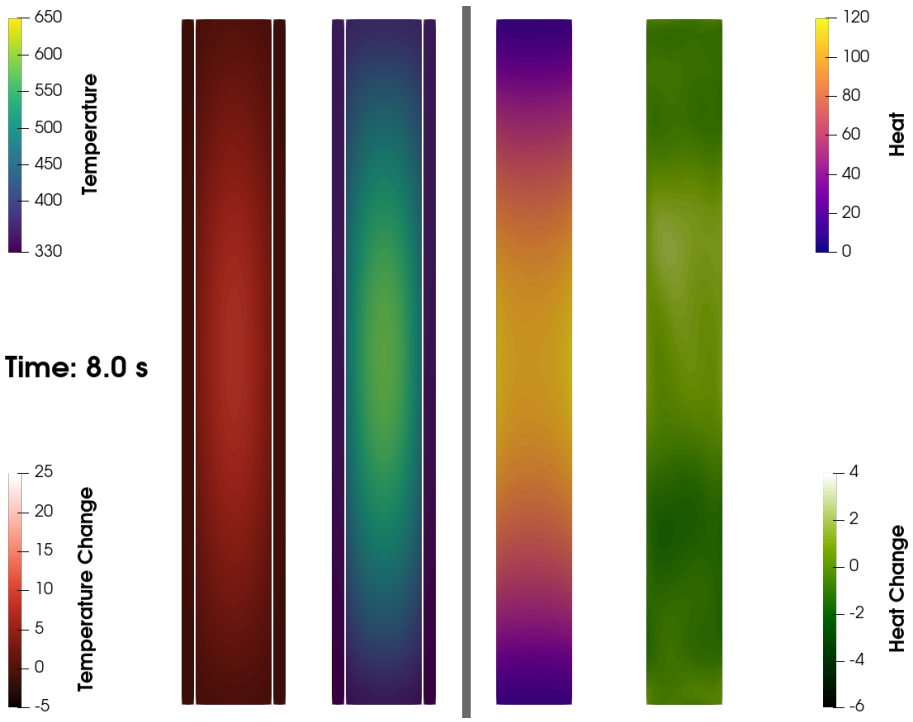


(o) Multiphysics fuel pin simulation at time $t = 7.0\,\mathrm{s}$

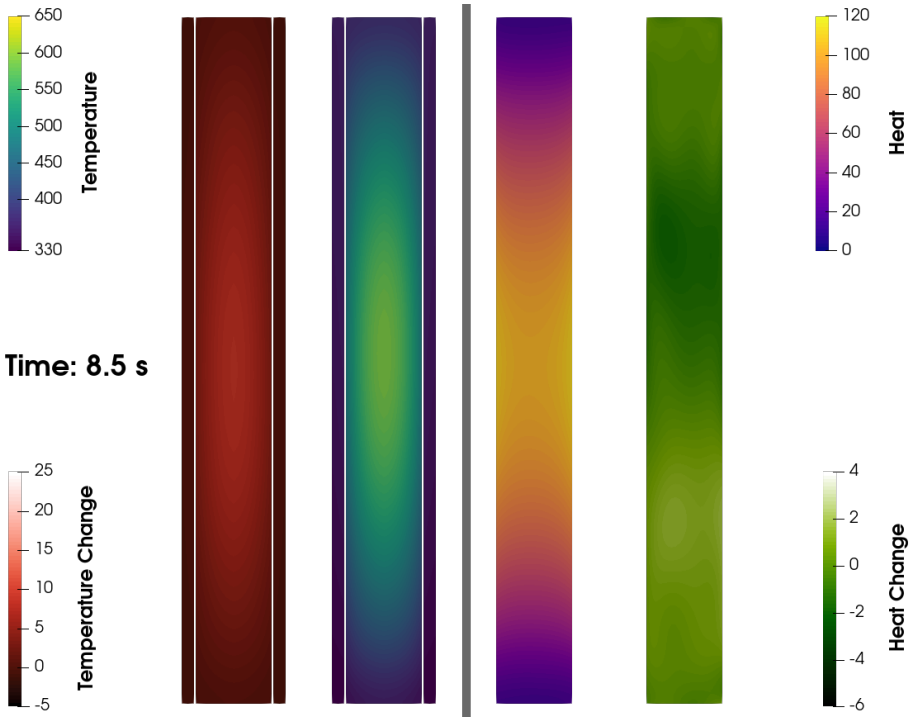


(p) Multiphysics fuel pin simulation at time $t = 7.5\,\mathrm{s}$

fig. 7.4 cont... Results of the MOOSE-Serpent coupled multiphysics simulation, at 0.03 axial scale. From left to right: the change from the previous timestep's temperature, the current temperature, the current heat generation, the change from the prevoious timestep's heat generation.

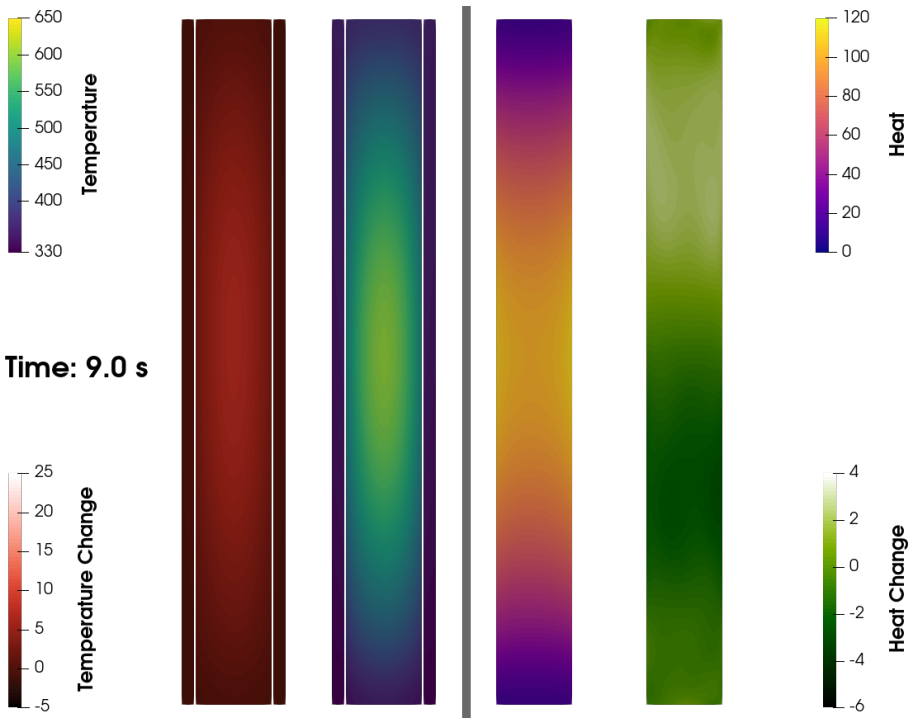


(q) Multiphysics fuel pin simulation at time $t = 8.0\,\text{s}$

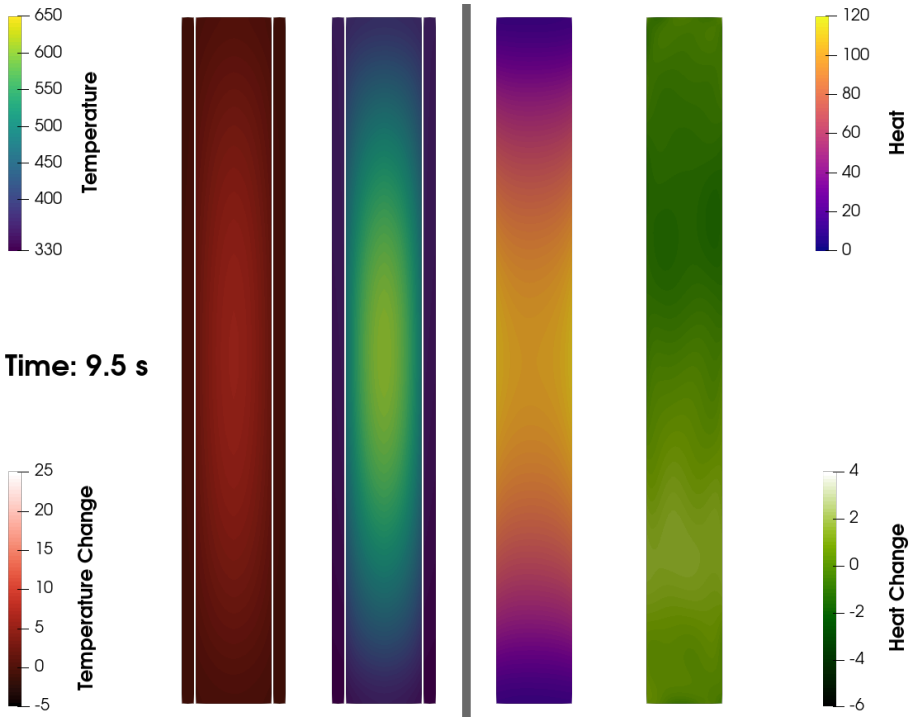


(r) Multiphysics fuel pin simulation at time $t = 8.5\,\text{s}$

fig. 7.4 cont... Results of the MOOSE-Serpent coupled multiphysics simulation, at 0.03 axial scale. From left to right: the change from the previous timestep's temperature, the current temperature, the current heat generation, the change from the prevoious timestep's heat generation.

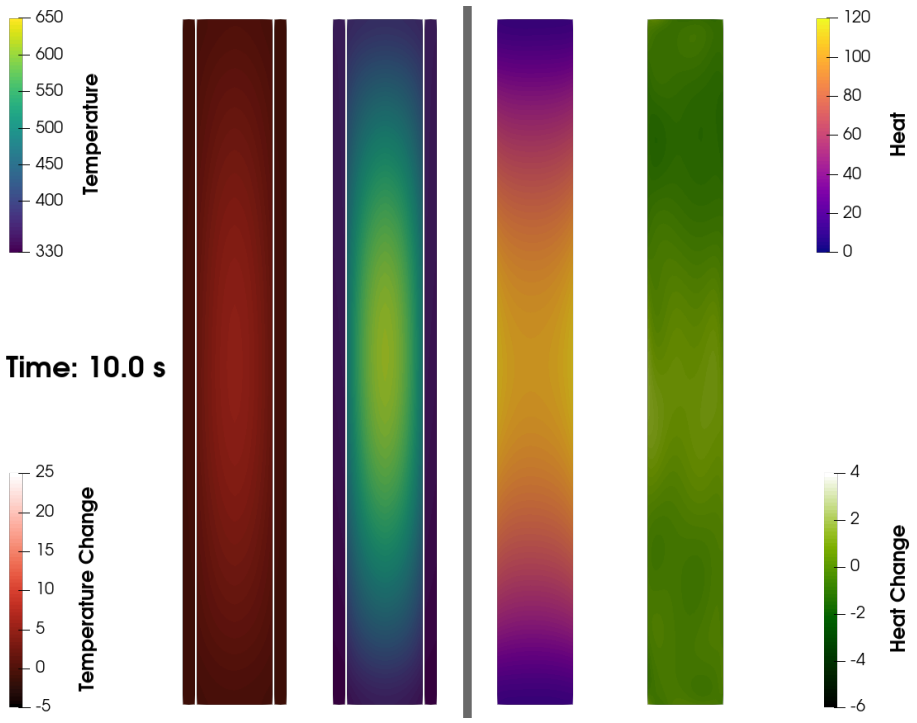


(s) Multiphysics fuel pin simulation at time $t = 9.0\,\mathrm{s}$

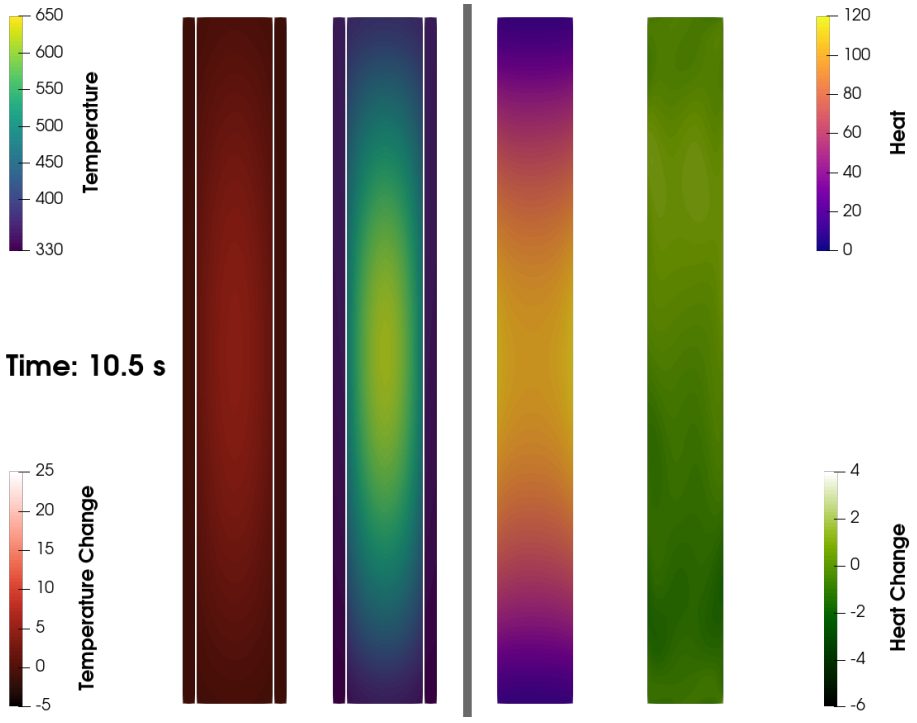


(t) Multiphysics fuel pin simulation at time $t = 9.5\,\mathrm{s}$

fig. 7.4 cont... Results of the MOOSE-Serpent coupled multiphysics simulation, at 0.03 axial scale. From left to right: the change from the previous timestep's temperature, the current temperature, the current heat generation, the change from the prevoious timestep's heat generation.

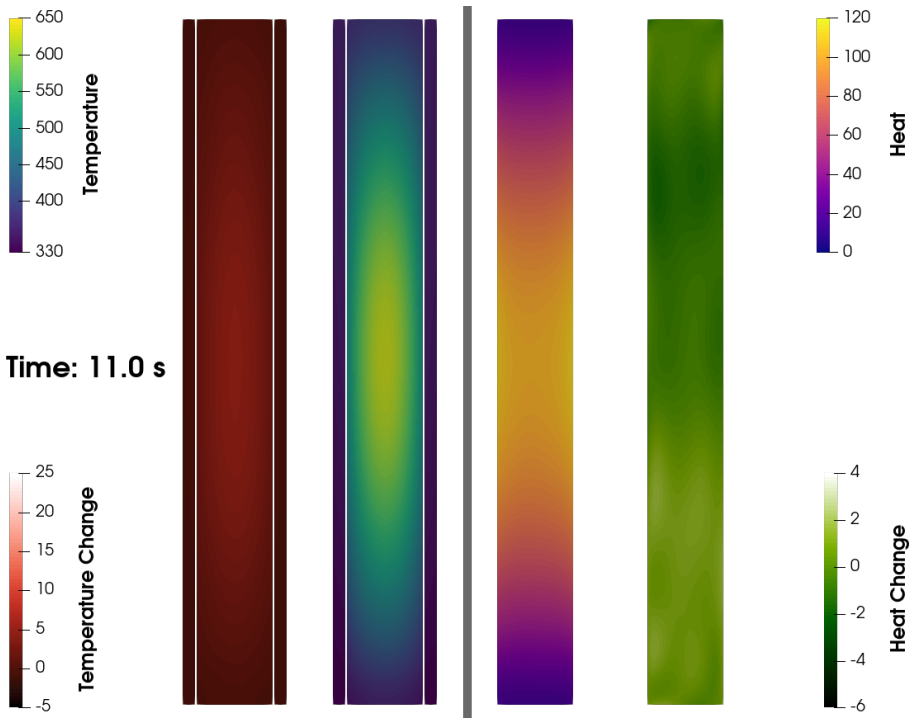


(u) Multiphysics fuel pin simulation at time $t = 10.0\,\mathrm{s}$

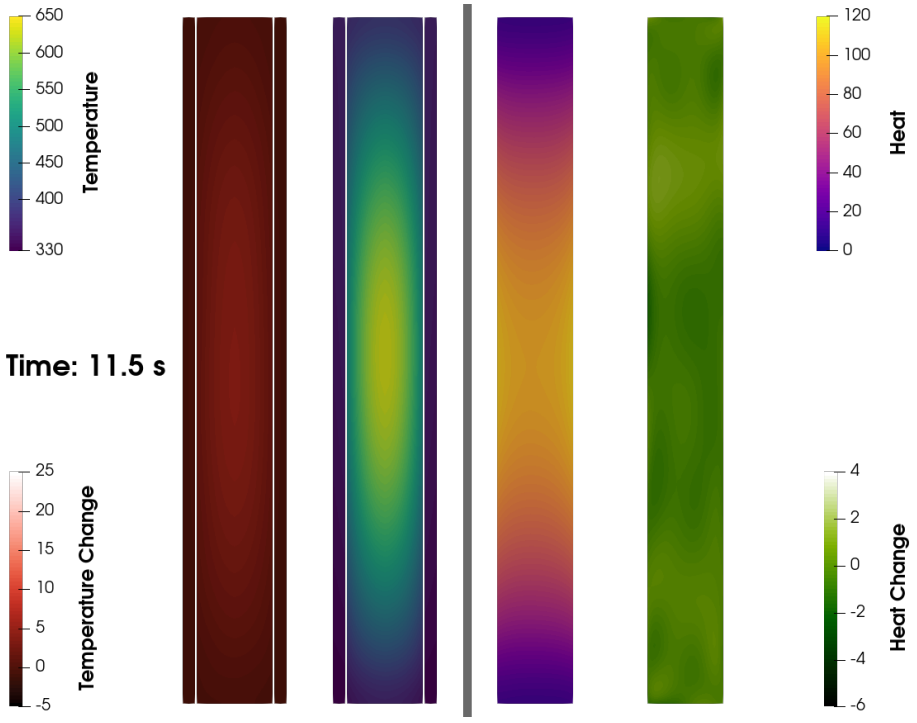


(v) Multiphysics fuel pin simulation at time $t = 10.5\,\mathrm{s}$

fig. 7.4 cont... Results of the MOOSE-Serpent coupled multiphysics simulation, at 0.03 axial scale. From left to right: the change from the previous timestep's temperature, the current temperature, the current heat generation, the change from the prevoious timestep's heat generation.

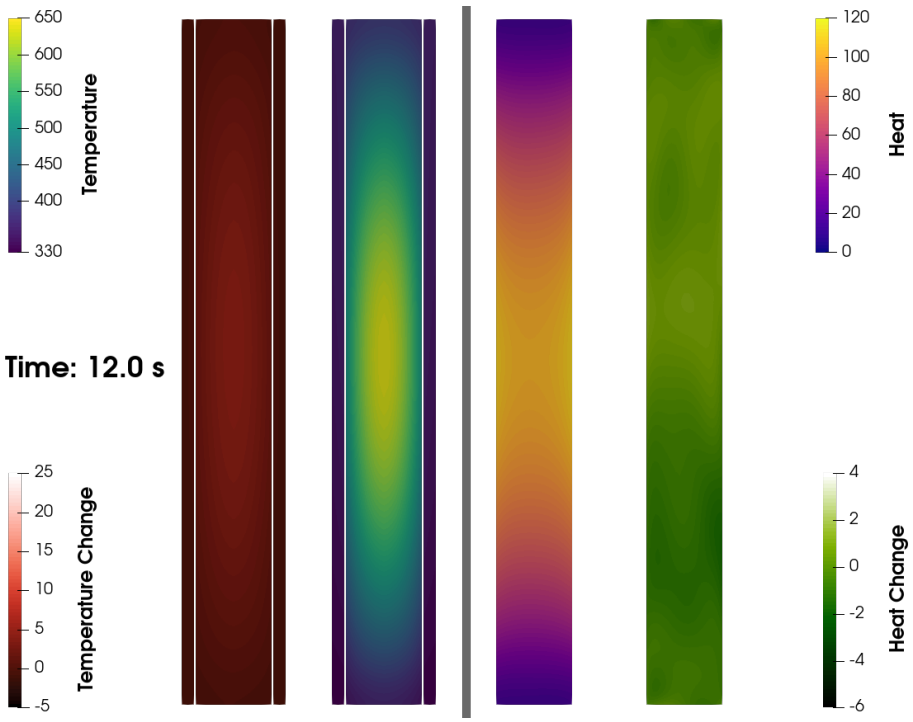


(w) Multiphysics fuel pin simulation at time $t = 11.0\,\mathrm{s}$

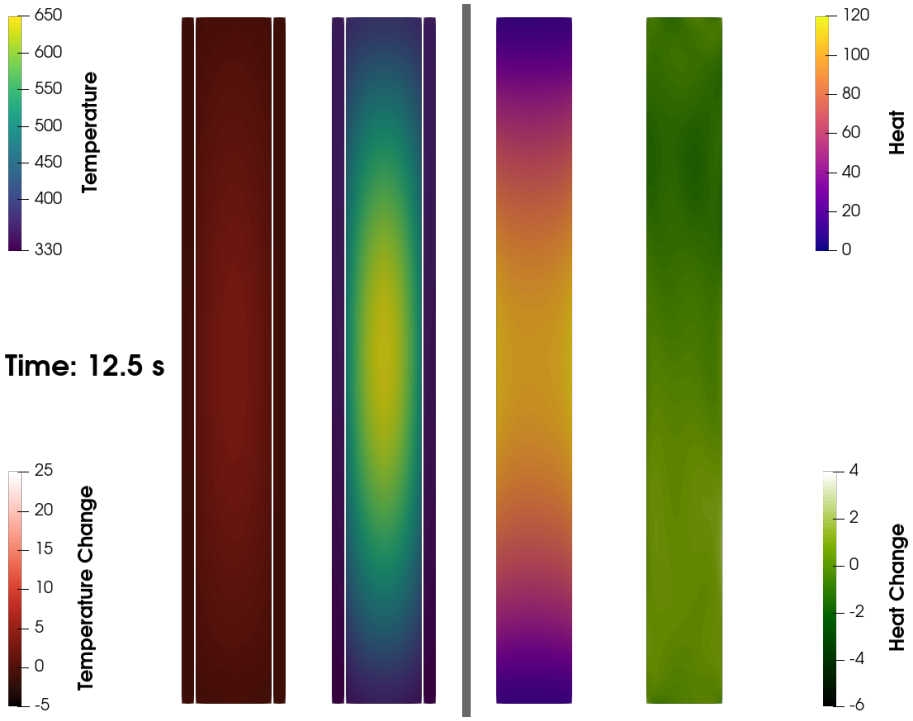


(x) Multiphysics fuel pin simulation at time $t = 11.5\,\mathrm{s}$

fig. 7.4 cont... Results of the MOOSE-Serpent coupled multiphysics simulation, at 0.03 axial scale. From left to right: the change from the previous timestep's temperature, the current temperature, the current heat generation, the change from the prevoious timestep's heat generation.



(y) Multiphysics fuel pin simulation at time $t = 12.0\,\mathrm{s}$



(z) Multiphysics fuel pin simulation at time $t = 12.5\,\mathrm{s}$

fig. 7.4 cont... Results of the MOOSE-Serpent coupled multiphysics simulation, at 0.03 axial scale. From left to right: the change from the previous timestep's temperature, the current temperature, the current heat generation, the change from the prevoious timestep's heat generation.

would also be a function of other factors such as the MC simulation's effective multiplication k_{eff}. Nevertheless, these results demonstrate that the multiphysics aspect of the coupling is indeed working, and that Serpent is responding to the changes in temperature calculated by the MOOSE fuel pin MultiApp.

Lastly, the temperature field behaves as expected. Increases in temperature are strongly correlated to the actual heat generation (as opposed to the minor variations within the heat generation field itself). However, the FE-based coupling is still able to capture visually-indistinguishable characteristics in the temperature distribution. The location specificity of this feedback within the heat generation simulation (fission power density or neutron interaction rate) is very sensitive, typically responding to temperature changes of less than 5 °C ($\approx$ 0.1 % to 1.5 %, depending on the timestep). FEs and FETs alone are able to provide this continuous highly-sensitive coupling methodology for high-fidelity multiphysics simulations between codes with completely different underlying geometries and/or models.

7.6 Conclusions

The capability to build and integrate Serpent as a library within MOOSE was created. This utilized a paired `Makefile` and bash script to copy, modify, and build the Serpent source files. Functionality was included to detect minor updates and build only incrementally as required.

Next, FET capabilities were added to Serpent's multiphysics interface. Much of the work implementing FETs in the detector framework was reused, although a few modifications were required to make the methods cross-compatible. The validity of the FET-based multiphysics interface code was ensured by visually inspecting a plot of the sampled temperatures and comparing the results to the expected distribution based on the input coefficients.

Subsequently, a MOOSE-Serpent driver was created that used FEs to communicate data for multiphysics M&S. This included the MOOSE-based classes `SerpentExecutioner` and `SerpentTimeStepper` for interacting with Serpent via the newly FE-enabled multiphysics interface. These allowed Serpent to acquire temperature or density information from MOOSE,

then return the MC-sampled fission power density data. Internally, `SerpentExecutioner` was based on a newly developed class, `FXExecutioner`, which provides extensive support for coupling to external applications using FE-based distributions. Methods for relaxation and convergence acceleration were also provided.

Finally, these developments were tested in an actual tightly-coupled MOOSE-Serpent multiphysics simulation. Both volumetric and boundary couplings were used, making the simulation an all-inclusive test suite for checking the entirety of the FE-based multiphysics code in both Serpent and MOOSE. Picard iterations were used to converge the finite element and MC simulation components; all data exchanges were performed using FES, in 3D and fully multivariate. The results positively indicated that FE-based 3D multiphysics coupling was performed between MOOSE and Serpent. Further, it was demonstrated that the FES showed remarkable sensitivity to the continuous spatial and temporal variations that are requisite for high-fidelity multiphysics M&S.

Part III

Outcomes

Chapter 8

Conclusions

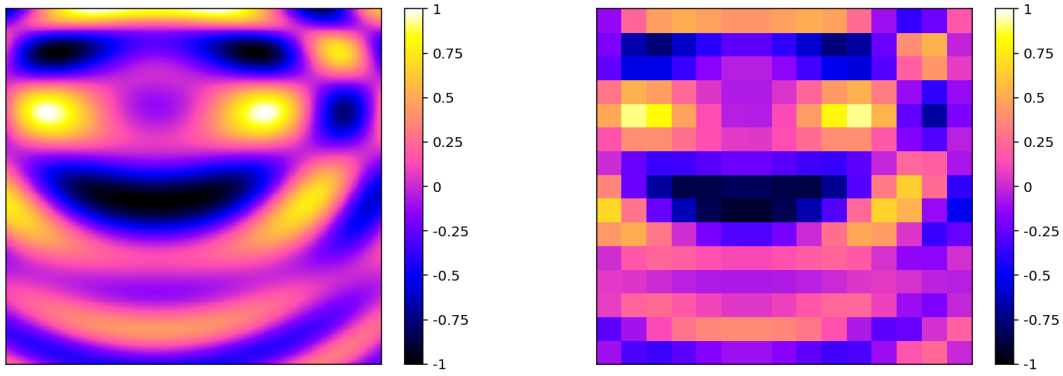
Where the telescope ends the microscope begins.

— Victor Hugo

Although the operational principles of the universe are constant, our comprehension of them is progressively revised. Often, scientific advancements of humankind's understanding are empowered by hypothesis and experiment. Throughout the ages, these modeling and simulation (M&S) efforts were often performed in laboratory test environments using scaled models or prototypes.

A significant number of modern M&S efforts are performed within a virtual laboratory. Year after year, larger and more complex simulations can be run due to corresponding evolutionary advances in computation hardware [18]. These are often effected as multiphysics simulations that encompass increasingly more physics fields, larger domains, and/or span numerous scales [2]. One objective is to model problems that were previously computationally-intractable, another is to improve the fidelity and accuracy of existing models.

Multiphysics simulation researchers are constantly seeking for methods to improve the fidelity and accuracy of their models [21]. Representation and communication of high-fidelity physics data undergird such advances; a very fundamental requirement of multiphysics simulations is the conveyance of data from one domain, field, or scale to another.



(a) Optically resolved FE representation using 225 coefficients

(b) Mesh representation using 225 elements, exhibiting a much coarser image than at left.

Figure 8.1 – Comparison between FE-based and mesh-based data. This figure is reproduced from components of fig. 2.1.

8.1 Description of Work

This research was performed with an objective to improve and facilitate data transfer within multiphysics simulations. Previously, functional expansions (FEs) were shown to have desirable qualities for such situations—particularly when compared to zeroth-order data quantification such as averaged or histogrammed values. These characteristics include: 1) continuous representation [78], 2) high data density [79], and 3) improved convergence in sample-based simulations [81]. Figure 8.1 provides a visual example of how FEs exemplify continuous representation and high data density compared to a zeroth-order mesh-based data quantification, while fig. 8.2 demonstrates high data density and improved convergence via a decrease in total error.

It is significant to note that, throughout this work, non-separable or multivariate FEs were used. This is in direct contrast to earlier research and implementations, in which only separable FEs were used. The reasons for using separable FEs often were a result of the associated algorithm complexity and/or large computational expense. Nevertheless, in these earlier implementations the potential for using multivariate FEs was often acknowledged cursorily. Furthermore, coefficients from a separable multidimensional FE are intrinsically compatible with multivariate implementations without stripping away information; an identically-ordered multivariate FE has the same functional terms as the separable FE, plus the

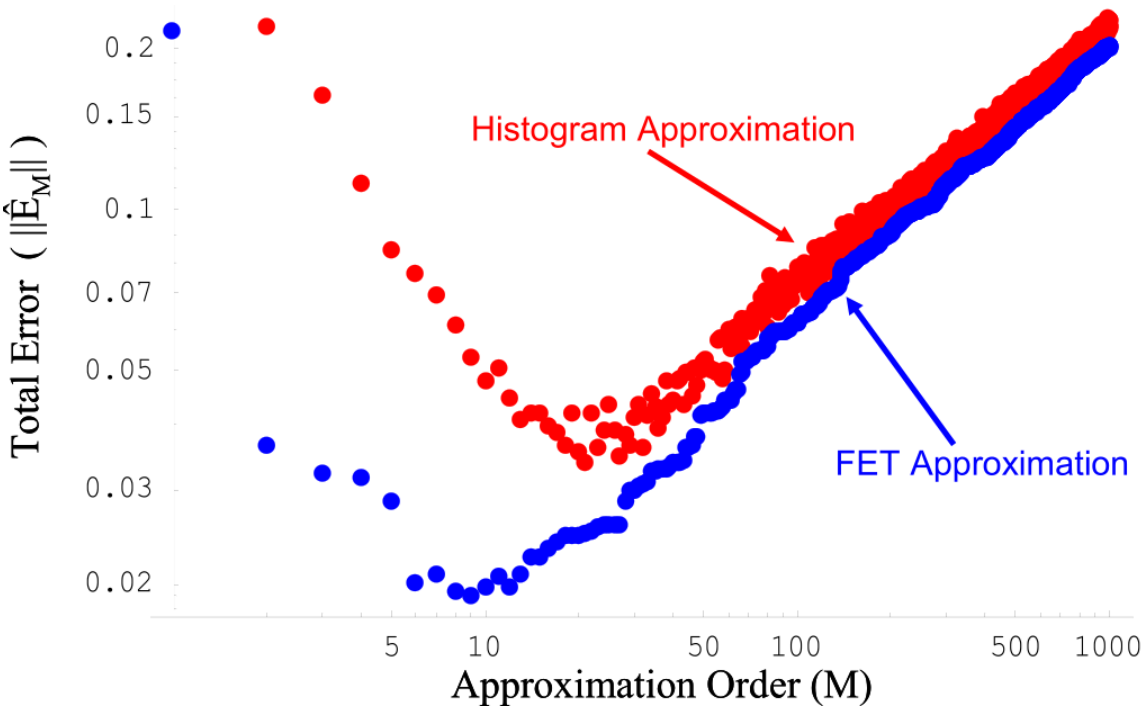


Figure 8.2 – A comparison of the total error present in representations of a stochastically sampled data set plotted against the approximation order. This image is reproduced here from fig. 2.2, again used with permission from *Functional Expansion Tallies for Monte Carlo Simulations* [76].

additional cross terms. The converse is not true, i.e., trying to use multivariate coefficients in a separable context essentially guarantees information loss due to the removal of all cross term coefficients.

Functional expansion methodologies have already been applied to data transfer and coupling in multiphysics simulations [71, 83]. The purpose and new unique contribution of this research in FES-based methods was to:

1. Improve the computation performance (Chapter 4)
2. Quantify the runtime-dependent convergence properties (Chapter 5)
3. Implement a flexible multiphysics coupling interface (Chapter 6)
4. Demonstrate an explicitly coupled 3D multiphysics simulation (Chapter 7)

8.2 Theory

Functional expansions are essentially a generalized form of the Fourier series. Instead of relying solely sin and cos function series over a cyclic domain, however, they provide the capability to select the underlying basis function series and applying the analysis to a bounded domain. The resulting set of coefficients correspond match the number of terms in the functional series. Each coefficient (or moment) represents the degree of correlation (or projection) between the corresponding series term and the measured data distribution. These term-wise coefficients are calculated using:

$$b_i = \int_{x_1}^{x_2} \breve{\psi}_i(x)\rho_\psi(x)F(x)dx \tag{8.1}$$

recreated here from eq. (3.14) for clarity. An FE is reconstructed as a continuous distribution using the relation:

$$F(x) = \sum_{i=0}^{\infty} b_i\psi_i(x) \tag{8.2}$$

recreated here from eq. (3.15) for clarity. Since using an infinite number of terms is not possible, these two relations are always truncated in i to a finite series order I.

Equations (8.1) and (8.2) are useful as long as the corresponding data distribution F can be integrated. In stochastic analyses, the distribution is integrated using Monte Carlo (MC) sampling. Thus, the version of eq. (8.1) for sample-based coefficient generation is:

$$b_i = \lim_{N\to\infty} \frac{1}{N} \sum_{n=0}^{N} S(n)\breve{\psi}_i(\mathbf{x}_n)\rho_\psi(\mathbf{x}_n) \tag{8.3}$$

recreated here from eq. (3.64) for clarity. The same relation, eq. (8.2), can be used for the reconstruction. These sample-based forms are referred to as functional expansion tallies, or functional expansion tallies (FETs). The corresponding statistical variance measure for each FET coefficient is:

$$\widehat{\sigma}^2_{\widehat{b}_i} = \frac{\sum_{p=1}^{N}\left(\sum_{\chi=1}^{p_X} S(p_\chi)\breve{\psi}_i(\mathbf{x}_{p_\chi,i})\rho_{\psi_i}(\mathbf{x}_{p_\chi,i})\right)^2 - \frac{1}{N}\left(\sum_{p=1}^{N}\sum_{\chi=1}^{p_X} S(p_\chi)\breve{\psi}_i(\mathbf{x}_{p_\chi,i})\rho_{\psi_i}(\mathbf{x}_{p_\chi,i})\right)^2}{N\,(N-1)} \tag{8.4}$$

recreated here from eq. (3.74) for clarity.

8.3 Algorithm Optimization

It is commonly acknowledged in the software-development field that "premature optimization is the root of all evil" (Donald Knuth). When first implementing an algorithm in code, it is much more important to get it right that to get it fast. The primary concept is to focus time and priorities on completion, rather than on optimizing perceived suboptimal constructions. Experience has shown that such preliminary incremental improvements will often be insignificant in the larger scope. Once the implementation is complete and the entirety of the code can be analyzed, however, it may be acceptable or necessary to revisit and improve those constructions that exhibit poor performance.

Performance-wise, severely suboptimal constructions have been used throughout the earlier FES algorithmic implementations. The code associated with the preliminary studies in Serpent was no exception. Initially, much of the code was algorithmically naive that contained a number of performance bottlenecks. A number of optimizations were discovered that unlocked significant improvements in computational efficiency. All future implementations of FE algorithms will benefit from these developments. This was critical to ensuring the future proliferation of multivariate FE implementations; heretofore, these were often too computationally intensive to justify adopting despite a superior representation fidelity.

The direct calculation of a series term is generally faster then any other algorithmic form. However, each term must be individually implemented. Recurrence relations, on the other hand, have the advantage of computing any arbitrary term of a series. The primary limitation is rooted in the floating-point accuracy of the computer system hardware architecture, through which minor inaccuracies can potentially pile-up over numerous repeated evaluations and produce false output. Further, for low-order terms, recurrence relation are comparatively expensive compared to direct calculation results. Some FE implementations used direct calculations, but were thereby limited the maximum series order that could be requested. Others used recurrence relations, but thereby incurred a relative performance penalty to support an unspecified maximum series order.

However, from a purely computational point of view, there is a trade-off that exists between the two when all terms in a series are to be computed in a single pass. Toward higher orders, eventually the term-wise direct calculation relations becomes so complex that it requires a significant number of floating point operations (flops) to calculate. Conversely, a recurrence relation requires only a fixed number of flops per term. In other words, a direct calculation approach will incur a quadratically increasing computational cost as $\mathcal{O}(N^2)$, while the recurrence relation cost will increase linearly as $\mathcal{O}(N)$.

8.3.1 Implementation

A hybrid approach was developed and implemented, in which the direct calculation form was used for lower-ordered terms. The calculation was then swapped over to the recurrence relation near the expected trade-off point.

It was also discovered that the method for convolving individual series together to create a multivariate function series could be greatly optimized. The hybrid approach was paired with a vector-based algorithm—similar in effect to memoization—in which all the lower-ordered terms were collected while calculating the highest-ordered desired term. This was done via fall-through enabled `switch-case` statements paired with a loop-based recurrence relation. Further, all these series evaluations could be pre-calculated before the actual convolution was performed. In other words: at each evaluation location, each individual series needed to be computed only once, and then the memoized output could be used in a highly-optimized convolution algorithm.

The hybrid-vector approach was implemented for both standardized and orthonormalized series evaluations. This necessitated the development of orthonormalized recurrence relation forms for the Legendre and Zernike polynomials; conversely, the direct computation relations were trivial to implement since all that was required was adjusting each term by the corresponding orthonormalization constant. However, this approach enabled seamless application in both the generation and reconstruction aspects of using fes. Finally, the implementation was designed to work in parallel computing situations leveraging OpenMP (omp) and/or message passing interface (mpi) technologies.

8.3.2 Benchmarking

Two types of benchmark evaluations were performed. The first benchmark tested the stand-alone speed of selected algorithmic implementations, including both the naive and final optimized versions. The second was a qualitative comparison in Serpent of just the original and vector-based FET algorithms; additional qualitative comparison was performed to compare the FET results to mesh tallies. All the Serpent tally runtimes were compared against a baseline simulation in which no tallies were collected. A complete effort was made to ensure that the hardware performed exactly for all the benchmarks.

The results of the standalone benchmarking are summarized in fig. 8.3, which shows the computational times as a function of the requested polynomial order of a cylindrical FET. This is a comprehensive benchmark of all the optimized methods because it includes all three of the Legendre, Zernike, and cylindrical convolution algorithms. The original line shows the results of the algorithms commonly used in most codes with FET implementations. The hybrid line shows the results of the hybrid series evaluation approach, but still using the typical convolution algorithm. Finally, the two vector lines—one for each of the standard and vector orthonormal forms—show the results of the optimizations using both hybrid series calculation and memoization-enabled convolution. It can clearly be seen that the vector-based cases are advantageous for nearly all expected use cases; the comparatively high computation time for orders 0 to 2 are due to the overhead associated with initializing the vector approach.

The results of the Serpent benchmarking are shown in table 8.1. The fine- and ultrafine-grade original algorithm FETS tests were not even run due to the incredibly long runtimes that they would have required; the results were already enough to confirm that the optimized algorithms were much more efficient.

An initial comparison of the FET and mesh tally results in table 8.1 appear to indicate a computational favor toward the mesh tallies. However, another critical piece of information needs to be considered: the quality of each tally. As shown in fig. 8.4, even just the coarse FET exhibits superior quality vs. runtime characteristics compared to the mesh tallies.

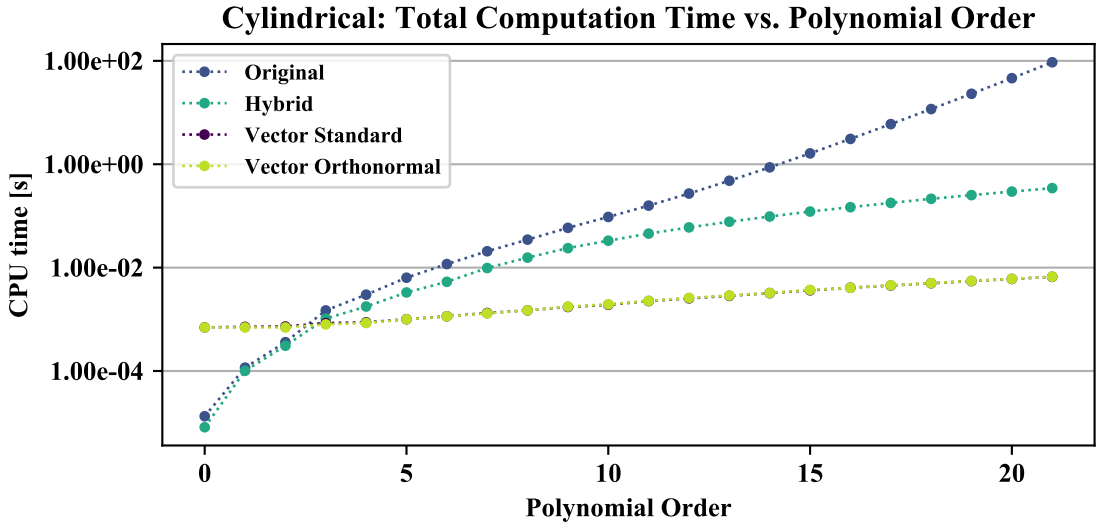
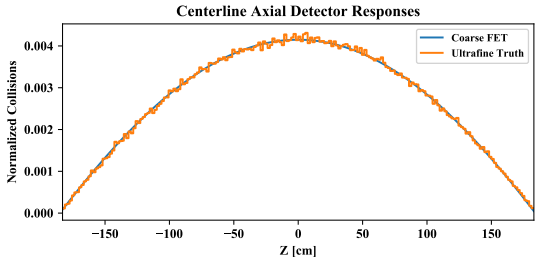


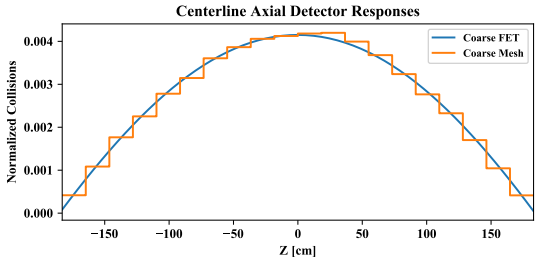
Figure 8.3 – Cylindrical polynomial basis benchmarking results. This figure is reproduced here from fig. 8.3 for clarity.

Table 8.1 – Serpent benchmark results: total average trial duration, percent increase with respect to the baseline, and qualitative results for the Cartesian tallies. This table is reproduced here from table 4.5.

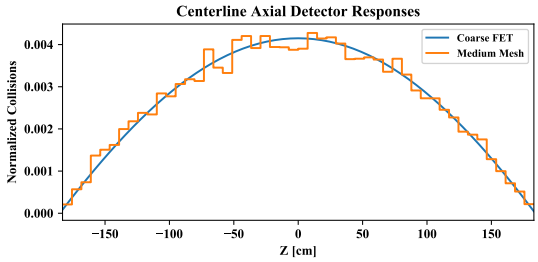
		Cylindrical		Cartesian	
Type	**Grade**	**Duration [min]**	**Change**	**Duration [min]**	**Change**
Baseline		7.83		8.01	
Mesh	*Coarse*	8.39	7.13 %	8.32	3.89 %
	Medium	8.45	7.89 %	8.36	4.39 %
	Fine	8.43	7.64 %	11.5	43.3 %
	Ultrafine	8.70	11.1 %	13.4	67.0 %
Vector FET	*Coarse*	9.70	23.9 %	9.76	21.8 %
	Medium	12.4	58.9 %	15.3	90.5 %
	Fine	15.6	98.8 %	28.5	266 %
	Ultrafine	26.4	237 %	95.9	1100 %
Original FET	*Coarse*	9.88	23.3 %	9.79	22.2 %
	Medium	17.16	114 %	29.7	270 %



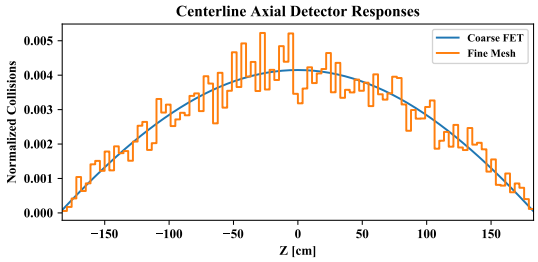
(a) Coarse FET (19.5 min) vs. Ultrafine 'truth' mesh using 4.35×10^9 histories (66 400 min)



(b) Coarse FET (19.5 min) vs. coarse mesh (16.6 min)



(c) Coarse FET (19.5 min) vs. medium mesh (16.7 min)



(d) Coarse FET (19.5 min) vs. fine mesh (23.0 min)

Figure 8.4 – Comparison between the coarse FET and other various tallies. All tallies are collected over 10×10^6 histories unless otherwise noted. The values in parenthesis are the CPU minutes required to generate each tally, calculated as the product of thread count and the runtime. This figure is reproduced here partially from fig. 4.5.

8.4 FET Runtime Convergence

FETs are known to have a superior convergence rate in sample space [76]. This means that an FET will converge faster per-sample than a mesh tally. However, it is also know that FETs take longer to evaluate per-sample than a mesh tally. The primary investigation then becomes a comparison in computation time space. The objective is to determine if the faster convergence rate of an FET is sufficient to overcome the longer per-sample computation time.

8.4.1 Principles

Quantitatively comparing the runtime convergences requires a firm measure of the convergence rate for each tally type. In MC contexts, the figure-of-merit (FOM) is one such measure. It balances the uncertainty of a tally against the runtime, simply calculated as:

$$\text{FOM} = \frac{1}{Ut} \tag{8.5}$$

Table 8.2 – Final FOMs of the tally types for each benchmark model, organized by granularity. This table is reproduced here from table 5.3.

Granularity	Type	Unreflected	Reflected
Coarse	FET	$3.57 \times 10^{+6}$	$9.80 \times 10^{+6}$
	Mesh	$6.70 \times 10^{+5}$	$3.33 \times 10^{+6}$
Fine	FET	$6.92 \times 10^{+5}$	$1.16 \times 10^{+6}$
	Mesh	$2.99 \times 10^{+4}$	$1.19 \times 10^{+5}$

recreated here from eq. (5.10) for clarity. A higher FOM is better, as it indicates a smaller uncertainty per unit time.

Providing a FOM-based characterization requires a comparable uncertainty measure for both FETs and mesh tallies. However, the uncertainty values of a mesh tally correspond to the bin locations, while the uncertainty values of an FET correspond to the coefficients themselves. Thus, a total relative uncertainty measure U was derived in ℓ^2 space for each tally type. This relative uncertainty measure relates the total information provided by the tally values against the combined uncertainty over the entire tally domain. The resulting expressions for each tally type are:

$$U_{\text{FET}} = \frac{\sum \frac{\widehat{\sigma}^2_{\widehat{b}_i}}{c_i}}{\sum \frac{\widehat{b}^2_i}{c_i}} \tag{8.6}$$

$$U_{\text{mesh}} = \frac{\sum \sigma_b^2 V_b}{\sum v_b^2 V_b} \tag{8.7}$$

recreated here from eqs. (5.6) and (5.9) for clarity.

8.4.2 Testing

Testing was performed in Serpent using two different International Criticality Safety Benchmark Evaluation Project (ICSBEP) experiments; although the fissile material was identical, one setup was unreflected and the other was encased in a Plexiglass reflector. Two mesh tallies and two FETs were evaluated, each at a coarse and fine grade. The FOM results are shown in table 8.2. It demonstrates that FETs do actually have a better convergence rate in computational runtime than the mesh tallies.

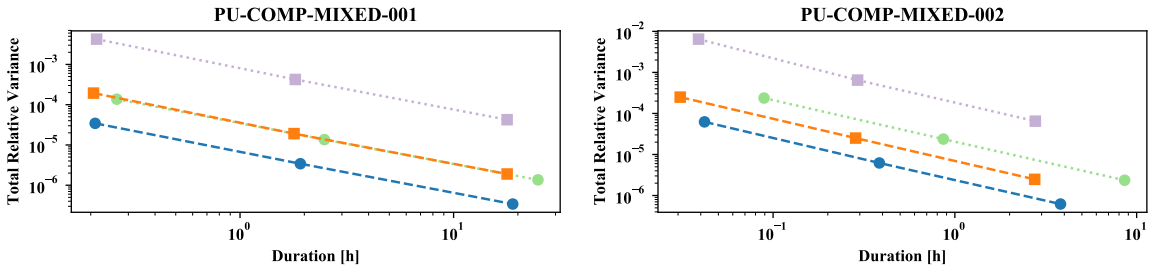


Figure 8.5 – Comparison of the total relative variances as a function of trial runtime (both axes are logarithmically scaled). This figure is reproduced here from fig. 5.2.

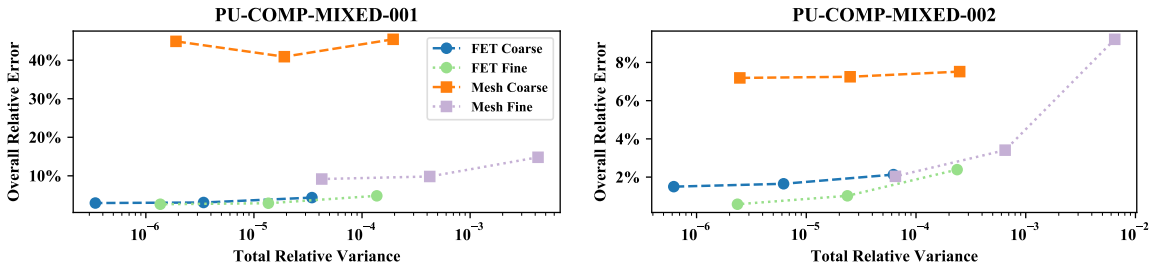


Figure 8.6 – Relationship between the total relative variances and the overall relative errors for the different trials (the horizontal axis is logarithmically scaled; the legend is common for figs. 8.5 and 8.6). PU-COMP-MIXED-001 is unreflected and PU-COMP-MIXED-002 is reflected. This figure is reproduced here from fig. 5.3.

Additional qualitative results are shown in figs. 8.5 and 8.6, which provide a comparison of the relationships between computational runtime, total relative uncertainty, and overall relative error. In all plots, the lower-left hand corner is the region of optimal performance. The FETs consistently have a lower overall relative error than the mesh tallies at the same duration and/or total relative uncertainty. This proves that, under normal circumstances, FETs have a superior convergence rate and fidelity than a comparable mesh tally.

8.5 Generalized Coupling Interface

In explicitly coupled simulations, model similarity between the two or more separate codes is not guaranteed—in fact, model dissimilarity is essentially guaranteed. This is especially true when coupling a finite element analysis code like the MOOSE framework to an MC code like Serpent. MOOSE, as a finite element code, contains a large number of mesh elements over which one or more differential equations are solved. Serpent, as a reactor physics

MC code, uses constructive solid geometry (CSG) to define a model in which the stochastic process transpires.

These two model definition approaches are not inherently compatible with each other for communicating multiphysics data. However, FE-based methods are agnostic to the underlying model construction. Instead, FEs rely solely on the domain boundary specifications within which they operate. That means that FE-based methods can be generally applied to multiphysics coupling simulations among numerous codes sharing different constructions.

MOOSE was designed with a great level of flexibility, allowing non-MOOSE codes to be integrated into a MOOSE-based simulation through the MultiApp system. No assumption is made about the type or format of these external codes; instead, the integration challenge is left in the hands of the user. This establishes MOOSE as a strong candidate for a primary multiphysics driver. A module to the MOOSE framework was identified as the best option for integrating FE methodologies and making them available any interested parties. To avoid confusion with 'FE' for finite element, the term 'FX' was used in the class names of this MOOSE module.

8.5.1 Implementation

A high-level design architecture of the `functional_expansion_tools` module is shown in fig. 8.7.

The primary workhorse of the module is the `FunctionSeries` class. It is derived from the MOOSE class `Function`, allowing it to be used in any related context. It also contains a memoization capability. This can be used to cache the location-specific evaluation results and recall them (as needed) to improve performance.

A concrete `CompositeSeriesBasisInterface` implementation, such as `Cartesian` or `CylindricalDuo`, provides the geometry-specific multivariate convolution algorithms necessary for working with FEs as a high level. In turn, each `CompositeSeriesBasisInterface`-derived class depends on one or more concrete `SingleSeriesBasisInterface` instances. These `SingleSeriesBasisInterface`-derived classes provide the actual individual functions such as the Legendre or Zernike polynomial series. The use of these interfaces facilitates

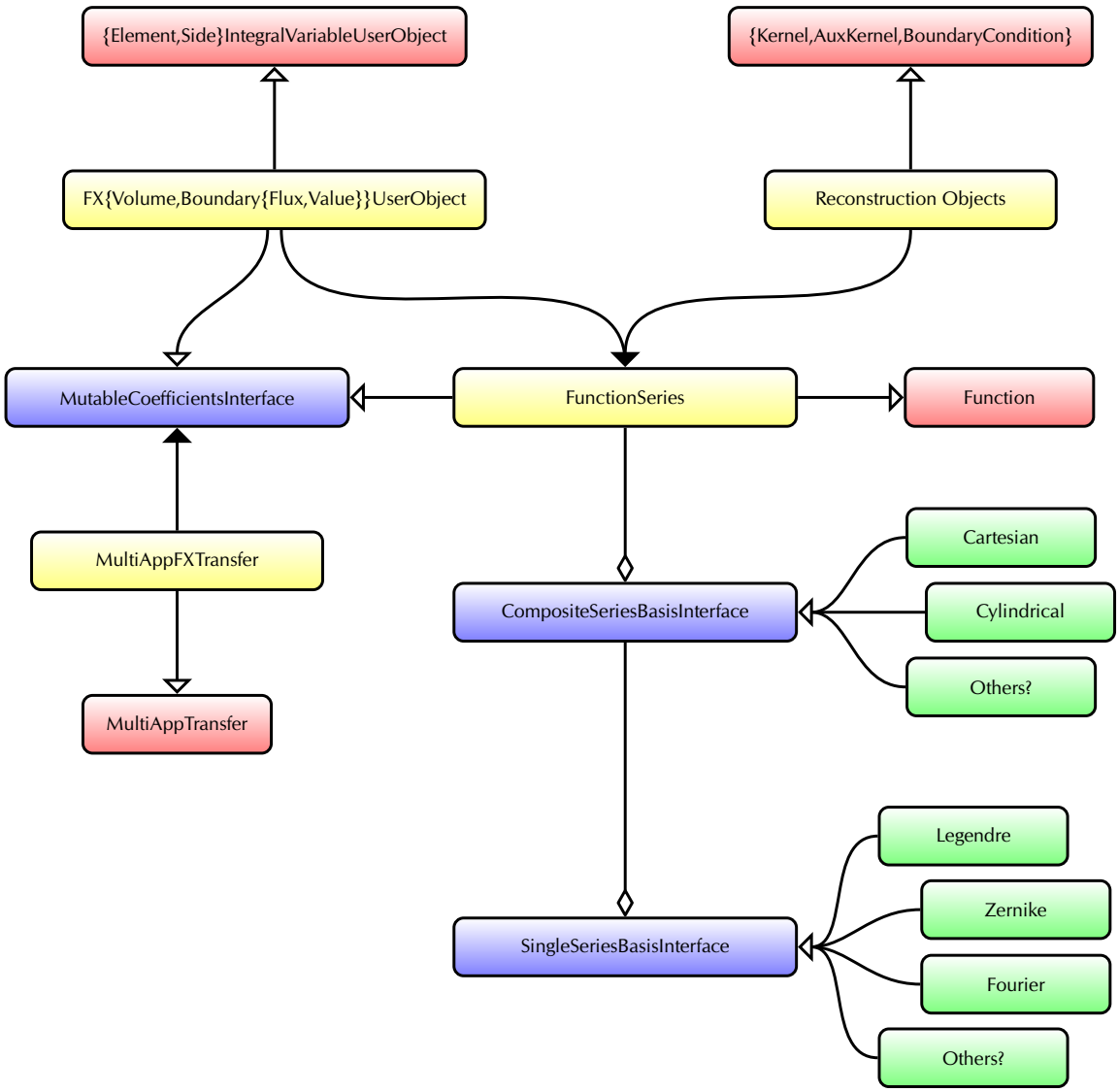


Figure 8.7 – MOOSE FE module architecture. The boxes are colored based on class type: red = existing MOOSE classes; blue = new interface definitions; yellow = external concrete classes available to users; green = internal concrete classes. The tips represent the relationships: filled arrow = a requires/uses association, open arrow = inheritance; diamond = composition. This figure is reproduced here from fig. 6.1.

the future addition of any individual series or convolutions. All operational parameters are specified via the MOOSE input file.

Next, the UserObject-based classes provide the algorithms required to generate FE coefficients. A policy-based design allows for FEs to be generation for any scalar distribution that can be measured by MOOSE UserObjects. This includes properties such as temperature, density, concentration, energy, power, speed, pressure, dose, and luminosity. The policies initially provided with the module include capabilities to represent: 1) a variable field value distribution in a volume. 2) a variable field value distribution at a boundary, or 3) a variable field flux distribution across a boundary interface. Each FE-enabled UserObject requires a `FunctionSeries` object.

FE reconstruction currently uses MOOSE `AuxKernel`-, `BC`-, and `Kernel`-based classes. These are used to expand an FE into a variable's solution. These, too, depend on a `FunctionSeries` object to provide access to the underlying functional expansion series.

Underlying everything is `MutableCoefficientsInterface`, from which most FE-enabled classes inherit. First and foremost, it is the storage container for FE coefficients. A mechanism for storing the characteristics of the stored FE is also provided. The coefficients themselves are transferred between compatible `MutableCoeffientInterface` objects by `MultiAppFXTransfer`. The characteristics of each FE involved in a transfer are checked against each other for compatibility first.

Finally, these classes were designed to be compatible with parallel computing situation using OMP and/or MPI technologies.

8.5.2 Testing

Both the volumetric- and boundary-based FE couplings were tested in MOOSE using tightly-coupled simulations utilizing Picard iterations to converge the solutions. These tests were respectively compared against ideal fully-coupled solutions. Testing with a few other data representation approaches, commonly used for multiphysics coupling, were also performed for enhanced comparison.

The volumetric tests were performed in a transient temperature-heat generation feedback system; explicitly, a higher localized temperature resulted in a lower localized heat generation. This effect was an empirical approximation of the thermal feedback effects in a nuclear reactor. A few other characteristics were included, designed to test the capabilities of an FE-coupled multiphysics simulation. These included varying the series' orders and altering the underlying mesh in one of the MultiApps.

First, a volumetric Cartesian test was performed. The purpose was to demonstrate that FE-based couplings performed similarly, if not better, than other coupling methodologies. It was also vital to demonstrate that the `Cartesian` implementation actually worked. Figure 8.8 compares the results of a few different coupling methods at a simulation time of 3.5 s. From this it can be understood that the FE-based multiphysics coupling implementation in MOOSE works. Further, the results are very similar to one another. It is important to note that the mesh-mesh transfer required the exchange of 500 000 values for each transfer, whereas the FE transfers exchanged most 432 values. A nonidentical mesh test was also performed; in this, the heat generation was solved over a mesh with a completely different construction than the temperature solution mesh. No alteration of the FE-related parameters was performed. In all cases, the tests containing the nonidentical mesh solutions performed similarly or better than the original simulations when compared to the ideal fully-coupled results.

Second, a volumetric cylindrical test was performed. The primary purpose was to demonstrate the `CylindricalDuo` implementation actually worked, and secondarily that FEs were compatible for use in simulations in which runtime adaptive mesh refinement techniques were used. The simulation used a similar heat generation model, but this time the finite element shapes were different; again, no alteration of the FE-related parameters or code was performed. Boundary conditions were also applied to influence the solution field near the outside edges of the model. Enabling mesh adaptivity actually worked to improve the accuracy of the FE-based coupling methods; again, note that no alteration of the FE code was required. However, adding the boundary conditions resulted in a steep gradient in the variable field at the edge of the model. This challenged the range of the radial Zernike polynomials, so a 'flattening' data transformation was introduced and tested as well.

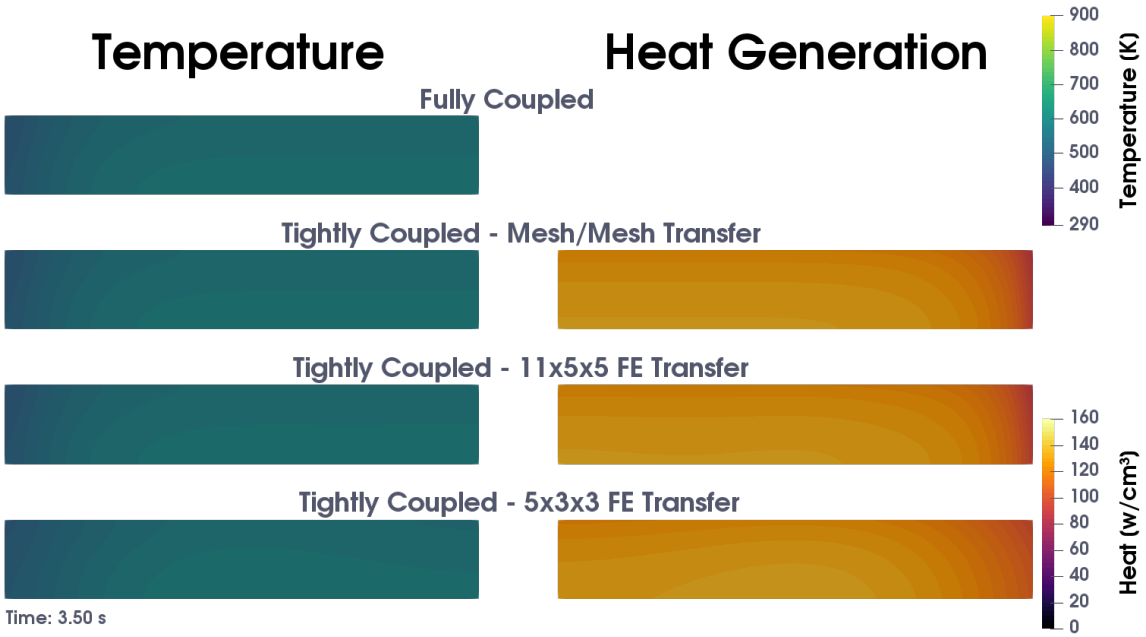


Figure 8.8 – Comparison of volumetric coupling test results between differing methodologies. Shown are 2D axial cross sections along the fuel element center, divided symmetrically to illustrate the relationship between temperature and heat generation. This figure is reproduced here from fig. 6.3a.

Figure 8.9 compares the results of a mesh-mesh direct copy to the FE-based couplings. As can be seen, the FE-coupled solution is essentially identical to the mesh-mesh copy. This demonstrated both the correctness of the `CylindricalDuo` implementation as well as the data flattening technique.

Third, a 1D boundary in a 2D problem was tested. The purpose was to demonstrate that the boundary FE-based coupling methods worked. This was done by splitting the model into two regions with district material properties. Heat was generated in one of the material regions, the fuel, while the other acted like a large water heat sink. Both temperature and heat flux were matched at the interface between the two. Figure 8.10 compares the results of the tightly-coupled FE-based boundary coupled simulation against the ideal fully-coupled solution. The solutions of the two are essentially identical, again confirming the validity of the boundary coupling methods.

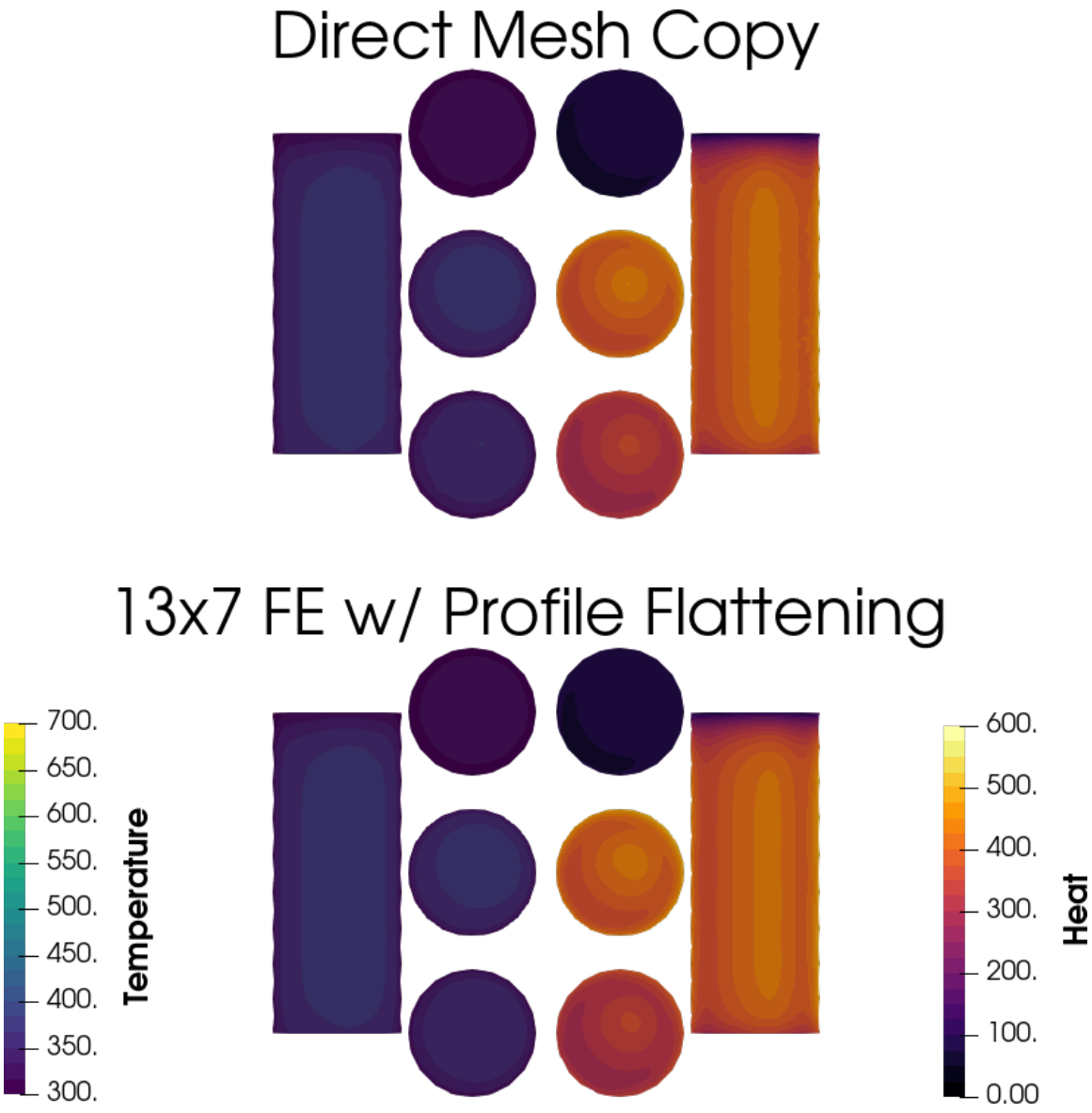


Figure 8.9 – Visual comparison between the Direct and flattened data FE coupling results, at the time of highest heat generation. On the outside edges are 2D axial slices of the field profiles. In the center are 2D radial slices of the field profiles at the bottom, middle, and top of the axial lengths. This figure is reproduced here from fig. 6.5.

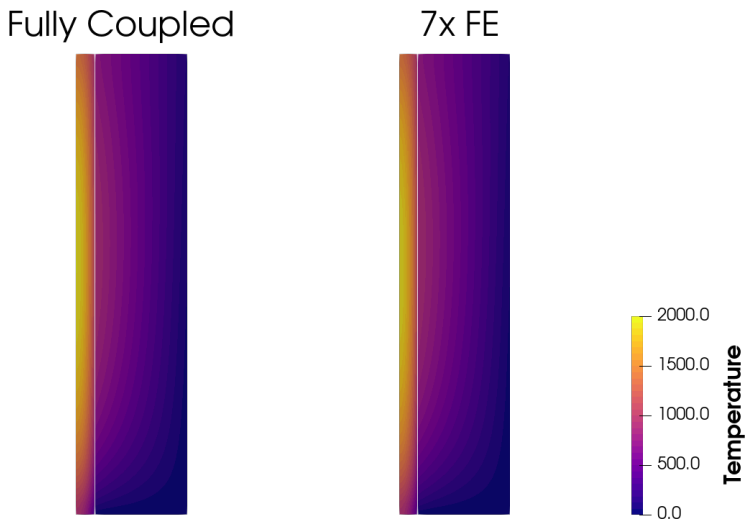


Figure 8.10 – Comparison of boundary coupling test results between differing methodologies. A thin white line marks the MultiApp interface between the fuel (left) and the water (right). This figure is reproduced here from fig. 6.7.

8.6 Multiphysics Demonstration

The demonstration of a 3D multivariate FE-based coupling is the culmination of this work; as could be expected, the two codes used are MOOSE and Serpent. This coupling demonstration was built upon the FE code optimizations, as the implementations in both codes utilized the advanced optimized algorithms.

8.6.1 Implementation

First, Serpent was integrated into MOOSE as a MOOSE-wrapped `MooseApp` named `chrysalis`. This included the development of a set of build tools to copy, modify, and build the Serpent source files as a library internal to MOOSE.

Second, the FET routines already integrated into Serpent as a detector option were expanded and ported to Serpent's multiphysics interface. This was incrementally tested and determined to be functional by enforcing a temperature variation through carefully crafted sets of FE coefficients. One example of the plotted output generated by Serpent is shown in fig. 8.11.

Third, the required elements for an FE-based MOOSE driver were developed. The two required components in MOOSE are an `Executioner` and matching `TimeStepper` classes. A

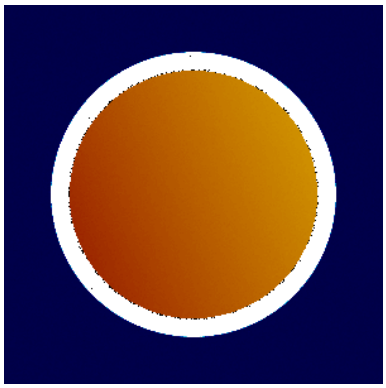


Figure 8.11 – Top-view plot of an FE-based continuously-varying temperature field in Serpent for a representative AP-1000 fuel pin. This figure is reproduced here from fig. 7.1b.

general `FXExecutioner` was created, which inherits from `MutableCoefficientsInterface` so that it supports coefficient transfers. `MultiAppFXTransfer` was also updated to support transfers with `FXExecutioner`-derived classes. Next, coefficient relaxation methodologies and rejection criteria were added to `FXExecutioner` to aid in the running of a multiphysics simulation.

Fourth, new `SingleSeriesBasisInterface` and `CompositeSeriesBasisInterface` classes were developed within MOOSE. These were `Fourier` and `AxialShell`, respectively. The `AxialShell` convolution used Fourier series to construct a radius-invariant FE for use with boundary conditions on a cylinder constructed from a finite element mesh. The primary motivation was to provide compatibility with the radial variations induced by representing curved surfaces using finite element shapes with linear edges.

Finally, a Serpent-specific `SerpentExecutioner` was created based on `FXExecutioner`. It is responsible for transferring coefficient data via Serpent's file-based interface. A specialized `TimeStepper` class, `SerpentTimeStepper`, was also created. It is responsible for initializing the Serpent interface and running a solution step via access to the methods of the compiled Serpent library.

8.6.2 Testing

The newly-created functional expansion module in MOOSE was leveraged fully in the 3D simulation of a boiling-water reactor (BWR) fuel pin. Three MultiApps were created, one

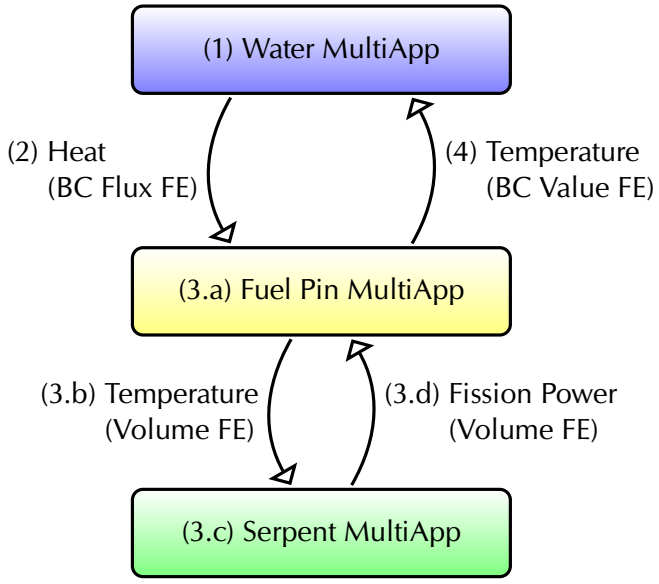


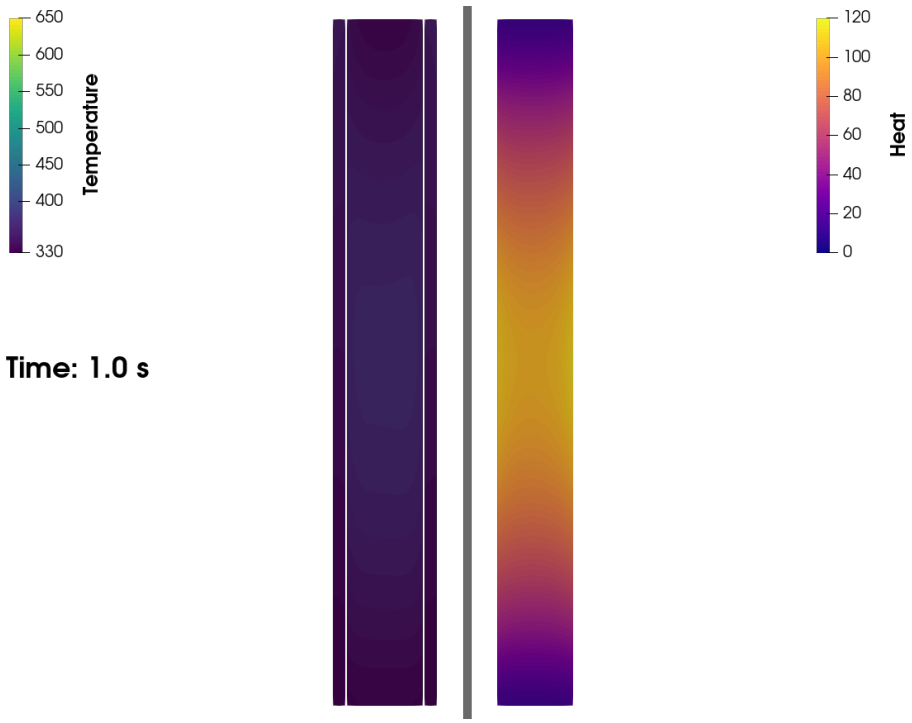
Figure 8.12 – MultiApp hierarchy of the test simulation for the MOOSE-Serpent coupling. Steps 3.a–d represent an inner iteration that was converged using Picard iterations, and then steps 1–4 are part of an outer iteration cycle that was also converged using Picard iterations. This figure is reproduced here from fig. 7.3.

each for water, fuel, and neutronics. The water and fuel MultiApps each solved heat transfer within their domains, while the Serpent-based neutronics performed the heat generation calculations. As shown in fig. 8.12, the water and fuel MultiApps were coupled at the interface between the two using FE-based boundary conditions, while the fuel and neutronics MultiApps were coupled using FE-based volume data representations

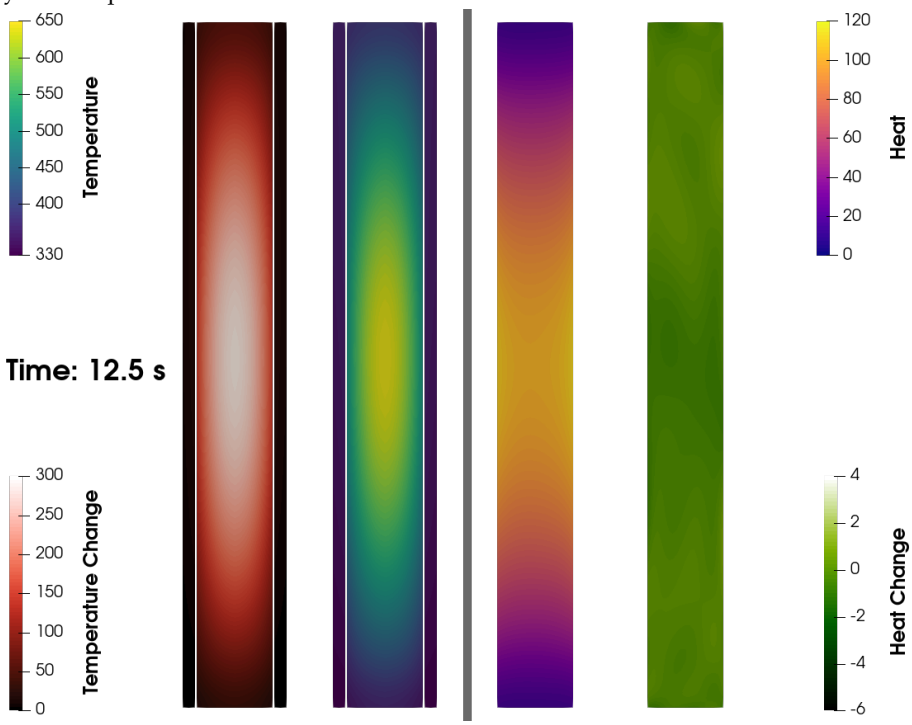
The simulation comprised a transient-type simulation that was run for 12.5 s from an initial cold state of 330 °C. Although very unrealistic for startup from a cold state, the power was assumed to be produced at a total value of 30 kW. Figure 8.13 shows the output at steps of 1.0 s and 12.5 s during this simulation. As can be seen, the increased central temperature at 12.5 s has caused the neutronics simulation to slightly flatten out axially compared to the colder solution at 1.0 s.

8.7 Outcome

In conclusion, this work has accomplished its proposed objectives. Significant contributions have made to advance the performance of FE-based methods. Most or all can likely be



(a) Multiphysics fuel pin simulation at time $t = 1.0\,\mathrm{s}$



(b) Multiphysics fuel pin simulation at time $t = 12.5\,\mathrm{s}$, showing differences compared to the simulation at time $t = 1.0\,\mathrm{s}$

Figure 8.13 – Results of the MOOSE-Serpent coupled multiphysics simulation, at 0.03 axial scale.

applied in any implementation, and in most cases they reduce the required computational time per evaluation by 80 % or more. The convergence rate of FETs has been measured and quantified, and they have been demonstrated to have a faster convergence rate and higher fidelity than comparable mesh tally methods. A generalized functional expansion module, `functional_expansion_tools`, was developed and integrated into the MOOSE framework. This expands the multiphysics driver features of MOOSE to include inherent support for communicating with all FE-based codes. Finally, this module was expanded and tested by successfully coupling to the Serpent MC reactor physics code. Fully-multivariate 3D-capable FE methodologies are used throughout.

Chapter 9

Future Work

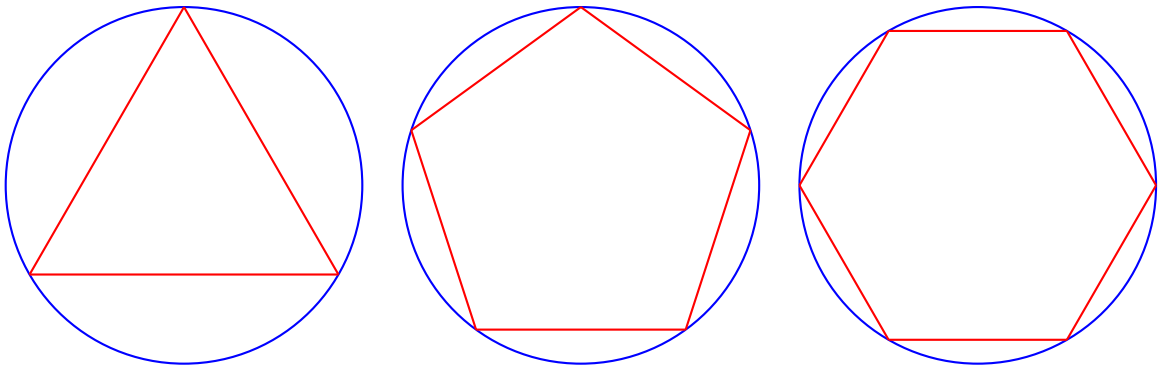
Every new beginning comes from some other beginning's end.

— Seneca

Nothing opens the mind more to the absence of knowledge than delving deeply into a specific subject. Such is the case of this research in FES; although many solutions were found and developed, a number of additional avenues for future research and development (R&D) were discovered.

9.1 Supporting Geometries without a Matching Functional Basis

A strength of FE-based methods is the pure mathematical underpinning; the methodology can be intrinsically applied to any geometrical construct supported by an available functional series. For example, multidimensional Cartesian domains can be represented using convolutions of the 1D Legendre and linear Fourier series. Adding in the 2D Zernike polynomial series expands the domain options to include 3D cylindrical spaces. Such a multivariate functional series is generated by convolving the Zernike polynomials, which provide the radial and azimuthal dimensions, with a 1D series, which provides the axial dimension. Finally, although not used in this work, the 3D spherical harmonics series could be applied as-is to spherical domains.



(a) Inscribed triangle with 41.3 % utilization

(b) Inscribed pentagon with 75.7 % utilization

(c) Inscribed hexagon with 82.7 % utilization

Figure 9.1 – Polygonal regions inscribed in a circular region, showing how much of the subtended space is actually occupied by the polygonal region.

However, this intrinsic characteristic could potentially break down when applied to geometrical constructions that are not hexahedral, cylindrical, nor spherical. One example would be working with hexagonal or triangular shapes, which do not fit conveniently into any of these categories. An option would be to evaluate the FE over a larger hexahedral or cylindrical domain that circumscribes the desired region. Figure 9.1 shows a 2D example of standard polygonal shapes inscribed in a simple circular region; such an approach would be equivalent to using the Zernike polynomials to represent data in the subtended region.

One likely detraction is the lack of coverage by the subtended region. For example, as shown in fig. 9.1a for a triangular region, at most 41.3 % of the domain provided by circumscribing FE would be used. It is highly likely that such a low utilization would negatively impact the accuracy of the FE coefficient generation results, regardless of whether an integrated or MC-sampled approach was used. Nevertheless, this method could be applied to an arbitrarily-defined region—given that it is fully inscribed within the FE domain.

Another option would be to pursue a conformal mapping technique, as illustrated by fig. 9.2. In this approach, a well-defined region is mathematically mapped into another region that may be natively supported by an existing functional series. Angles and other essential data are preserved through such mapping, ensuring that information is not lost in the translation. For example, using the mappings provided in Mathews and Howell [120], a

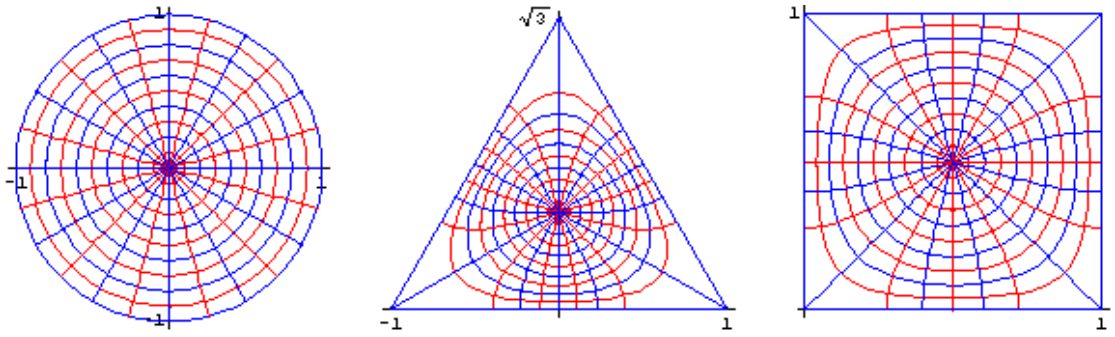


Figure 9.2 – Conformal mapping possibilities, showing the same data mapped between a circle, triangle, and square regions. This image is reproduced from Mathews and Howell [120].

equilateral triangular region could be mapped to either a circular or square region. The downside is that the transformation routines may be computational expensive.

Future research could provide insight on the degree to which using a subtended region would impact the accuracy of an FE coefficient generation routine. Further, in the case of an arbitrary region being circumscribed by the FE, this approach is likely the only solution. Quantification of the FE accuracy degradation would be especially useful in such circumstances. Conversely, conformal mapping may be optimal if the region is well-defined but doesn't have a supporting functional series basis. A quantification of the computation burden associated with the transformation algorithms would be required. Using results from this potential research, an implementor of FE-based technologies could make an informed decision on which approach would be optimal for their particular use case.

9.2 FE Coefficient Filtering

In MC-based FE generation processes, the stochastic uncertainty of each individual coefficient is often weighed against its actual value to determine the total worth. Currently, all filtering implementations use a fixed cut-off value: if the ratio between the uncertainty and value is too high, then the coefficient is discarded or ignored. However, this approach provides very little respect for the actual relative contribution of each individual coefficient. Additionally, these cut-off values are not universal and will need adjusted for each situation.

One avenue for future research would be to investigate the application a Bayesian filter to evaluate the best-possible set of coefficients. This would require a post-processing step, but could be very useful for maximizing the effective accuracy of each stochastically-generated FE. Conceptually, the uncertainty-value ratio for each coefficient would be calculated, then the resulting values would be sorted from smallest to largest. The Bayesian filter would then process the sorted list and identify the optimal subset that minimizes the uncertainty contribution while maximizing the actual information provided. This would provide a filter that responds dynamically to changing stochastic sampling conditions between differing Picard iterations, yet would remain inherently suitable for nearly any simulation.

A second avenue would be to investigated the utilization of pre-determined truncations for a fully-convolved multivariate functional series. Essentially, in multivariate expansions the relevance of the higher-order cross terms becomes statistically unlikely. Various truncation strategies could be parametrically studied and evaluated to determine the impact on the overall FET solution. One option would be to clip all cross terms for which the sum of the corresponding orders exceeds a certain value. Another similar approach would be to clip based on the product of corresponding orders.

9.3 Support for Multiple Expansion Sets

Currently, the `MutableCoefficientsInterface` support only a single FE coefficient set per instance. In pure MOOSE simulations this is not generally an issue, as separate `FunctionSeriesFX...UserObject` pairs can be easily defined for each individual need. A corresponding `Transfer` would be set up to move the coefficient data as required for the simulation. The primary challenge would be keeping everything organized in large simulations utilizing a large number of FEs.

However, a challenge can arise when coupling to external codes. For example, a Serpent fuel-pin lattice calculation will often contain tens or hundreds of fuel pins. In an ideal high-fidelity multiphysics simulation, each fuel pin would be represented by its own individual FET. If coupled to MOOSE, all these FETs would be dumped to the

multiphysics interface output file to be read in by `SerpentExecutioner`. Unfortunately, as a subclass of `MutableCoefficientsInterface` only one of these coefficients could be stored and transferred into MOOSE. In fact, this situation is already detected in `SerpentExecution`, and an error is emitted if multiple FETs are found in the interface output file. This causes the step to fail and the simulation to stop.

Future R&D could expand the capabilities of `MutableCoefficientsInterface` to store an arbitrary number of FE coefficient sets. These could be distinguished by name, or even something as simple as an integer index. To become more robust and user-friendly, a new mechanism could be developed in MOOSE that would allow for these sets to be enumerated and automatically assigned to the correct blocks in a large multiphysics simulation.

9.4 Mathematical Basis for Data Flattening

A technique was applied to flatten a data distribution for inter-MOOSE FE-based couplings. By assuming that no negative values were present in the field values, the input distribution could be flattened using an exponential function. This flattening was performed prior to the FE generation routine. Later, when the FE was reconstructed, the resulting value was inflated using the inverse of the original exponential function.

Although intuitively complete, the actual mathematical basis for such an approach was not attempted. Thus, this technique would benefit from a robust mathematical treatment to prove its validity and applicability.

9.5 MC Iteration Skipping

When used as part of tightly-coupled multiphysics simulations, MC solves are often the most time consuming component. Often, not much can be done to reduce the solve time of each MC run without causing a significant increase in statistical uncertainty. Thus, mechanisms have been sought to decrease the number of MC solves that are performed during the iterative convergence process. One proposed option is to skip a specified set of MC solves, instead reusing the previous results for in the solution of the other coupled code(s).

This MC solve skipping technique was implemented in `FXExecutioner` and leveraged to reduce to total runtime of the initial simulations. With this capability already in place, there is a great need to parametrically evaluated and quantitatively study the impact of skipping MC solved on the final solution. Ideally, a set of generalized guidelines could be established to guide others as the determine whether to apply this technique for convergence acceleration.

Appendices

Appendix A

Orthonormal Function Formulae

Formulae for orthogonal functional bases are plentiful in literature, but the orthonormal variants must generally be created from an expression for the norm. The following tables provide the orthonormal variants used for the optimized FE algorithms in both MOOSE and Serpent.

Table A.1 – A compilation of the first twelve orthonormal Legendre polynomial formulae with the orthonormalization constant c calculated from eq. (3.25).

Order [k]	ω	Orthonormal Formula
0	$\frac{1}{2}$	$\frac{1}{2}$
1	$\frac{3}{2}$	$\frac{3}{2}x$
2	$\frac{5}{2}$	$\frac{5}{4}\left(3*x^2-1\right)$
3	$\frac{7}{2}$	$\frac{7}{4}\left(5*x^2-3\right)*x$
4	$\frac{9}{2}$	$\frac{9}{16}\left(\left(35*x^2-30\right)*x^2+3\right)$
5	$\frac{11}{2}$	$\frac{11}{16}\left(\left(63*x^2-70\right)*x^2+15\right)*x$
6	$\frac{13}{2}$	$\frac{13}{32}\left(\left(\left(231*x^2-315\right)*x^2+105\right)*x^2-5\right)$
7	$\frac{15}{2}$	$\frac{15}{32}\left(\left(\left(429*x^2-693\right)*x^2+315\right)*x^2-35\right)*x$
8	$\frac{17}{2}$	$\frac{17}{256}\left(\left(\left(\left(6435*x^2-12012\right)*x^2+6930\right)*x^2-1260\right)*x^2+35\right)$
9	$\frac{19}{2}$	$\frac{19}{256}\left(\left(\left(\left(12155*x^2-25740\right)*x^2+18018\right)*x^2-4620\right)*x^2+315\right)*x$
10	$\frac{21}{2}$	$\frac{21}{512}\left(\left(\left(\left(\left(46189*x^2-109395\right)*x^2+90090\right)*x^2-30030\right)*x^2+3465\right)*x^2-63\right)$
11	$\frac{23}{2}$	$\frac{23}{512}\left(\left(\left(\left(\left(88179*x^2-230945\right)*x^2+218790\right)*x^2-90090\right)*x^2+15015\right)*x^2-693\right)*x$
12	$\frac{25}{2}$	$\frac{25}{2048}\left(\left(\left(\left(\left(\left(676039*x^2-1939938\right)*x^2+2078505\right)*x^2-1021020\right)*x^2+225225\right)*x^2-18018\right)*x^2+231\right)$

All numbers are shown here in rational form, but can be converted to decimal representations in-code to save FLOPS

Table A.2 – A compilation of the first ten orders of the orthonormal Zernike polynomials formulae with the orthonormalization constant c calculated from Equation (3.58).

Order [n]	**Rank [m]**	ω	**Formula**
0	0	$\frac{1}{\pi}$	$\frac{1}{\pi}$
1	-1	$\frac{4}{\pi}$	$\frac{4}{\pi} r \sin(\phi)$
1	1	$\frac{4}{\pi}$	$\frac{4}{\pi} r \cos(\phi)$
2	-2	$\frac{6}{\pi}$	$\frac{6}{\pi} r^2 \sin(\phi)$
2	0	$\frac{3}{\pi}$	$\frac{3}{\pi} \left(2r^2 - 1\right)$
2	2	$\frac{6}{\pi}$	$\frac{6}{\pi} r^2 \cos(\phi)$
3	-3	$\frac{8}{\pi}$	$\frac{8}{\pi} r^3 \sin(3\phi)$
3	-1	$\frac{8}{\pi}$	$\frac{8}{\pi} \left(3r^2 - 2\right) r \sin(\phi)$
3	1	$\frac{8}{\pi}$	$\frac{8}{\pi} \left(3r^2 - 2\right) r \cos(\phi)$
3	3	$\frac{8}{\pi}$	$\frac{8}{\pi} r^3 \cos(3\phi)$
4	-4	$\frac{10}{\pi}$	$\frac{10}{\pi} r^4 \sin(4\phi)$
4	-2	$\frac{10}{\pi}$	$\frac{10}{\pi} \left(4r^2 - 3\right) r^2 \sin(2\phi)$

continued . . .

. . . continued

Order [n]	Rank [m]	ω	Formula
4	0	$\frac{5}{\pi}$	$\frac{5}{\pi}\left(\left(6r^2-6\right)r^2+1\right)$
4	2	$\frac{10}{\pi}$	$\frac{10}{\pi}\left(4r^2-3\right)r^2\cos(2\phi)$
4	4	$\frac{10}{\pi}$	$\frac{10}{\pi}r^4\cos(4\phi)$
5	-5	$\frac{12}{\pi}$	$\frac{12}{\pi}r^5\sin(5\phi)$
5	-3	$\frac{12}{\pi}$	$\frac{12}{\pi}\left(5r^2-4\right)r^3\sin(3\phi)$
5	-1	$\frac{12}{\pi}$	$\frac{12}{\pi}\left(\left(10r^2-12\right)r^2+3\right)r\sin(\phi)$
5	1	$\frac{12}{\pi}$	$\frac{12}{\pi}\left(\left(10r^2-12\right)r^2+3\right)r\cos(\phi)$
5	3	$\frac{12}{\pi}$	$\frac{12}{\pi}\left(5r^2-4\right)r^3\cos(3\phi)$
5	5	$\frac{12}{\pi}$	$\frac{12}{\pi}r^5\cos(5\phi)$
6	-6	$\frac{14}{\pi}$	$\frac{14}{\pi}r^6\sin(6\phi)$
6	-4	$\frac{14}{\pi}$	$\frac{14}{\pi}\left(6r^2-5\right)r^4\sin(4\phi)$
6	-2	$\frac{14}{\pi}$	$\frac{14}{\pi}\left(\left(15r^2-20\right)r^2+6\right)r^2\sin(2\phi)$
6	0	$\frac{7}{\pi}$	$\frac{7}{\pi}\left(\left(\left(20r^2-30\right)r^2+12\right)r^2-1\right)$

continued . . .

. . . continued

Order [n]	Rank [m]	ω	Formula	
6	2	$\frac{14}{\pi}$	$\frac{14}{\pi}\left(\left(15r^2-20\right)r^2+6\right)r^2\cos(2\phi)$	
6	4	$\frac{14}{\pi}$	$\frac{14}{\pi}\left(6r^2-5\right)r^4\cos(4\phi)$	
6	6	$\frac{14}{\pi}$	$\frac{14}{\pi}r^6\cos(6\phi)$	
7	-7	$\frac{16}{\pi}$	$\frac{16}{\pi}r^7\sin(7\phi)$	
7	-5	$\frac{16}{\pi}$	$\frac{16}{\pi}\left(7r^2-6\right)r^5\sin(5\phi)$	
7	-3	$\frac{16}{\pi}$	$\frac{16}{\pi}\left(\left(21r^2-30\right)r^2+10\right)r^3\sin(3\phi)$	[1]
7	-1	$\frac{16}{\pi}$	$\frac{16}{\pi}\left(\left(\left(35r^2-60\right)r^2+30\right)r^2-4\right)r\sin(\phi)$	
7	1	$\frac{16}{\pi}$	$\frac{16}{\pi}\left(\left(\left(35r^2-60\right)r^2+30\right)r^2-4\right)r\cos(\phi)$	
7	3	$\frac{16}{\pi}$	$\frac{16}{\pi}\left(\left(21r^2-30\right)r^2+10\right)r^3\cos(3\phi)$	[1]
7	5	$\frac{16}{\pi}$	$\frac{16}{\pi}\left(7r^2-6\right)r^5\cos(5\phi)$	
7	7	$\frac{16}{\pi}$	$\frac{16}{\pi}r^7\cos(7\phi)$	
8	-8	$\frac{18}{\pi}$	$\frac{18}{\pi}r^8\sin(8\phi)$	
8	-6	$\frac{18}{\pi}$	$\frac{18}{\pi}\left(8r^2-7\right)r^6\sin(6\phi)$	

continued . . .

. . . continued

Order [n]	Rank [m]	ω	Formula
8	-4	$\frac{18}{\pi}$	$\frac{18}{\pi}\left(\left(28r^2-42\right)r^2+15\right)r^4\sin(4\phi)$
8	-2	$\frac{18}{\pi}$	$\frac{18}{\pi}\left(\left(\left(56r^2-105\right)r^2+60\right)r^2-10\right)r^2\sin(2\phi)$
8	0	$\frac{9}{\pi}$	$\frac{9}{\pi}\left(\left(\left(\left(70r^2-140\right)r^2+90\right)r^2-20\right)r^2+1\right)$
8	2	$\frac{18}{\pi}$	$\frac{18}{\pi}\left(\left(\left(56r^2-105\right)r^2+60\right)r^2-10\right)r^2\cos(2\phi)$
8	4	$\frac{18}{\pi}$	$\frac{18}{\pi}\left(\left(28r^2-42\right)r^2+15\right)r^4\cos(4\phi)$
8	6	$\frac{18}{\pi}$	$\frac{18}{\pi}\left(8r^2-7\right)r^6\cos(6\phi)$
8	8	$\frac{18}{\pi}$	$\frac{18}{\pi}r^8\cos(8\phi)$
9	-9	$\frac{20}{\pi}$	$\frac{20}{\pi}r^9\sin(9\phi)$
9	-7	$\frac{20}{\pi}$	$\frac{20}{\pi}\left(9r^2-8\right)r^7\sin(7\phi)$
9	-5	$\frac{20}{\pi}$	$\frac{20}{\pi}\left(\left(36r^2-56\right)r^2+21\right)r^5\sin(5\phi)$
9	-3	$\frac{20}{\pi}$	$\frac{20}{\pi}\left(\left(\left(84r^2-168\right)r^2+105\right)r^2-20\right)r^3\sin(3\phi)$
9	-1	$\frac{20}{\pi}$	$\frac{20}{\pi}\left(\left(\left(\left(126r^2-280\right)r^2+210\right)r^2-60\right)r^2+5\right)r\sin(\phi)$
9	1	$\frac{20}{\pi}$	$\frac{20}{\pi}\left(\left(\left(\left(126r^2-280\right)r^2+210\right)r^2-60\right)r^2+5\right)r\cos(\phi)$

continued . . .

. . . continued

Order [n]	Rank [m]	ω	Formula
9	3	$\frac{20}{\pi}$	$\frac{20}{\pi}\left(\left(\left(84r^2-168\right)r^2+105\right)r^2-20\right)r^3\cos(3\phi)$
9	5	$\frac{20}{\pi}$	$\frac{20}{\pi}\left(\left(36r^2-56\right)r^2+21\right)r^5\cos(5\phi)$
9	7	$\frac{20}{\pi}$	$\frac{20}{\pi}\left(9r^2-8\right)r^7\cos(7\phi)$
9	9	$\frac{20}{\pi}$	$\frac{20}{\pi}r^9\cos(9\phi)$
10	-10	$\frac{22}{\pi}$	$\frac{22}{\pi}r^10\sin(10\phi)$
10	-8	$\frac{22}{\pi}$	$\frac{22}{\pi}\left(10r^2-9\right)r^8\sin(8\phi)$
10	-6	$\frac{22}{\pi}$	$\frac{22}{\pi}\left(\left(45r^2-72\right)r^2+28\right)r^6\sin(6\phi)$
10	-4	$\frac{22}{\pi}$	$\frac{22}{\pi}\left(\left(\left(120r^2-252\right)r^2+168\right)r^2-35\right)r^4\sin(4\phi)$
10	-2	$\frac{22}{\pi}$	$\frac{22}{\pi}\left(\left(\left(\left(210r^2-504\right)r^2+420\right)r^2-140\right)r^2+15\right)r^2\sin(2\phi)$
10	0	$\frac{11}{\pi}$	$\frac{11}{\pi}\left(\left(\left(\left(252r^2-630\right)r^2+560\right)r^2-210\right)r^2+30\right)r^2-1\right)$
10	2	$\frac{22}{\pi}$	$\frac{22}{\pi}\left(\left(\left(\left(210r^2-504\right)r^2+420\right)r^2-140\right)r^2+15\right)r^2\cos(2\phi)$
10	4	$\frac{22}{\pi}$	$\frac{22}{\pi}\left(\left(\left(120r^2-252\right)r^2+168\right)r^2-35\right)r^4\cos(4\phi)$
10	6	$\frac{22}{\pi}$	$\frac{22}{\pi}\left(\left(45r^2-72\right)r^2+28\right)r^6\cos(6\phi)$

continued . . .

. . . continued

Order [n]	**Rank [m]**	ω	**Formula**
10	8	$\frac{22}{\pi}$	$\frac{22}{\pi}\left(10r^2-9\right)r^8\cos(8\phi)$
10	10	$\frac{22}{\pi}$	$\frac{22}{\pi}r^10\cos(10\phi)$

All numbers are shown here in rational form, but can be converted to decimal representations in-code to save FLOPS

[1] *Zernike polynomials: a guide* [99] contained an error in the formulae for Z_7^{-3} and Z_7^3, in which the provided values of 2 should be replaced by 21.

Appendix B

FE Example

This appendix contains a Python script that was developed that demonstrated the behavior and characteristics of FE-based methodologies. Some aspects that were explored were comparison to mesh-type data and the differences between separable and multivariate FES.

File B.1 – A script that demonstrates the characteristics of FE-based methodologies. FEDemonstration.py

```
#!/bin/env python3
"""
Creates FET example
"""

from mpl_toolkits.mplot3d import Axes3D
import math
import numpy
import matplotlib.pyplot as pyplot
from matplotlib.ticker import MaxNLocator
import scipy.integrate as integrate

def CrazyFunction(x, y):
    xOffset = -0.4
    yOffset = -0.5
    damping = .4
    frequency = 1 * math.pi
    space = numpy.sin(frequency * numpy.exp(y - yOffset)) * numpy.cos(frequency * 1.5 * ((x + xOffset)*(x - 2 * xOffset) + (y + yOffset)**2)) * numpy.exp(-damping * ((x + xOffset)*(x - 2 *xOffset) + (y + yOffset)**2))
    return space

dpi=144
plot3D = True
plotMap = True
plotDiff = True
legendreHigh = 20
mapHigh = 5
# I use rhomberg numerical integration, so the sizes must one greater than a power of 2
# Also note that 2**11 + 1 is as big as I can get for the map plot on a machine with 32 GB RAM
fidelity = 11
xSize = 2**fidelity + 1
ySize = 2**fidelity + 1
xLin = numpy.linspace(-1, 1, xSize)
yLin = numpy.linspace(-1, 1, ySize)
x, y = numpy.meshgrid(xLin, yLin, indexing = "ij")
vmin = -1
vmax = 1
fineness=4

params = {"legend.fontsize": "small", "legend.framealpha": 0.7, "legend.loc": "upper right", "legend.fancybox": True}
pyplot.rcParams.update(params)

truth = CrazyFunction(x, y)

if plot3D:
    print("\nGenerating 3D representation of the sample distribution...")
    figureSplit = pyplot.figure(figsize=(10, 4))

    axes = pyplot.subplot2grid((1, 2), (0, 0), projection='3d')
    axes.set_axis_off()
    axes.view_init(80, -45)
    surf = axes.plot_surface(x, y, truth, cmap=pyplot.cm.gnuplot2, linewidth=0, rcount=xSize/fineness, ccount=ySize/fineness, vmin=vmin, vmax=vmax, antialiased=False)

    axes = pyplot.subplot2grid((1, 2), (0, 1))
    axes.xaxis.set_visible(False)
    axes.yaxis.set_visible(False)
    mesh = pyplot.pcolormesh(xLin, yLin, truth.T, cmap=pyplot.cm.gnuplot2, vmin=vmin, vmax=vmax)
    pyplot.colorbar(mesh, format = "%.3g")
    pyplot.title("Source")
```

```
59
60      pyplot.tight_layout()
61      # Save
62      figureSplit.savefig("3DRepresentation.png", dpi=dpi)
63      pyplot.close()
64      print("3D representation image saved!")
65
66  if plotMap or plotDiff:
67      print("\nGenerating expansion coefficients... ")
68      a = numpy.zeros((legendreHigh + 1, legendreHigh + 1))
69      b = numpy.zeros((legendreHigh + 1, legendreHigh + 1))
70      legendreXLin = numpy.polynomial.legendre.legvander(xLin, legendreHigh)
71      legendreYLin = numpy.polynomial.legendre.legvander(yLin, legendreHigh)
72      legendreX = numpy.zeros((legendreHigh + 1, xSize, ySize))
73      legendreY = numpy.copy(legendreX)
74      for i in range(legendreHigh + 1):
75          legendreX[i], legendreY[i] = numpy.meshgrid(legendreXLin[:, i] * (i + 0.5), ↲
    ↳legendreYLin[:, i] * (i + 0.5), indexing = "ij")
76      projectLin = numpy.zeros(ySize)
77      for i in range(legendreHigh + 1):
78          for j in range(legendreHigh + 1):
79              projection = truth * legendreX[i] * legendreY[j]
80              for p in range(ySize):
81                  projectLin[p] = integrate.romb(projection[:, p], xLin[1] - xLin[0])
82              a[i, j] = integrate.romb(projectLin, yLin[1] - yLin[0])
83      print("done!")
84      for i in range(legendreHigh + 1):
85          for j in range(legendreHigh + 1):
86              if i == 0 or j == 0:
87                  b[i, j] = a[i, j]
88
89  if plotMap:
90      print("\nGenerating map elements...")
91      figureMap = pyplot.figure(figsize=(6, 4))
92
93      allLegendre = numpy.zeros((mapHigh, mapHigh, xSize, ySize))
94      individualLegendre = numpy.copy(allLegendre)
95      for i in range(mapHigh):
96          print("\tComputing order {} of {}".format(i + 1, mapHigh))
97          for j in range(mapHigh):
98              coef = a[:(i + 1), :(j + 1)]
99              axes = pyplot.subplot2grid((mapHigh, mapHigh), (i, j))
100             axes.xaxis.set_visible(False)
101             axes.yaxis.set_visible(False)
102             allLegendre[i, j] = numpy.polynomial.legendre.leggrid2d(xLin, yLin, coef)
103             individualLegendre[i, j] = numpy.copy(allLegendre[i, j])
104             if i == 0:
105                 axes.set_title("i={}".format(j))
106             else:
107                 individualLegendre[i, j] -= allLegendre[i - 1, j]
108             if j == 0:
109                 axes.yaxis.set_visible(True)
110                 axes.set_yticks([])
111                 axes.set_ylabel("j={}        ".format(i), rotation=0, size='large')
112             else:
113                 individualLegendre[i, j] -= allLegendre[i, j - 1]
114             if i > 1 and j > 1:
115                 individualLegendre[i, j] += allLegendre[i - 1, j - 1]
116             plot = pyplot.pcolormesh(xLin, yLin, individualLegendre[i, j].T, cmap=pyplot.↲
    ↳cm.gnuplot2)
117     print("Plotting the data...")
118     pyplot.tight_layout()
119     # Save
120     figureMap.savefig("ProjectionMap.png", dpi=dpi)
121     pyplot.close()
122     print("Saved map plot")
```

```
123
124 if plotDiff:
125     print("\nGenerating diff plots...")
126     figureSplit = pyplot.figure(figsize=(10, 4))
127
128     fullSet = numpy.zeros((legendreHigh + 1, xSize, ySize))
129     separableSet = numpy.zeros((legendreHigh + 1, xSize, ySize))
130     orders = range(legendreHigh + 1)
131     variance = numpy.zeros((legendreHigh + 1))
132     sepVar = numpy.zeros((legendreHigh + 1))
133     meshVariance = numpy.zeros((legendreHigh + 1))
134     for i, coefficients in enumerate(range(1, legendreHigh + 2)):
135         print("\tComputing order {:>2} of {:>2}".format(i, legendreHigh))
136         # Compute the FE
137         if i < mapHigh and plotMap:
138             fullSet[i] = allLegendre[i, i]
139         else:
140             fullSet[i] = numpy.polynomial.legendre.leggrid2d(xLin, yLin, a[:coefficients,
    :coefficients])
141         separableSet[i] = numpy.polynomial.legendre.leggrid2d(xLin, yLin, b[:coefficients
    , :coefficients])
142
143         # Generate the equivalent mesh
144         bins = i + 1
145         histogram = numpy.zeros((bins, bins))
146         histogramError = numpy.zeros((xSize, ySize))
147         xSpace = xSize / bins
148         ySpace = ySize / bins
149         for m in range(bins):
150             for n in range(bins):
151                 mMin = numpy.around(m * xSpace).astype(int)
152                 mMax = numpy.around((m + 1) * xSpace - 1).astype(int)
153                 nMin = numpy.around(n * ySpace).astype(int)
154                 nMax = numpy.around((n + 1) * ySpace - 1).astype(int)
155                 histogram[m, n] = numpy.mean(truth[mMin:mMax, nMin:nMax])
156                 for p in range(mMin, mMax + 1):
157                     for q in range(nMin, nMax + 1):
158                         histogramError[p, q] = histogram[m, n] - truth[p, q]
159         xEdges = numpy.linspace(-1, 1, bins + 1)
160         yEdges = numpy.linspace(-1, 1, bins + 1)
161         dataDiff = fullSet[i] - truth
162         variance[i] = dataDiff.var()
163         sepDiff = separableSet[i] - truth
164         sepVar[i] = sepDiff.var()
165         meshVariance[i] = histogramError.var()
166         if i == 0:
167             diffMax = numpy.max((numpy.absolute(dataDiff).max(), numpy.absolute(
    histogramError).max()))
168             diffMin = -diffMax
169
170         # Clear the graph
171         pyplot.clf()
172
173         if i == 0:
174             plural = ""
175         else:
176             plural = "s"
177
178         # Plot the FE vs. FE Error
179         # FE
180         axes = pyplot.subplot2grid((1, 2), (0, 0))
181         axes.xaxis.set_visible(False)
182         axes.yaxis.set_visible(False)
183         fe = pyplot.pcolormesh(xLin, yLin, fullSet[i].T, cmap=pyplot.cm.gnuplot2, vmin=
    vmin, vmax=vmax)
184         pyplot.title("FE: {n} coefficient{p}".format(n=(coefficients**2), p=plural))
```

```
185         pyplot.tight_layout()
186         pyplot.colorbar(fe, format = "%.3g")
187         # Error
188         axes = pyplot.subplot2grid((1, 2), (0, 1))
189         axes.xaxis.set_visible(False)
190         axes.yaxis.set_visible(False)
191         dif = pyplot.pcolormesh(xLin, yLin, dataDiff.T, cmap=pyplot.cm.gnuplot2, vmin=
    ↳diffMin, vmax=diffMax)
192         pyplot.title("Difference with Source")
193         pyplot.colorbar(dif, format = "%.3g")
194         pyplot.tight_layout()
195         # Save
196         figureSplit.savefig("FE{:02}.png".format(i), dpi=dpi)
197 
198         # Plot the FE vs. Mesh
199         # FE
200         axes = pyplot.subplot2grid((1, 2), (0, 0))
201         axes.xaxis.set_visible(False)
202         axes.yaxis.set_visible(False)
203         fet = pyplot.pcolormesh(xLin, yLin, fullSet[i].T, cmap=pyplot.cm.gnuplot2, vmin=
    ↳vmin, vmax=vmax)
204         pyplot.title("FE: {n} coefficient{p}".format(n=(coefficients**2), p=plural))
205         pyplot.tight_layout()
206         pyplot.colorbar(fet, format = "%.3g")
207         # Mesh
208         axes = pyplot.subplot2grid((1, 2), (0, 1))
209         axes.xaxis.set_visible(False)
210         axes.yaxis.set_visible(False)
211         mesh = pyplot.pcolormesh(xEdges, yEdges, histogram.T, cmap=pyplot.cm.gnuplot2, 
    ↳vmin=vmin, vmax=vmax)
212         pyplot.colorbar(mesh, format = "%.3g")
213         pyplot.title("Mesh: {n} coefficient{p}".format(n=(coefficients**2), p=plural))
214         pyplot.tight_layout()
215         # Save
216         figureSplit.savefig("Mesh{:02}.png".format(i), dpi=dpi)
217 
218         # Multivariate FE
219         axes = pyplot.subplot2grid((1, 2), (0, 0))
220         axes.xaxis.set_visible(False)
221         axes.yaxis.set_visible(False)
222         fet = pyplot.pcolormesh(xLin, yLin, fullSet[i].T, cmap=pyplot.cm.gnuplot2, vmin=
    ↳vmin, vmax=vmax)
223         pyplot.title("Multivariate: {n} coefficient{p}".format(n=(coefficients**2), p=
    ↳plural))
224         pyplot.tight_layout()
225         pyplot.colorbar(fet, format = "%.3g")
226         # Separable FE
227         axes = pyplot.subplot2grid((1, 2), (0, 1))
228         axes.xaxis.set_visible(False)
229         axes.yaxis.set_visible(False)
230         fet = pyplot.pcolormesh(xLin, yLin, separableSet[i].T, cmap=pyplot.cm.gnuplot2, 
    ↳vmin=vmin, vmax=vmax)
231         pyplot.title("Separable: {n} coefficient{p}".format(n=(coefficients**2 - (
    ↳coefficients-1)**2), p=plural))
232         pyplot.tight_layout()
233         pyplot.colorbar(fet, format = "%.3g")
234         #Save
235         figureSplit.savefig("SeparableFE{:02}.png".format(i), dpi=dpi)
236 
237 
238     # Multivariate FE
239     i = 5
240     coefficients = 6
241     axes = pyplot.subplot2grid((1, 2), (0, 0))
242     axes.xaxis.set_visible(False)
243     axes.yaxis.set_visible(False)
```

```
244     fet = pyplot.pcolormesh(xLin, yLin, fullSet[i].T, cmap=pyplot.cm.gnuplot2, vmin=vmin,
        ↳ vmax=vmax)
245     pyplot.title("Multivariate: {n} coefficient{p}".format(n=(coefficients**2), p=plural)
        ↳)
246     pyplot.tight_layout()
247     pyplot.colorbar(fet, format = "%.3g")
248     # Separable FE
249     i = 20
250     coefficients = 21
251     axes = pyplot.subplot2grid((1, 2), (0, 1))
252     axes.xaxis.set_visible(False)
253     axes.yaxis.set_visible(False)
254     fet = pyplot.pcolormesh(xLin, yLin, separableSet[i].T, cmap=pyplot.cm.gnuplot2, vmin=
        ↳vmin, vmax=vmax)
255     pyplot.title("Separable: {n} coefficient{p}".format(n=(coefficients**2 - (
        ↳coefficients-1)**2), p=plural))
256     pyplot.tight_layout()
257     pyplot.colorbar(fet, format = "%.3g")
258     #Save
259     figureSplit.savefig("CloseSeparable.png", dpi=dpi)
260
261     # Plot the statistical history
262     meanVariancePlot = pyplot.figure(figsize=(8, 3.3))
263     variancePlot, = pyplot.semilogy(orders, variance)
264     sepVariancePlot, = pyplot.semilogy(orders, sepVar)
265     meshVariancePlot, = pyplot.semilogy(orders, meshVariance)
266     pyplot.title("Statistical variance as a function of order")
267     pyplot.xlabel("Order")
268     pyplot.ylabel("Variance")
269     meanVariancePlot.gca().xaxis.set_major_locator(MaxNLocator(integer=True))
270     pyplot.autoscale(enable=True, axis='x', tight=True)
271     legend = pyplot.legend([variancePlot, sepVariancePlot, meshVariancePlot], ["FE
        ↳Multivariate", "FE Separable", "Mesh"], loc='lower left')
272     legend.get_frame().set_linewidth(0.0)
273     pyplot.tight_layout()
274         # Save
275     meanVariancePlot.savefig("FEStatisticsHistory.pdf".format(i))
```

Appendix C

Serpent Detector Processing Scripts

This appendix provides a set of Python tools that were developed for plotting Serpent detector results.

C.1 Detector Data

File C.1 – A front-end script for plotting the Serpent Cartesian FET-based detector output. ProcessDetectors.py

```
#!/bin/env python3
"""
Provides an interface for reading Serpent detector output.

Intended for comparing mesh tallies with functional expansion tally results.
"""

import argparse
import copy
import math
import matplotlib.pyplot as pyplot
#import matplotlib.transforms as transforms
import numpy
import os


import det
import pd_utilities as util

# Configurations
util.InitTerminal()
pyplot.style.use('seaborn-colorblind')
params = {"legend.fontsize" : "small", "legend.framealpha" : 0.85, "legend.loc" : "best", "legend.fancybox" : True, "font.family" : "Times New Roman"}
pyplot.rcParams.update(params)

# Record the history for easy editing and regeneration of plots
history = []

# Configure the argument parser
class SmartFormatter(argparse.HelpFormatter):
    def _split_lines(self, text, width):
        if text.startswith('R|'):
            return text[2:].splitlines()
        # this is the RawTextHelpFormatter._split_lines
        return argparse.HelpFormatter._split_lines(self, text, width)
parser = argparse.ArgumentParser(
    description = "Process a Serpent detector file and interactively plot the data",
    formatter_class=SmartFormatter,
    epilog = "This script will generate a \"history.pd\" upon succesful completion, which can "
             "then be redirected as input to this script to reproduce the same results. "
             "Alternatively, it could first be edited manually to change the values, e.g. "
             "correcting a spelling error in a label or changing a slice position.")
parser.add_argument("file",             nargs = "+",
                    help = "path of the Serpent detector file(s)")
saving = parser.add_argument_group("Automated saving of images")
saving.add_argument("--eps",            action = "store_true",
                    help = "store each plot as an encapsulated Postscript (.eps)")
saving.add_argument("--pdf",            action = "store_true",
                    help = "store each plot as a vector-based (.pdf)")
saving.add_argument("--pgf",            action = "store_true",
                    help = "store each plot as LaTeX PGF code (.pgf)")
saving.add_argument("--png",            action = "store_true",
                    help = "store each plot as a rasterized (.png)")
saving.add_argument("--raw",            action = "store_true",
                    help = "store each plot as a rasterized raw RGB bitmap (.raw)")
saving.add_argument("--svg",            action = "store_true",
                    help = "store each plot as a vector (.svg)")
```

```
58 saving.add_argument("--save-2D-vector", action = "store_true",
59                     help = "allow saving 2D plots with vectorized formats (large files, "
60                            "i.e. > 3 MB, may be produced when generating a vectorized "
61                            "form of the 2D plots")
62 saving.add_argument("--individual-2D",  action = "store_true",
63                     help = "save individual files for each of the subplots in the diff "
64                            "view in addition to the full image")
65 saving.add_argument("--name",                               metavar = "output",
66                     help = "R|file name to use when saving an image, the default is"
67                            "\nto use the name of the input file with \"_plot\""
68                            "\nappended"
69                            "\nNOTE: when multiple images are generated then each file"
70                            "\nwill be given a sequential name following \"outputN\"")
71 images = parser.add_argument_group("Image creation parameters")
72 images.add_argument("--dpi",        default = 300,          metavar = "n",          type
   ↳= int,
73                     help = "sets the DPI to use when saving rasterized images [default:
   ↳300]")
74 images.add_argument("--height",     default = 5,            metavar = "y",          type
   ↳= float,
75                     help = "specifies the figure height in inches [default: 5]")
76 images.add_argument("--width",      default = 8,            metavar = "x",          type
   ↳= float,
77                     help = "specifies the figure width in inches [default: 8]")
78 tuning = parser.add_argument_group("Plot tuning")
79 tuning.add_argument("--colormap",   default = "gnuplot2",   metavar = "map_name",
80                     help = "R|specifies the colormap to use when generating 2D plots"
81                            "\n[default: \"gnuplot2\"]"
82                            "\nNOTE: more information about colormaps can be found at"
83                            "\n\"http://matplotlib.org/users/colormaps.html\"; be sure"
84                            "\nto pay attention to the grayscale analysis section")
85 tuning.add_argument("--pointfine",  default = math.inf,     metavar = "f",          type
   ↳= float,
86                     help = "sets number of points per inch to use when generating the
   ↳FETs data [default: 10]")
87 tuning.add_argument("--meshfine",   default = math.inf,     metavar = "m",          type
   ↳= float,
88                     help = "sets the number of sub-evaluations to use when comparing an
   ↳FET to a mesh tally [default: 20]")
89 tuning.add_argument("--allfine",    default = math.inf,     metavar = "F",          type
   ↳= float,
90                     help = "sets the values for both \"f\" and \"m\"")
91 tuning.add_argument("--no-subdiff", action = "store_true",
92                     help = "don't clear the screen for each menu")
93 others = parser.add_argument_group("Other miscellaneous parameters")
94 volume = others.add_mutually_exclusive_group()
95 volume.add_argument("--dirty",      action = "store_true",
96                     help = "don't clear the screen for each menu")
97 volume.add_argument("--silent",      action = "store_true",
98                     help = "don't print any output - I know what I'm doing")
99 others.add_argument("--rename",       action = "store_true",
100                    help = "prompt to rename the detectors; it is recommended to enable
   ↳this option if you plan on"
101                           " resuing the same command history while changing the files
   ↳with the possiblity of"
102                           " duplicate detector names")
103 others.add_argument("--keep",       action = "store_true",
104                    help = "keeps the current \"history.pd\" file, i.e. do not overwrite
   ↳if it exists")
105 others.add_argument("--print",       action = "store_true",
106                    help = "print out the input, essentially dump the history to the
   ↳terminal")
107 others.add_argument("--show",       action = "store_true",
108                    help = "shows each image and pauses execution until the user "
109                           "closes the figure window (is is recommended to NOT use "
110                           "this option if you will be generating many plots)")
```

```
111 # Parse the arguments
112 arguments = parser.parse_args()
113 
114 # Were entring the thick of it now! Get ready...
115 util.dirty |= arguments.dirty
116 util.silent = arguments.silent
117 util.ClearTerminal()
118 
119 # Process the arguments
120 detectorFiles = arguments.file
121 multipleFiles = len(detectorFiles) > 1
122 # Image output specifications
123 util.saveEPS = arguments.eps
124 util.savePDF = arguments.pdf
125 util.savePGF = arguments.pgf
126 util.savePNG = arguments.png
127 util.saveRAW = arguments.raw
128 util.saveSVG = arguments.svg
129 saveSomething = util.saveEPS or util.savePDF or util.savePGF or util.savePNG or util.saveRAW or util.saveSVG
130 if saveSomething:
131     saveName = arguments.name or util.LongestCommonSubstring(detectorFiles) + "_plot"
132 save2DVector = arguments.save_2D_vector
133 saveIndividual2D = arguments.individual_2D
134 # Plot generation parameters
135 imageDPI = arguments.dpi
136 plotHeight = arguments.height
137 plotWidth = arguments.width
138 # Check the colormap and ensure it is valid
139 try:
140     colormap = pyplot.get_cmap(arguments.colormap)
141 except:
142     colormap = pyplot.cm.plasma
143     util.MyPrint("Unknown colormap name \"{}\". Defaulting to \"{}\"".format(arguments.colormap[0], colormap.name))
144 # Plot fineness options
145 pointFine = 10
146 meshFine = 20
147 if not math.isinf(arguments.allfine):
148     if not (math.isinf(arguments.pointfine) and math.isinf(arguments.meshfine)):
149         util.MyPrint("{}: error: argument --allfine: not allowed with --pointfine nor --meshfine".format(parser.prog))
150         util.MyExit()
151     else:
152         pointFine = arguments.allfine
153         meshFine = arguments.allfine
154 else:
155     if not math.isinf(arguments.pointfine):
156         pointFine = arguments.pointfine
157     if not math.isinf(arguments.meshfine):
158         meshFine = arguments.meshfine
159 # Check pointfine and ensure that we will have at least 4 points to plot
160 smallestFineness = min([pointFine * plotHeight, pointFine * plotWidth])
161 if smallestFineness < 4:
162     originalFineness = pointFine
163     pointFine *= (4 / smallestFineness)
164     util.MyPrint("\"pointfine\" is set too small, resetting from {} to {}.".format(originalFineness, pointFine))
165 subdiff = not arguments.no_subdiff
166 plotShow = arguments.show
167 renameDetectors = arguments.rename
168 keepOldHistory = arguments.keep
169 if not util.isTerminalInput:
170     util.MyPrint("Piped/Redirected input detected, I will not overwrite the history file")
171     keepOldHistory = True
```

```
172 util.printHistory = arguments.print
173
174 # Check the contents of the file
175 detectorFileNames, detectorNames, detectorTypes, detectorBounds, detectorSizes = det.GetDetectors(detectorFiles)
176 if len(detectorNames) == 0:
177     util.MyPrint("\n\n\"{}\" contains no Serpent detectors!\n".format(*detectorFiles))
178     util.MyExit()
179
180 # Check for duplicates / prompt for rename
181 isDuplicates = len(detectorNames) != len(set(detectorNames))
182 if isDuplicates or renameDetectors:
183     if isDuplicates:
184         util.MyPrint("Multiple detectors with the same names were found.")
185     rename = util.MyInput("Would you like to rename the detectors? [y/n] ").strip().lower()
186     history.append(rename)
187     if len(rename) == 0 or rename[0] == "q":
188         util.MyExit(history)
189     if rename[0] == "y":
190         renameDetectors = True
191     else:
192         renameDetectors = False
193
194 # Simplify the detector file names
195 if multipleFiles:
196     # Strip out the path name and any common endings
197     commonDir = os.path.commonprefix(detectorFileNames).rfind("/") + 1
198     simplifiedFileNames = [file[commonDir:] for file in detectorFileNames]
199     reversedFileNames = [file[::-1] for file in simplifiedFileNames]
200     commonEnd = os.path.commonprefix(reversedFileNames).rfind(".")
201     simplifiedFileNames = [file[:commonEnd:-1] for file in reversedFileNames]
202 else:
203     simplifiedFileNames = detectorFileNames
204
205 # Rename the detectors if requested
206 detectorNewName = [name for name in detectorNames]
207 while renameDetectors:
208     # Print everything
209     util.ClearTerminal()
210     util.MyPrint("Here are the current detectors and their parameters:\n")
211     if multipleFiles:
212         util.MyPrint("{:>2})  {:<20}  {:<16}  {:<18}  {:>6}  to  {:<6} [bins]  {:>6}\n{}".format("#", "File", "Name", "Type", "Min0", "Max0", "...", "=" * 80))
213     else:
214         util.MyPrint("{:>2})  {:<16}  {:<18}  {:>6}  to  {:<6} [bins]  {:>6}\n{}".format("#", "Name", "Type", "Min0", "Max0", "...", "=" * 80))
215     for index, detector in enumerate(detectorNewName):
216         if multipleFiles:
217             util.MyPrint("{:>2})  {:<16}  {:<16}  {:<18}".format(index + 1, simplifiedFileNames[index][-20:], detector, det.GetFullTypeName(detectorTypes[index])), end="")
218         else:
219             util.MyPrint("{:>2})  {:<16}  {:<18}".format(index + 1, detector, det.GetFullTypeName(detectorTypes[index])), end="")
220         for dimension, bounds in enumerate(detectorBounds[index]):
221             util.MyPrint("  {:>6}  to  {:<6} [{:<4}]".format(bounds[0], bounds[1], detectorSizes[index][dimension]), end="")
222         util.MyPrint("")
223     util.MyPrint("=" * 80)
224     # Prompt for rename
225     rename = util.MyInput("Which detector(s) would you like to rename (or enter \"e\" to exit renaming)?  ").strip().lower()
226     history.append(rename)
227     if len(rename) > 0:
228         if rename[0] == "q":
```

```
            util.MyExit(history)
        elif rename[0] == "e":
            util.ClearTerminal()
            renameDetectors = False
            break
        rename = util.CleanNumericInput(rename)
        for detectorIndex in rename:
            if detectorIndex > len(detectorNewName):
                util.MyPrint("IGNORING: {}\t(invalid index)".format(index))
            else:
                if multipleFiles:
                    newName = util.MyInput("Enter the new name for detector {:>2}  {:<20}  {:<16}: ".format(detectorIndex, simplifiedFileNames[detectorIndex - 1][-206:], detectorNewName[detectorIndex - 1]))
                else:
                    newName = util.MyInput("Enter the new name for detector {:>2}  {:<16}: ".format(detectorIndex, detectorNewName[detectorIndex - 1]))
                history.append(newName)
                detectorNewName[detectorIndex - 1] = newName

# List the detectors
util.MyPrint("The following detectors are available:\n")
if multipleFiles:
    util.MyPrint("{:>2})  {:<20}  {:<16}  {:<18}  {:>6}  to  {:<6} [bins]  {:>6}\n{}".format("#", "File", "Name", "Type", "Min0", "Max0", "...", "=" * 80))
else:
    util.MyPrint("{:>2})  {:<16}  {:<18}  {:>6}  to  {:<6} [bins]  {:>6}\n{}".format("#", "Name", "Type", "Min0", "Max0", "...", "=" * 80))
for index, detector in enumerate(detectorNewName):
    if multipleFiles:
        util.MyPrint("{:>2})  {:<20}  {:<16}  {:<18}".format(index + 1, simplifiedFileNames[index][-20:], detector, det.GetFullTypeName(detectorTypes[index])), end="")
    else:
        util.MyPrint("{:>2})  {:<16}  {:<18}".format(index + 1, detector, det.GetFullTypeName(detectorTypes[index])), end="")
    for dimension, bounds in enumerate(detectorBounds[index]):
        util.MyPrint("  {:>6}  to  {:<6} [{:<4}]".format(bounds[0], bounds[1], detectorSizes[index][dimension]), end="")
    util.MyPrint("")
util.MyPrint("=" * 80)
# Ask for the detectors to plot
plotUs = util.MyInput("Which detectors would you like to visualize? (an empty choice will select all) ").strip().lower()
history.append(plotUs)
if len(plotUs) > 0:
    if plotUs[0] == "q":
        util.MyExit(history)
    else:
        plotUs = util.CleanNumericInput(plotUs)
        for index in plotUs:
            if index > len(detectorNewName):
                util.MyPrint("IGNORING: {}\t(invalid index)".format(index))
                plotUs.remove(index)
else:
    plotUs = list(range(1, len(detectorNewName) + 1))

# Compare the detectors for equivalent coordinates
groups = []
groupCount = 0
if len(plotUs) > 1:
    similar = []
    for index, detector in enumerate(plotUs):
        match = True
        if detector in [item for sublist in similar for item in sublist]:
            continue
```

```
285         similar.append([plotUs[index]])
286         if index < len(plotUs) - 1:
287             for subindex in range(index + 1, len(plotUs)):
288                 if not det.IsEquivalent(detectorBounds[plotUs[index] - 1], detectorBounds
    [plotUs[subindex] - 1]):
289                     continue
290                 else:
291                     for group in similar:
292                         if plotUs[index] in group:
293                             group.append(plotUs[subindex])
294                             match = False
295                             break
296 
297     # Ask how the equivalent-coordinate detectors should be plotted
298     groups = copy.deepcopy(similar)
299     maxLength = max(len(group) for group in similar)
300     if maxLength > 1:
301         util.ClearTerminal()
302         util.MyPrint("The following detectors were determined to have the same
    coordinates:")
303         for group in similar:
304             if len(group) > 1:
305                 util.MyPrint("\n\n{}\n{:<20}  {:>6}  to  {:<6}  {:>6}  to  {:<6}  {:>6}
    to  {:<6}\n{}"
306                     .format("=" * 80, "     Detector", "Min0", "Max0", "Min1", "Max1",
    "Min2", "Max2", "-" * 80), end="")
307                 for index, detector in enumerate(group):
308                     util.MyPrint("\n{:>2}) {:<16}".format(group[index], detectorNewName[
    group[index] - 1]), end="")
309                     if index == 0:
310                         for bounds in detectorBounds[group[index] - 1]:
311                             util.MyPrint("  {:>6}  to  {:<6}".format(bounds[0], bounds
    [1]), end="")
312                 util.MyPrint("\n{}".format("=" * 80))
313         plottingGroups = util.MyInput("Would you like to plot the detectors: grouped as (
    s)hown [default], grouped (m)anually, or (i)ndividually?  ").strip().lower()
314         history.append(plottingGroups)
315 
316         # Organize the plotting groups accordingly
317         if len(plottingGroups) > 0:
318             if plottingGroups[0] == "q":
319                 util.MyExit(history);
320             elif plottingGroups[0] == "i":
321                 groups = [[detector] for detector in plotUs]
322             elif plottingGroups[0] == "m":
323                 groups = []
324                 for batchIndex, group in enumerate(similar):
325                     addThisGroup = True
326                     util.MyPrint()
327                     while addThisGroup:
328                         util.MyPrint("Input options:")
329                         util.MyPrint("  -  a list of the detectors to be included in
    group #{}".format(groupCount + 1))
330                         if batchIndex < len(plottingGroups) - 1:
331                             util.MyPrint("  -  c) continue on to the next set of paired
    detectors")
332                         else:
333                             util.MyPrint("  -  e) exit the manual configuation: I\'m all
    done!")
334                         choices = util.MyInput(">>>? ".format(groupCount + 1)).strip().
    lower()
335                         history.append(choices)
336                         if len(choices) == 1:
337                             if choices == "q":
338                                 util.MyExit(history);
339                             elif choices == "c" and batchIndex < len(plottingGroups) - 1:
```

```
340                                 addThisGroup = False
341                             elif choices == "e":
342                                 addThisGroup = False
343                         if addThisGroup:
344                             choices = util.CleanNumericInput(choices)
345                             for index in choices:
346                                 if index > len(detectorNewName):
347                                     util.MyPrint("IGNORING: {}\t(invalid index)".format(index))
348                                     choices.remove(index)
349                             if not len(choices):
350                                 util.MyPrint("No valid detectors found in input, please try again.")
351                             else:
352                                 groups.append(choices)
353                                 groupCount += 1
354             else:
355                 # Nothing to do, already grouped the way the user wants or invalid answer
356                 util.MyPrint("Thanks, I\'m glad you appreciate my work!")
357 else:
358     groups = [[detector] for detector in plotUs]
359 
360 # Reparse the data file and get the bin values for the requested detectors
361 util.MyPrint("\n\nReading detector data. Please be patient... ", end="")
362 correspondingFiles = []
363 names = []
364 sizes = []
365 types = []
366 for detectorReference in plotUs:
367     correspondingFiles.append(detectorFileNames[detectorReference - 1])
368     names.append(detectorNames[detectorReference - 1])
369     sizes.append(detectorSizes[detectorReference - 1])
370 data = det.GetDetectorData(correspondingFiles, names, sizes)
371 util.MyPrint("done!\n\n", end="")
372 
373 #Set up the plotting
374 count = 0
375 domain = []
376 groupCount = 0
377 for index, group in enumerate(groups):
378     # Create a list of the detector names in this group
379     labels = [detectorNewName[detectorReference - 1] for detectorReference in group]
380     domain = detectorBounds[group[0] - 1]
381     groupCount = len(group)
382     plotHandles = [[] for p in range(groupCount)]
383     thisGroup = copy.deepcopy(group) # create a copy that does not mute the values in the original array
384     # At least one of the detectors must be an FET for the diff to be meaningful
385     isDiffable = False
386     areBothMeshes = False
387     if groupCount == 2:
388         if (True in [det.IsFET(detectorTypes[detector - 1]) for detector in group]):
389             isDiffable = True
390         elif detectorBounds[group[0] - 1] == detectorBounds[group[1] - 1]:
391             isDiffable = True
392             areBothMeshes = True
393     diffWithMesh = areBothMeshes
394     diffMeshAxes = []
395     # Display the detectors each group
396     util.ClearTerminal()
397     cuts = []
398     while not len(cuts):
399         util.MyPrint("{}\n{:<20}  {:>6}  to  {:<6}  {:>6}  to  {:<6}  {:>6}  to  {:<6}\n{}"
400             .format("=" * 80, "Group #{:<2}".format(index + 1), "Min0", "Max0", "Min1", "Max1", "Min2", "Max2", "-" * 80), end="")
```

```
401         for labelIndex, label in enumerate(labels):
402             util.MyPrint("\n  {:<18}".format(label), end="")
403             if labelIndex == 0:
404                 for bounds in domain:
405                     util.MyPrint("  {:>6}  to  {:<6}".format(bounds[0], bounds[1]), end="
    ↳")
406         util.MyPrint("\n{}".format("=" * 80))
407 
408         # Display the cutting options
409         util.MyPrint("\nPlotting Options:")
410         if groupCount == 1:
411             plotTypes = det.plotNames1D + det.plotNames2D
412             plotTypeIndices = det.plotIndex1D + det.plotIndex2D
413         elif isDiffable:
414             plotTypes = det.plotNames1D + det.plotNames2DDiff
415             plotTypeIndices = det.plotIndex1D + det.plotIndex2D
416         else:
417             plotTypes = det.plotNames1D
418             plotTypeIndices = det.plotIndex1D
419         for cutNumber, cutName in enumerate(plotTypes, start=1):
420             util.MyPrint("  {:>2}) {}".format(cutNumber, cutName))
421         if len(plotTypes) > len(det.plotIndex1D):
422             util.MyPrint("Note: specify a negative value for the 2D plots to swap axes")
423         cutsRaw = util.MyInput("How would you like to slice the data? ").strip().lower()
424         if len(cutsRaw) > 0 and cutsRaw[0] == "q":
425             util.MyExit(history);
426         trueCuts = util.CleanNumericInput(cutsRaw, True)
427         if len(trueCuts) == 0:
428             util.MyPrint("No valid cuts specified, please try again")
429             continue
430         if groupCount <= 2:
431             trueCuts = [cut for cut in trueCuts if cut in plotTypeIndices + [swapped * -1
    ↳ for swapped in det.plotIndex2D]]
432         else:
433             trueCuts = [cut for cut in trueCuts if cut in plotTypeIndices]
434         cuts = [abs(cut) for cut in trueCuts]
435         if len(cuts) == 0:
436             util.MyPrint("No valid cuts specified, please try again")
437         else:
438             history.append(cutsRaw)
439 
440     # Loop over the specified cuts
441     for cutCount, cut in enumerate(cuts):
442         plot = cut # A rose by any other name... just used in a different context, this makes it easier
    ↳ to read
443         xMin = detectorBounds[group[0] - 1][0][0]
444         xMax = detectorBounds[group[0] - 1][0][1]
445         if len(domain) > 1:
446             yMin = domain[1][0]
447             yMax = domain[1][1]
448             if len(domain) > 2:
449                 zMin = domain[2][0]
450                 zMax = domain[2][1]
451         horizontalAxis = []
452         verticalAxis = []
453         meshBounds = []
454         xSlize = 0
455         ySlice = 0
456         zSlice = 0
457         xNorm = 0
458         yNorm = 0
459         zNorm = 0
460         xIndex = 0
461         yIndex = 0
462         zIndex = 0
463         values = []
```

```
464         previousValues = []
465         makeSubplot = False
466         relativeDiff = True
467 
468         # Print the details of this plot
469         util.ClearTerminal()
470         util.MyPrint("{}\n{:<20}{:<20}{:<20}".format(
471               "-" * 80,
472               "Group #{}".format(index + 1), "Cut #{}".format(cutCount + 1),
473               "Type: \"{}\"{swapped}\n".format(plotTypes[plot - 1], swapped = " (swapped axes)" if trueCuts[cutCount] < 0 else ""))
474         for label in labels:
475             util.MyPrint(label)
476         util.MyPrint("-" * 80)
477 
478         # Get the slicing specifications
479         if cut in det.cutIndexX:
480             xSlice = float(util.MyInput("Enter the x-axis slice location ({:>6}  to {:<6}): ".format(xMin, xMax)))
481             history.append(xSlice)
482             if xSlice < xMin:
483                 util.MyPrint("{:>6} is to low, setting to {:<6}".format(xSlice, xMin))
484                 xSlice = xMin
485             elif xSlice > xMax:
486                 util.MyPrint("{:>6} is to high, setting to {:<6}".format(xSlice, xMax))
487                 xSlice = xMax
488             xNorm = (xSlice - xMin) / (xMax - xMin) * 2 - 1
489             xIndex = int(math.floor((xSlice - xMin) / (xMax - xMin)))
490         if cut in det.cutIndexY:
491             if len(domain) > 1:
492                 ySlice = float(util.MyInput("Enter the y-axis slice location ({:>6}  to {:<6}): ".format(yMin, yMax)))
493                 history.append(ySlice)
494                 if ySlice < yMin:
495                     util.MyPrint("{:>6} is to low, setting to {:<6}".format(ySlice, yMin))
496                     ySlice = yMin
497                 elif ySlice > yMax:
498                     util.MyPrint("{:>6} is to high, setting to {:<6}".format(ySlice, yMax))
499                     ySlice = yMax
500                 yNorm = (ySlice - yMin) / (yMax - yMin) * 2 - 1
501                 yIndex = int(math.floor((ySlice - yMin) / (yMax - yMin)))
502             else:
503                 util.MyPrint("Group only contains 1D data! Ignoring cut specification {}.".format(cut))
504                 continue
505         if cut in det.cutIndexZ:
506             if len(domain) > 2:
507                 zSlice = float(util.MyInput("Enter the z-axis slice location ({:>6}  to {:<6}): ".format(zMin, zMax)))
508                 history.append(zSlice)
509                 if zSlice < zMin:
510                     util.MyPrint("{:>6} is to low, setting to {:<6}".format(zSlice, zMin))
511                     zSlice = zMin
512                 elif zSlice > zMax:
513                     util.MyPrint("{:>6} is to high, setting to {:<6}".format(zSlice, zMax))
514                     zSlice = zMax
515                 zNorm = (zSlice - zMin) / (zMax - zMin) * 2 - 1
516                 zIndex = int(math.floor((zSlice - zMin) / (zMax - zMin)))
517             else:
518                 util.MyPrint("Group only contains 2D data! Ignoring cut specification {}.".format(cut))
519                 continue
```

```
520
521         # Get the basic plot options
522         title = util.MyInput("What is the title? ")
523         history.append(title)
524         hLabel = util.MyInput("What is the horizontal axis label? ")
525         history.append(hLabel)
526         vLabel = util.MyInput("What is the vertical axis label? ")
527         history.append(vLabel)
528         if isDiffable and plot in det.plotIndex2D and subdiff:
529             pyplot.figure(figsize=(plotWidth * 2, plotHeight * 2))
530             makeSubplot = True
531         else:
532             pyplot.figure(figsize=(plotWidth, plotHeight))
533             pyplot.title(title)
534             pyplot.xlabel(hLabel)
535             pyplot.ylabel(vLabel)
536         # 2D plots have an additional colorbar to label
537         if plot in det.plotIndex2D:
538             if isDiffable and subdiff:
539                 relativeDiffQuery = util.MyInput("Plot the differences as (a)bsolute values or (r)elative to the mesh tally [default]? ").strip().lower()
540                 if len(relativeDiffQuery) > 0:
541                     if relativeDiffQuery[0] == "q":
542                         util.MyExit(history);
543                     elif relativeDiffQuery[0] == "a":
544                         relativeDiff = False
545                 history.append(relativeDiffQuery)
546                 # Ensure the smaller sized detector is plotted first
547                 if not areBothMeshes and not det.IsFET(detectorTypes[thisGroup[1] - 1]):
548                     thisGroup[1], thisGroup[0] = thisGroup[0], thisGroup[1]
549                 if not areBothMeshes and not det.IsFET(detectorTypes[thisGroup[0] - 1]):
550                     diffWithMesh = True
551             else:
552                 colorbarLabel = util.MyInput("What is the color bar label? ")
553                 history.append(colorbarLabel)
554
555         # Loop over the detectors in this group and generate the data for plotting
556         util.MyPrint("\n\nGenerating the plot... ", end = "")
557         for plotIndex, detectorReference in enumerate(thisGroup):
558             if diffWithMesh and plotIndex == 1 and plot in det.plotIndex2D:
559                 horizontalPoints = (len(meshBounds[0]) - 1) * meshFine + 1
560                 verticalPoints = (len(meshBounds[1]) - 1) * meshFine + 1
561             else:
562                 horizontalPoints = pointFine * plotWidth + 1
563                 verticalPoints = pointFine * plotHeight + 1
564             if det.IsFET(detectorTypes[detectorReference - 1]): # FET Detector
565                 # Create the bin edges
566                 if plot in det.plotIndexX:
567                     horizontalAxis = numpy.linspace(xMin, xMax, horizontalPoints)
568                     if plot in det.plotIndexY:
569                         verticalAxis = numpy.linspace(yMin, yMax, verticalPoints)
570                     elif plot in det.plotIndexZ:
571                         verticalAxis = numpy.linspace(zMin, zMax, verticalPoints)
572                 elif plot in det.plotIndexY:
573                     horizontalAxis = numpy.linspace(yMin, yMax, horizontalPoints)
574                     if plot in det.plotIndexZ:
575                         verticalAxis = numpy.linspace(zMin, zMax, verticalPoints)
576                 else: # plot in det.plotIndexZ
577                     horizontalAxis = numpy.linspace(zMin, zMax, horizontalPoints)
578                 # Normalize the edges to the range [-1,1], then convert to meshes within the defined bin edges
579                 horizontalNorm = (horizontalAxis - horizontalAxis.min()) / (horizontalAxis.max() - horizontalAxis.min()) * 2 - 1
580                 if plot in det.plotIndex2D:
581                     verticalNorm = (verticalAxis - verticalAxis.min()) / (verticalAxis.max() - verticalAxis.min()) * 2 - 1
```

```
582                     if isDiffable:
583                         horizontalNorm = det.ConvertEdgesToMidpoints(horizontalNorm)
584                         verticalNorm = det.ConvertEdgesToMidpoints(verticalNorm)
585                 # Evaluate the FET at each point
586                 if det.IsFETCart(detectorTypes[detectorReference - 1]):
587                     if plot in det.plotIndex1D:
588                         if plot in det.plotIndexX:
589                             values = numpy.polynomial.legendre.leggrid3d(horizontalNorm, yNorm, zNorm, data[plotUs.index(detectorReference)])
590                         elif plot in det.plotIndexY:
591                             values = numpy.polynomial.legendre.leggrid3d(xNorm, horizontalNorm, zNorm, data[plotUs.index(detectorReference)])
592                         else: # plot in det.plotIndexZ:
593                             values = numpy.polynomial.legendre.leggrid3d(xNorm, yNorm, horizontalNorm, data[plotUs.index(detectorReference)])
594                     else: # plot in det.plotIndex2D
595                         if plot in [cutIndex for cutIndex in det.plotIndexX if cutIndex in det.plotIndexY]:
596                             values = numpy.polynomial.legendre.leggrid3d(horizontalNorm, verticalNorm, zNorm, data[plotUs.index(detectorReference)])
597                         elif plot in [cutIndex for cutIndex in det.plotIndexX if cutIndex in det.plotIndexZ]:
598                             values = numpy.polynomial.legendre.leggrid3d(horizontalNorm, yNorm, verticalNorm, data[plotUs.index(detectorReference)])
599                         else: # plot in [cutIndex for cutIndex in det.plotIndexY if cutIndex in det.plotIndexZ]
600                             values = numpy.polynomial.legendre.leggrid3d(xNorm, horizontalNorm, verticalNorm, data[plotUs.index(detectorReference)])
601             else: # Mesh detector
602                 # Determine the bin indices. We have to do this here, not when getting the slices, because each detector may have a different number of bins
603                 if plot in det.cutIndexX:
604                     xIndex = int(math.floor((xSlice - xMin) / (xMax - xMin) * (detectorSizes[detectorReference - 1][0] - 1)))
605                 if plot in det.cutIndexY:
606                     yIndex = int(math.floor((ySlice - yMin) / (yMax - yMin) * (detectorSizes[detectorReference - 1][1] - 1)))
607                 if plot in det.cutIndexZ:
608                     zIndex = int(math.floor((zSlice - zMin) / (zMax - zMin) * (detectorSizes[detectorReference - 1][2] - 1)))
609                 # Create the bin edges
610                 if plot in det.plotIndexX:
611                     horizontalAxis = numpy.linspace(xMin, xMax, detectorSizes[detectorReference - 1][0] + 1)
612                     if plot in det.plotIndexY:
613                         verticalAxis = numpy.linspace(yMin, yMax, detectorSizes[detectorReference - 1][1] + 1)
614                     elif plot in det.plotIndexZ:
615                         verticalAxis = numpy.linspace(zMin, zMax, detectorSizes[detectorReference - 1][2] + 1)
616                 elif plot in det.plotIndexY:
617                     horizontalAxis = numpy.linspace(yMin, yMax, detectorSizes[detectorReference - 1][1] + 1)
618                     if plot in det.plotIndexZ:
619                         verticalAxis = numpy.linspace(zMin, zMax, detectorSizes[detectorReference - 1][2] + 1)
620                 else: # plot in det.plotIndexZ:
621                     horizontalAxis = numpy.linspace(zMin, zMax, detectorSizes[detectorReference - 1][2] + 1)
622                 # Slice the data
623                 if plot in det.plotIndex1D:
624                     if plot in det.plotIndexX:
625                         if len(domain) == 1:
626                             values = data[plotUs.index(detectorReference)][:]
627                         elif len(domain) == 2:
628                             values = data[plotUs.index(detectorReference)][:, yIndex]
```

```
629                         else:
630                             values = data[plotUs.index(detectorReference)][:, yIndex, zIndex]
631                     elif plot in det.plotIndexY:
632                         if len(domain) == 2:
633                             values = data[plotUs.index(detectorReference)][xIndex, :]
634                         else:
635                             values = data[plotUs.index(detectorReference)][xIndex, :, zIndex]
636                     else: # plot in det.plotIndexZ:
637                         values = data[plotUs.index(detectorReference)][xIndex, yIndex, :]
638                 else: # plot in det.plotIndex2D
639                     print('plot: {}'.format(plot))
640                     if len(domain) == 2:
641                         values = data[plotUs.index(detectorReference)][:, :]
642                     elif plot in det.cutIndexX:
643                         print('xIndex: {}'.format(xIndex))
644                         values = data[plotUs.index(detectorReference)][xIndex, :, :]
645                     elif plot in det.cutIndexY:
646                         print('yIndex: {}'.format(yIndex))
647                         values = data[plotUs.index(detectorReference)][:, yIndex, :]
648                     else: # plot in det.cutIndexZ
649                         print('zIndex: {}'.format(zIndex))
650                         values = data[plotUs.index(detectorReference)][:, :, zIndex]
651 
652             # Plot the data!
653             if plot in det.plotIndex1D:
654                 if det.IsFET(detectorTypes[detectorReference - 1]): # FET == plot a straight line
655                     values = det.Normalize1DCont(values, horizontalAxis)
656                     plotHandles[plotIndex] = pyplot.plot(horizontalAxis, values)
657                 else: # Mesh == plot a step function
658                     values = det.Normalize1DMesh(values, horizontalAxis)
659                     # Repeat the first value once so the step function plots the data correctly
660                     values = numpy.pad(values, (1, 0), 'edge')
661                     plotHandles[plotIndex] = pyplot.step(horizontalAxis, values)
662             else: # 2D
663                 extent = [horizontalAxis.min(), horizontalAxis.max(), verticalAxis.min(), verticalAxis.max()]
664                 values = det.Normalize2D(values, horizontalAxis, verticalAxis)
665                 # Perform the diff measurement if requested
666                 if isDiffable:
667                     if plotIndex == 0:
668                         previousValues = values
669                         meshBounds = [horizontalAxis, verticalAxis]
670                         continue
671                     else:
672                         if diffWithMesh:
673                             if areBothMeshes:
674                                 horizontalDiffAxis, verticalDiffAxis, diffValues, weights = det.DiffTwoMeshes2D(previousValues, meshBounds[0], meshBounds[1], values, horizontalAxis, verticalAxis, relativeDiff)
675                             else:
676                                 diffValues, averageDiffValues = det.DiffWithMesh2D(previousValues, values, meshFine, relativeDiff)
677                                 weights = numpy.ones(diffValues.shape)
678                                 horizontalDiffAxis = horizontalAxis
679                                 verticalDiffAxis = verticalAxis
680                             if relativeDiff:
681                                 diffValues *= 100
682                                 if not areBothMeshes:
683                                     averageDiffValues *= 100
684                         else:
685                             diffValues = values - previousValues
686                             if relativeDiff:
687                                 diffValues /= (previousValues / 100)
```

```
688                         diffAverage, diffStandardDeviation = util.Statistics(diffValues, weights)
689                         if diffWithMesh and not areBothMeshes:
690                             averageDiffAverage, averageDiffstandardDeviation = util.Statistics(averageDiffValues, weights = numpy.ones(averageDiffValues.shape))
691                 # Swap the axes if requested by a negative count number
692                 if trueCuts[cutCount] < 0:
693                     values = values.T
694                     horizontalAxis, verticalAxis = verticalAxis, horizontalAxis
695                 if not subdiff or not isDiffable:
696                     pyplot.pcolormesh(horizontalAxis, verticalAxis, values.T, cmap = colormap)
697                     pyplot.colorbar(label = colorbarLabel)
698                 else:
699                     pMin = numpy.amin([previousValues.min(), values.min()])
700                     pMax = numpy.amax([previousValues.max(), values.max()])
701                     dMin = numpy.amin([diffValues.min(), -diffValues.max()])
702                     dMax = numpy.amax([diffValues.max(), -diffValues.min()])
703                     if relativeDiff:
704                         type = "Relative"
705                         diffColorbarFormatString = "%.2f%%"
706                         diffTitleFormatString = "{}  --  {}\nMin: {:.2f}%     Max: {:.2f}%     Mean: {:.2f}%     σ²: {:.2f}%"
707                     else:
708                         type = "Absolute"
709                         diffColorbarFormatString = "%.2g"
710                         diffTitleFormatString = "{}  --  {}\nMin: {:.4g}     Max: {:.4g}     Mean: {:.4g}     σ²: {:.4g}"
711                     # Mesh detector
712                     pyplot.subplot2grid((2, 2), (0, 0))
713                     meshAxes = pyplot.pcolormesh(meshBounds[0], meshBounds[1], previousValues.T,
714                                                  cmap = colormap, vmin = pMin, vmax = pMax).axes
715                     meshColorbar = pyplot.colorbar(format = "%.3g")
716                     pyplot.title("Neutron Flux Distribution @ {}  --  Detector: {}\nMin: {:.3g}     Max: {:.3g}".format(title, detectorNewName[thisGroup[0] - 1], previousValues.min(), previousValues.max()))
717                     meshAxes.set_xlabel(hLabel)
718                     meshAxes.set_ylabel(vLabel)
719                     # FET detector
720                     pyplot.subplot2grid((2, 2), (0, 1))
721                     fetAxes = pyplot.pcolormesh(horizontalAxis, verticalAxis, values.T,
722                                                 cmap = colormap, vmin = pMin, vmax = pMax).axes
723                     fetColorbar = pyplot.colorbar(format = "%.3g")
724                     pyplot.title("Neutron Flux Distribution @ {}  --  Detector: {}\nMin: {:.3g}     Max: {:.3g}".format(title, detectorNewName[thisGroup[1] - 1], values.min(), values.max()))
725                     fetAxes.set_xlabel(hLabel)
726                     fetAxes.set_ylabel(vLabel)
727                     # Diff plot
728                     pyplot.subplot2grid((2, 2), (1, 0))
729                     diffAxes = pyplot.pcolormesh(horizontalDiffAxis, verticalDiffAxis, diffValues.T,
730                                                  cmap = colormap, vmin = dMin, vmax = dMax).axes
731                     diffColorbar = pyplot.colorbar(format = diffColorbarFormatString)
732                     pyplot.title(diffTitleFormatString.format(title, "{} Difference".format(type), diffValues.min(), diffValues.max(), diffAverage, diffStandardDeviation))
733                     diffAxes.set_xlabel(hLabel)
734                     diffAxes.set_ylabel(vLabel)
735                     # Average difference per mesh bin
736                     if diffWithMesh and not areBothMeshes:
737                         pyplot.subplot2grid((2, 2), (1, 1))
```

```
738                         diffMeshAxes = pyplot.pcolormesh(meshBounds[0], meshBounds[1], averageDiffValues.T,
739                                                          cmap = colormap, vmin = dMin, vmax = dMax).axes
740                         diffMeshColorbar = pyplot.colorbar(format = diffColorbarFormatString)
741                         pyplot.title(diffTitleFormatString.format(title, "Average Error per Mesh Bin", averageDiffValues.min(), averageDiffValues.max(), averageDiffAverage, averageDiffstandardDeviation))
742                         diffMeshAxes.set_xlabel(hLabel)
743                         diffMeshAxes.set_ylabel(vLabel)
744         pyplot.tight_layout()
745         # Shrink the 1D plot the horizontal values, but autofit the vertical axis
746         if plot in det.plotIndex1D:
747             pyplot.autoscale(enable=True, axis='x', tight=True)
748             legend = pyplot.legend([item for sublist in plotHandles for item in sublist], labels)
749 #           legend.get_frame().set_linewidth(0.0)
750         util.MyPrint("done!")
751 
752         # Save the plot if requested
753         if saveSomething:
754             figure = pyplot.gcf()
755             try:
756                 if len(groups) > 1 or len(cuts) > 1:
757                     currentSaveName = "{}{:03}".format(saveName, count)
758                 else:
759                     currentSaveName = saveName
760                 util.MyPrint("\n\nSaving image(s) for group #{}: cut #{} as \"{}\" ... ".format(index + 1, cutCount + 1, currentSaveName), end = "")
761                 if plot in det.plotIndex1D:
762                     util.SaveFigure(figure, currentSaveName)
763                 else:
764                     util.SaveFigure(figure, currentSaveName, save2DVector)
765                     if saveIndividual2D:
766                         # Mesh plot
767                         extent = util.GetAxesExtent(meshAxes, meshColorbar.ax).transformed(figure.dpi_scale_trans.inverted())
768                         extent = util.WidenAndCenter(extent, plotWidth, plotHeight)
769                         util.SaveFigure(figure, currentSaveName + "_mesh", save2DVector, bbox_inches = extent)
770                         # Fet plot
771                         extent = util.GetAxesExtent(fetAxes, fetColorbar.ax).transformed(figure.dpi_scale_trans.inverted())
772                         extent = util.WidenAndCenter(extent, plotWidth, plotHeight)
773                         util.SaveFigure(figure, currentSaveName + "_fet", save2DVector, bbox_inches = extent)
774                         # Diff plot
775                         extent = util.GetAxesExtent(diffAxes, diffColorbar.ax).transformed(figure.dpi_scale_trans.inverted())
776                         extent = util.WidenAndCenter(extent, plotWidth, plotHeight)
777                         util.SaveFigure(figure, currentSaveName + "_diff", save2DVector, bbox_inches = extent)
778                         if diffMeshAxes:
779                             extent = util.GetAxesExtent(diffMeshAxes, diffMeshColorbar.ax).transformed(figure.dpi_scale_trans.inverted())
780                             extent = util.WidenAndCenter(extent, plotWidth, plotHeight)
781                             util.SaveFigure(figure, currentSaveName + "_diffmesh", save2DVector, bbox_inches = extent)
782                 util.MyPrint("done!\n")
783             except:
784                 util.MyPrint("\nError saving image file. Are write permissions enabled?")
785         if plotShow:
786             pyplot.show()
787         count += 1
788         pyplot.close()
```

```
789
790 # Save history to file if something was actually generated
791 if count > 0:
792     util.MyPrint("\n")
793     if keepOldHistory:
794         openString = "x"
795     else:
796         openString = "w"
797     try:
798         historyFile = open("history.pd", openString)
799         util.MyPrint("Saving command history to \"{}\"".format(historyFile.name))
800         for command in history:
801             historyFile.write("{}\n".format(command))
802     except:
803         if keepOldHistory:
804             if util.isTerminalInput:
805                 util.MyPrint("It appears that \"history.pd\" exists. I will leave it
    alone since you specified the \"--keep\" option.")
806         else:
807             util.MyPrint("Error creating \"history.pd\". Are wrtie permissions enabled?")
808
809 util.MyPrint("{} plots generated. Have a nice day!\n\n".format(count))
810 util.MyExit(history)
```

File C.2 – A companion library for ProcessDetectors.py with methods for parsing and manipulating detector data.

det.py 

```
1  """
2  Provides utilities for reading Serpent detector output.
3
4  Provides utilities for parsing and plotting Serpent detector outputs.
5  Intended for comparing with functional expansion tally results.
6
7  Note that the Serpent detector output may change format in the future.
8  This utility is valid for Serpent versions matching the formatting of 2.1.29
9  """
10
11 import math
12 import numpy
13 import scipy.integrate
14
15 numberOfPlots1D = 3
16 numberOfPlots2D = 3
17 plotNames1D = ["1D X", "1D Y", "1D Z"]
18 plotIndex1D = [1, 2, 3]
19 plotNames2D = ["2D XY", "2D XZ", "2D YZ"]
20 plotNames2DDiff = ["2D XY Diff", "2D XZ Diff", "2D YZ Diff"]
21 plotIndex2D = [4, 5, 6]
22 plotIndexX = [1, 4, 5]
23 cutIndexX = [2, 3, 6]
24 plotIndexY = [2, 4, 6]
25 cutIndexY = [1, 3, 5]
26 plotIndexZ = [3, 5, 6]
27 cutIndexZ = [1, 2, 4]
28
29
30 """
31 Step-wise diffs a mesh plot with an FET plot
32 """
33 def ConvertEdgesToMidpoints(edges):
34     midpoints = numpy.zeros(len(edges) - 1)
35     for index in range(len(midpoints)):
36         midpoints[index] = (edges[index] + edges[index + 1]) / 2.0
37     return midpoints
```

```
38
39
40  """
41  Step-wise diffs a mesh plot with an FET plot, aligned with 'fineness' FET bins per mesh bin
42  """
43  def DiffWithMesh2D(meshValues, fetValues, fineness, relativeDiff = True):
44      width, height = fetValues.shape
45      maxX, maxY = meshValues.shape
46      diffPlot = numpy.zeros((width, height))
47      averageDiffPlot = numpy.zeros(meshValues.shape)
48
49      for index in range(width):
50          meshX = int(index * 1.0 / fineness)
51          for indey in range(height):
52              meshY = int(indey * 1.0 / fineness)
53              diffPlot[index][indey] = fetValues[index][indey] - meshValues[meshX][meshY]
54              if relativeDiff:
55                  diffPlot[index][indey] /=  meshValues[meshX][meshY]
56              averageDiffPlot[meshX][meshY] += diffPlot[index][indey]
57      averageDiffPlot /= (fineness * fineness)
58      return diffPlot, averageDiffPlot
59
60
61  """
62  Step-wise diffs for two mesh plots
63  """
64  def DiffTwoMeshes2D(bins1, edgesx1, edgesy1, bins2, edgesx2, edgesy2, relativeDiff = True
    ↳):
65      combinedX = numpy.unique(numpy.concatenate((edgesx1, edgesx2)))
66      width = len(combinedX) - 1
67      combinedY = numpy.unique(numpy.concatenate((edgesy1, edgesy2)))
68      height = len(combinedY) - 1
69      diffs = numpy.zeros((width, height))
70      weights = numpy.zeros((width, height))
71      bin1Area = (edgesx1[1] - edgesx1[0]) * (edgesy1[1] - edgesy1[0])
72      bin2Area = (edgesx2[1] - edgesx2[0]) * (edgesy2[1] - edgesy2[0])
73      maxArea = numpy.max([bin1Area, bin2Area])
74
75      # Ensure the mins and maxs are equal
76      if (    min(edgesx1) == min(edgesx2)
77          and max(edgesx1) == max(edgesx2)
78          and min(edgesy1) == min(edgesy2)
79          and max(edgesy1) == max(edgesy2)
80         ):
81          binx1 = 0
82          binx2 = 0
83          for index in range(width):
84              if binx1 < width and edgesx1[binx1 + 1] < combinedX[index + 1]:
85                 binx1 += 1
86              if binx2 < width and edgesx2[binx2 + 1] < combinedX[index + 1]:
87                  binx2 += 1
88              biny1 = 0
89              biny2 = 0
90              for indey in range(height):
91                  if biny1 < height and edgesy1[biny1 + 1] < combinedY[indey + 1]:
92                      biny1 += 1
93                  if biny2 < height and edgesy2[biny2 + 1] < combinedY[indey + 1]:
94                      biny2 += 1
95
96                  diffs[index][indey] = (bins2[binx2][biny2] - bins1[binx1][biny1])
97                  diffArea = (combinedX[index + 1] - combinedX[index]) * (combinedY[indey +
    ↳ 1] - combinedY[indey])
98                  weights[index][indey] = diffArea / maxArea
99                  if relativeDiff:
100                     diffs[index][indey] /= bins1[binx1][biny1]
101     return combinedX, combinedY, diffs, weights
```

```
102
103
104 """
105 Opens a Serpent-generated detector file and parses it to find the detectors defined therein
106 """
107 def GetDetectors(files):
108     currentDetectorIndex = 0
109     dimCount = 0
110     dimNextLine = False
111     previousDetector = "ted"
112     sizeCount = 0
113
114     correspondingFile = []
115     coordinates = []
116     detectors = []
117     sizes = []
118     types = []
119
120     for fileIndex, file in enumerate(files):
121         with open(file) as search:
122             for line in search:
123                 if dimNextLine:
124                     if line.endswith("];\n"):
125                         dimNextLine = False
126                         dimCount += 1
127                         continue
128
129                     dims = list(filter(None, line.split(" ")))
130                     dims = [float(i) for i in dims]
131
132                     if len(coordinates) < currentDetectorIndex + 1:
133                         coordinates.append([])
134                         sizes.append([])
135                         sizeCount = 0
136
137                     if types[currentDetectorIndex] in ["CART1", "CART2",  "CART3"]:
138                         sizeCount = sizeCount + 1
139                         if len(coordinates[currentDetectorIndex]) < dimCount + 1:
140                             coordinates[currentDetectorIndex].append([dims[0],  math.inf
    ↳])
141                             sizes[currentDetectorIndex].append([1])
142                         else:
143                             coordinates[currentDetectorIndex][dimCount][1] = dims[1]
144                             sizes[currentDetectorIndex][dimCount] = sizeCount
145                     elif types[currentDetectorIndex] in ["FETCART", "FETCART3"]:
146                         coordinates[currentDetectorIndex].append([dims[0],  dims[1]])
147                         sizes[currentDetectorIndex].append(int(dims[2]))
148                 elif line.startswith("DET"):
149                     name = line.split(" ")[0]
150
151                     # Identify if this is a new named detector
152                     if previousDetector != name[:len(previousDetector)]:
153                         detectors.append(name)
154                         correspondingFile.append(files[fileIndex])
155                         previousDetector = name
156                         dimCount = 0
157                     else:
158                         # Try to identify the type
159                         if currentDetectorIndex < len(detectors) - 1:
160                             currentDetectorIndex += 1
161
162                         type = line[len(detectors[currentDetectorIndex] ) :].split(" ")
    ↳[0]
163                         dimNextLine = True
164                         if len(types) == currentDetectorIndex + 1:
165                             if types[currentDetectorIndex] == "CART1" and type == "Y":
```

```
                                types[currentDetectorIndex] = "CART2"
                                sizeCount = 0
                            elif types[currentDetectorIndex] == "CART2" and type == "Z":
                                types[currentDetectorIndex] = "CART3"
                                sizeCount = 0
                            else:
                                print("Unrecognized detector type {}, ignoring".format(type))
                        else:
                            if type in ["FETCART", "FETCART3"]:
                                types.append(type)
                            elif type == "X":
                                types.append("CART1")
    return correspondingFile, detectors, types, coordinates, sizes

"""
Opens a Serpent-generated detector file and extracts the data for the named detectors
"""
def GetDetectorData(files, detectors, sizes):
    detectorIndex = 0
    parsing = False

    data = []
    for index, size in enumerate(sizes):
        if len(size) == 1:
            data.append(numpy.zeros((size[0],  1,  1)))
        elif len(size) == 2:
            data.append(numpy.zeros((size[0],  size[1],  1)))
        else:
            data.append(numpy.zeros((size[0],  size[1],  size[2])))

    for index, file in enumerate(files):
        with open(file) as search:
            for line in search:
                if line.endswith("];\n"):
                    parsing = False
                    continue

                if parsing:
                    values = list(filter(None, line.split(" ")))
                    zbin = int(values[7]) - 1
                    ybin = int(values[8]) - 1
                    xbin = int(values[9]) - 1
                    value = float(values[10])
                    data[detectorIndex][xbin][ybin][zbin] = value
                elif line.startswith("DET") :
                    name = line.split(" ")[0]
                    if name == detectors[index]:
                        detectorIndex = index
                        parsing = True
                    else:
                        parsing = False
    return data

def GetFullTypeName(shortType):
    if shortType in ["FETCART", "FETCART3"]:
        return "3D Cartesian FET"
    else:
        return "TYPE UNKNOWN"

"""
Compares the two coordinate specifications for two detectors and return true if they are equivalent
"""
def IsEquivalent(coordinates1,  coordinates2):
    if len(coordinates1) != len(coordinates2):
        return False
```

```
231
232     for index, check1 in enumerate(coordinates1):
233         if coordinates1[index][0] != coordinates2[index][0] or coordinates1[index][1] !=
  ↳coordinates2[index][1]:
234             return False
235
236     return True
237
238 """
239 Checks if the detector type is a 3D Cartesian
240 """
241 def IsFETCart(detectorType):
242     if detectorType in ["FETCART", "FETCART3"]:
243         return True
244     else:
245         return False
246
247 """
248 Checks if the detector type is a 3D Zernike/Legendre
249 """
250 def IsFETCyl(detectorType):
251     if detectorType == "FETCYL":
252         return True
253     else:
254         return False
255
256 """
257 Checks if the detector type is one of the known FET types
258 """
259 def IsFET(detectorType):
260     if  IsFETCart(detectorType) or IsFETCyl(detectorType):
261         return True
262     else:
263         return False
264
265 """
266 Normalizes a 1D mesh so that the area under the bins is 1.
267 """
268 def Normalize1DMesh(values,  binEdges):
269     area = 0
270     for index,  value in enumerate(values):
271         area += value * (binEdges[index + 1] - binEdges[index])
272     values = values / area
273     return values
274
275 """
276 Normalizes a 1D contiious function so that the area under the curve is approximately 1
277 """
278 def Normalize1DCont(values,  points):
279     return values / scipy.integrate.simps(values,  points)
280
281 """
282 Normalizes a 2D histogram so that the volume under the bins is 1.
283 """
284 def Normalize2D(values,  binEdgesX,  binEdgesY):
285     area = 0
286     xLen, yLen = values.shape
287     for index in range(xLen):
288         for indey in range(yLen):
289             area += values[index][indey] * (binEdgesX[index + 1] - binEdgesX[index]) * (
  ↳binEdgesY[indey + 1] - binEdgesY[indey])
290     values = values / area
291     return values
```

File C.3 – A companion library for ProcessDetectors.py with ancillary utility methods. pd_utilities.py

```
"""
Provides an interface for reading Serpent detector output.

Intended for comparing mesh tallies with functional expansion tally results.
"""

import math
import matplotlib.transforms as transforms
import numpy
import re
import sys

dirty = False
silent = False
isTerminalInput = True
printHistory = True
saveEPS = False
savePDF = False
savePGF = False
savePNG = False
saveRAW = False
saveSVG = False
imageDPI = 300

def LongestCommonSubstring(listOfStrings):
    substr = ''
    if len(listOfStrings) > 1 and len(listOfStrings[0]) > 0:
        for i in range(len(listOfStrings[0])):
            for j in range(len(listOfStrings[0])-i+1):
                if j > len(substr) and all(listOfStrings[0][i:i+j] in x for x in listOfStrings):
                    substr = listOfStrings[0][i:i+j]
    return substr

def Statistics(values, weights):
    average = numpy.average(values, weights=weights)
    variance = numpy.average((values-average)**2, weights=weights)  # Fast and numerically precise
    return (average, math.sqrt(variance))

def MyPrint(*args, **kwargs):
    if not silent:
        print(*args, **kwargs)

def MyInput(*args):
    if not silent:
        response = input(*args)
        if not isTerminalInput:
            print(response)
        return response
    else:
        return input("")

def MakeUnique(input):
  output = []
  for x in input:
    if x not in output:
      output.append(x)
  return output

def CleanNumericInput(input, allowMultiple = False):
    input = re.split("[^-?\\d]", input)
    output = [x for x in input if x]
```

```
62      output = list(map(int, output))
63      if not allowMultiple:
64          output = MakeUnique(output)
65      return output
66
67  def ClearTerminal():
68      if not dirty:
69          MyPrint(chr(27) + "[1J", end="")
70          MyPrint(chr(27) + "[H", end="")
71
72  def GetTerminalSize():
73      def ioctl_GWINSZ(fd):
74          try:
75              import fcntl, termios, struct
76              cr = struct.unpack('hh', fcntl.ioctl(fd, termios.TIOCGWINSZ,
77          '1234'))
78          except:
79              return
80          return cr
81      cr = ioctl_GWINSZ(0) or ioctl_GWINSZ(1) or ioctl_GWINSZ(2)
82      if not cr:
83          cr = (-1,  -1)
84      return int(cr[1]), int(cr[0])
85
86  def IsTerminalInput():
87      import os, stat
88
89      mode = os.fstat(0).st_mode
90      if stat.S_ISFIFO(mode) or stat.S_ISREG(mode):
91           return False
92      else:
93           return True
94
95  def InitTerminal():
96      global dirty
97      global isTerminalInput
98      try:
99          columns, rows = GetTerminalSize()
100         if columns == -1:
101             MyPrint("Cannot detect console size")
102             dirty = True
103     except:
104         print("Cannot detect console size")
105         dirty = True
106     isTerminalInput = IsTerminalInput()
107
108 def SaveFigure(figure, currentSaveName, allowSaveVector = True, **kwargs):
109     if savePNG:
110         figure.savefig(currentSaveName + ".png", dpi = imageDPI, **kwargs)
111     if saveRAW:
112         figure.savefig(currentSaveName + ".raw", dpi = imageDPI, **kwargs)
113     if allowSaveVector:
114         if saveEPS:
115             figure.savefig(currentSaveName + ".eps", **kwargs)
116         if savePDF:
117             figure.savefig(currentSaveName + ".pdf", **kwargs)
118         if savePGF:
119             figure.savefig(currentSaveName + ".pgf", **kwargs)
120         if saveSVG:
121             figure.savefig(currentSaveName + ".svg", **kwargs)
122
123 def GetAxesExtent(axes, colorbar = [], pad = 0.0):
124     axes.figure.canvas.draw()
125     items = axes.get_xticklabels() + axes.get_yticklabels() + colorbar.get_xticklabels()
      + colorbar.get_yticklabels()
```

```
126     items += [axes, axes.title, axes.get_xaxis().get_label(), axes.get_yaxis().get_label
    ↳()]
127     items += [colorbar]
128     bbox = transforms.Bbox.union([item.get_window_extent() for item in items])
129     return bbox.expanded(1.0 + pad, 1.0 + pad)
130 
131 def WidenAndCenter(box, width, height):
132     widthScale = width / box.width
133     heightScale = height / box.height
134     return box.expanded(widthScale, heightScale)
135 
136 def MyExit(history):
137     if printHistory and len(history):
138         MyPrint("{}\n  HISTORY\n{}".format("=" * 80,  "-" * 80))
139         for command in history:
140             MyPrint(command)
141         MyPrint("=" * 80)
142     sys.exit()
```

C.2 FOM Data

File C.4 – A script for for plotting the data in a CSV file containing the time and uncertainty data for detector tallies. TimesVsErrors.py

```python
#!/bin/env python3
"""
Plots the data in a csv file containing the time and uncertainty data for detector tallies
"""

import argparse
import csv
import matplotlib
import matplotlib.pyplot as pyplot
from matplotlib.ticker import FuncFormatter
import numpy as np

params = {"legend.fontsize" : "small", "legend.framealpha" : 0.65, "legend.loc" : "best", "legend.fancybox" : True, "font.family" : "Times New Roman"}
pyplot.rcParams.update(params)

pyplot.style.use('seaborn-colorblind')
colors = [(31, 119, 180), (152, 223, 138), (255, 127, 14), (197, 176, 213)]
for i in range(len(colors)):
    r, g, b = colors[i]
    colors[i] = (r / 255., g / 255., b / 255.)
lineStyles = ['o--', 'o:', 's--', 's:']

def to_percent(y, position):
    # Ignore the passed in position. This has the effect of scaling the default
    # tick locations.
    s = '{:.0f}'.format(100 * y)

    # The percent symbol needs escaping in latex
    if matplotlib.rcParams['text.usetex'] is True:
        return s + r'$\%$'
    else:
        return s + '%'

# Configure the argument parser
class SmartFormatter(argparse.HelpFormatter):
    def _split_lines(self, text, width):
        if text.startswith('R|'):
            return text[2:].splitlines()
        # this is the RawTextHelpFormatter._split_lines
        return argparse.HelpFormatter._split_lines(self, text, width)
parser = argparse.ArgumentParser(description='Get input csv-formatted data files.', formatter_class=SmartFormatter)
parser.add_argument("file",
                    help = "path of the csv data file")
parser.add_argument("--title",      default = "MakeMeATitleNOW",     metavar = "title",
                    help = "title of the plot")
parser.add_argument("--dpi",        default = 300,          metavar = "n",          type = int,
                    help = "sets the DPI to use when saving rasterized images [default: 300]")
parser.add_argument("--height",     default = 3,            metavar = "y",          type = float,
                    help = "specifies the figure height in inches [default: 3]")
parser.add_argument("--width",      default = 5,            metavar = "x",          type = float,
                    help = "specifies the figure width in inches [default: 5]")
parser.add_argument("--nolegend",   action = "store_true",
                    help = "don't include a legend on the plot")

```

```
# Parse the arguments
arguments = parser.parse_args()
file = arguments.file
title = arguments.title
imageDPI = arguments.dpi
plotHeight = arguments.height
plotWidth = arguments.width

# Process the arguments
# Plot generation parameters
imageDPI = arguments.dpi
plotHeight = arguments.height
plotWidth = arguments.width
legend = not arguments.nolegend

detectorNames = []
detectorTimes = []
detectorErrors = []
detectorFOM = []

reader = csv.reader(open(file, 'rt'))
next(reader) # Skip the header row
for row in reader:
    name = row[0]
    data = row[2].split(":")
    time = int(data[0]) * 3600 + int(data[1]) * 60 + float(data[2])
    error = float(row[3].strip(' %')) / 100.0
    FOM = float(row[4])

    if name in detectorNames:
        index = detectorNames.index(name)
        detectorTimes[index].append(time)
        detectorErrors[index].append(error)
        detectorFOM[index].append(FOM)
    else:
        detectorNames.append(name)
        detectorTimes.append([time])
        detectorErrors.append([error])
        detectorFOM.append([FOM])

detectorTimes = np.asarray(detectorTimes) / 3600 # Convert to hours
detectorErrors = np.asarray(detectorErrors)
detectorFOM = np.asarray(detectorFOM)


pyplot.figure(figsize=(plotWidth, plotHeight))
pyplot.title('{}'.format(title))
pyplot.xlabel('Duration [h]')
pyplot.ylabel('Total Relative Variance')
for index, name in enumerate(detectorNames):
    pyplot.plot(detectorTimes[index], detectorFOM[index], lineStyles[index], color=colors
    ↳[index])
if legend:
    pyplot.legend(detectorNames)
pyplot.gca().set_xscale('log')
pyplot.gca().set_yscale('log')
pyplot.tight_layout()
#pyplot.show()
pyplot.savefig('{}TimeVsU.pdf'.format(title))
pyplot.close()

pyplot.figure(figsize=(plotWidth, plotHeight))
pyplot.title('{}'.format(title))
pyplot.xlabel('Duration [h]')
pyplot.ylabel('Overall Relative Error')
for index, name in enumerate(detectorNames):
```

```
120     pyplot.plot(detectorTimes[index], detectorErrors[index], lineStyles[index], color=
  ↳colors[index])
121 if legend:
122     pyplot.legend(detectorNames)
123 pyplot.gca().set_xscale('log')
124 yFormatter = FuncFormatter(to_percent)
125 pyplot.gca().yaxis.set_major_formatter(yFormatter)
126 pyplot.tight_layout()
127 #pyplot.show()
128 pyplot.savefig('{}TimeVsError.pdf'.format(title))
129 
130 pyplot.figure(figsize=(plotWidth, plotHeight))
131 pyplot.title('{}'.format(title))
132 pyplot.xlabel('Total Relative Variance')
133 pyplot.ylabel('Overall Relative Error')
134 for index, name in enumerate(detectorNames):
135     pyplot.plot(detectorFOM[index], detectorErrors[index], lineStyles[index], color=
  ↳colors[index])
136 if legend:
137     pyplot.legend(detectorNames)
138 pyplot.gca().set_xscale('log')
139 yFormatter = FuncFormatter(to_percent)
140 pyplot.gca().yaxis.set_major_formatter(yFormatter)
141 pyplot.tight_layout()
142 #pyplot.show()
143 pyplot.savefig('{}UVsError.pdf'.format(title))
```

Appendix D

Input Files

This appendix contains sample input for Serpent and MOOSE using FE-based functionalities. Examples for Serpent FET-based detectors (see chapters 4 and 5), MOOSE FE-based coupling (see chapter 6), and the MOOSE-Serpent multiphysics interface (see chapter 7) are provided.

The Serpent input files use the `include` directive, allowing nested file levels. This approach was used to split the input files into the basic problem setup, materials, geometry, and then any simulation-specific parameters. An example of a basic problem setup is shown in file D.1, with file D.2 providing the model geometry and file D.3 the material descriptions. A Cartesian FET detector is shown in file D.4, and a cylindrical FET is shown in file D.5. Note that these are example files only, and may not necessarily be cross-compatible with each other. File D.6 demonstrates the FET output generated by Serpent

Next, files D.7 and D.8 demonstrates a pure FE-based volumetric multiphysics coupling. Files D.9 and D.10 shows the boundary-based coupling.

Finally, files D.11 to D.13 demonstrate the MOOSE input files used to create the tightly-coupled simulation presented in chapter 7. The parameters for the Serpent multiphysics interface input are autogenerated by MOOSE. As the MOOSE-Serpent interface modifies the primary Serpent input file, file D.14 is useful to compare against file D.1 to see what additional parameters are added by MOOSE to set up the Serpent simulation for multiphysics. File D.15 demonstrates the interface file that is generated by the `SerpentExecutioner` in MOOSE, and file D.16 demonstrates the FET output generated by Serpent.

D.1 Serpent Input Examples for FETs

File D.1 – An example standard Serpent input file. basic.sss

```
% This is being used for a benchmark, so set the starting seed and reproducibility
set seed 4392750125
set repro 2

% --- Cross section library file path:
set acelib mcnp6.1_endfb-7.1.xsdata

% -- Boundary conditions
set bc 3 3 1

% unresolved resonance
set ures 1 1

% --- Neutron population and criticality cycles:
set pop 10000 1000 200 1.3

% -- Geometry plots
plot         1   300 900   0.0   -0.62992  0.62992   -183.0    183.0     % Axial view
plot         3   300 300   0.0   -0.62992  0.62992   -0.62992  0.62992   % Cross section

% --- Mesh plots of fission rate
mesh         1   300 900   % Axial view
mesh         3   300 300   % Cross section

% --- Temperature plots
mesh  10     1   300 900   % Axial view
mesh  10     2   300 900   % Axial view
mesh  10     3   300 300   % Cross section

include geometry.inp
include materials_600K.inp
```

File D.2 – An example Serpent geometry specifications. geometry.inp

```
pin     1
UO2     0.409575
helium_pin_scoring

surf 10 cyl     0.0              0.0              0.41783
surf 20 cyl     0.0              0.0              0.47498
surf 30 cuboid -0.62992 0.62992 -0.62992 0.62992 -183.0   183.0

cell 100 0 fill    1 -10
cell 200 0 zirlo      10 -20
cell 300 0 water          20 -30
cell 400 0 outside            30
```

File D.3 – An example Serpent materials definitions. materials_600K.inp

```
% --- Materials
mat UO2     -10.424 moder o2_u 8016 moder u_o2 92238   rgb 100 100 100
8016.81c   4.6688131274253E-02
8017.81c   1.7748234213217E-05
92234.81c  8.4512220988817E-06
92235.81c  1.0518948891610E-03
92238.81c  2.2292593642974E-02

```

```
mat helium_pin_scoring  sum tmp 650.0                  rgb 240 240 40
2004.81c   2.4044E-04

mat zirlo  sum tmp 640.0                               rgb 150 0   150
50112.81c  3.1985E-06
50114.81c  2.1763E-06
50115.81c  1.1211E-06
50116.81c  4.7944E-05
50117.81c  2.5324E-05
50118.81c  7.9863E-05
50119.81c  2.8325E-05
50120.81c  1.0743E-04
50122.81c  1.5267E-05
50124.81c  1.9092E-05
26054.81c  4.0970E-06
26056.81c  6.4313E-05
26057.81c  1.4853E-06
26058.81c  1.9766E-07
41093.81c  4.2132E-04
8016.81c   3.0571E-04
8017.81c   1.1621E-07
40090.81c  2.1586E-02
40091.81c  4.7073E-03
40092.81c  7.1952E-03
40094.81c  7.2917E-03
40096.81c  1.1747E-03

mat water -1.0 moder lwtr 1001                         rgb 150 150 255
5010.80c  8.0042E-6
5011.80c  3.2218E-5
1001.80c  4.9457E-02
1002.80c  7.4196E-06
8016.80c  2.4723E-02
8017.80c  9.3982E-06

therm lwtr lwtr.16t
therm o2_u o2_u.13t
therm u_o2 u_o2.13t
```

File D.4 – An example Serpent Cartesian FET definition. FETSlab.inp

```
/* FET coarse detector settings */
det "FETCoarseCartesian"   dfet    1
      -20.355 20.355    4
      -22.905 22.905    4
       0.0    22.06     4
```

File D.5 – An example Serpent cylindrical FET definition. FETCylinder.inp

```
/* FET coarse detector settings */
det "FETCoarseCylindrical"      dfet    2
        29.5            3
        0.0   52.93     4
  3     0.0     0.0
```

File D.6 – A truncated example of the Serpent FET output. FETDetectorOutput.m

```

DETThermalReflectedSlabFETCoarseMedium = [
    1    1    1    1    1    1    1    1    1    1  4.46700E-04  1.97112E-04
    2    1    1    1    1    1    1    1    1    2  6.38376E-08  4.01718E+00
```

```
    3    1    1    1    1    1    1    1    1    3 -1.80165E-04  1.75042E-03
    4    1    1    1    1    1    1    1    1    4  3.85609E-07  9.22049E-01
    5    1    1    1    1    1    1    1    1    5  1.96880E-05  1.91520E-02
    6    1    1    1    1    1    1    1    2    1 -2.71657E-06  9.44765E-02
    7    1    1    1    1    1    1    1    2    2 -7.22520E-07  5.28670E-01
    8    1    1    1    1    1    1    1    2    3  6.44125E-07  7.47426E-01
  ...
  125    1    1    1    1    1    1    5    5    5  2.29414E-06  9.41696E-01
];


DETThermalReflectedSlabFETCoarseMediumFETCART = [
-2.03550E+01  2.03550E+01    5
-2.29050E+01  2.29050E+01    5
 0.00000E+00  2.20600E+01    5
];
```

D.2 MOOSE Input Examples for FEs

File D.7 – An example MOOSE FE-based volumetric coupling. FXVolumeMain.i

```
# This is a simplistic simulation of a single fuel rod in TREAT. Many of the parameters were taken from:
# http://www.iaea.org/inis/collection/NCLCollectionStore/_Public/12/627/12627688.pdf
[Mesh]
  type = GeneratedMesh
  dim = 3
  nx = 100
  ny = 25
  nz = 25
  xmin = -60
  xmax = 60
  ymin = -5
  ymax = 5
  zmin = -5
  zmax = 5
[]

[Variables]
  [./Temperature]
    order = FIRST
    family = LAGRANGE
  [../]
[]

[ICs]
  [./T_IC]
    type = ConstantIC
    variable = Temperature
    value = 290
  [../]
[]

[Kernels]
  [./HeatDiff]
    type = HeatConduction
    variable = Temperature
  [../]
  [./HeadDiffTime]
    type = HeatConductionTimeDerivative
    variable = Temperature
  [../]
  [./HeatIn]
    type = BodyForce
    function = FX_Basis
    variable = Temperature
  [../]
[]

[Materials]
  [./Fuel] # Essentially graphite, from http://www.azom.com/article.aspx?ArticleID=1630
    type = GenericConstantMaterial
    prop_names =  'thermal_conductivity specific_heat density'
    prop_values = '2.475                0.800         1.8' # W/(cm K), J/(g K), g/cm^3
  [../]
[]

[Functions]
  [./FX_Basis]
    type = FunctionSeries
    series_type = Cartesian
```

```
    orders = '5   3   3'
    physical_bounds = '-60 60   -5 5    -5 5'
    x = Legendre
    y = Legendre
    z = Legendre
    enable_cache = true
  [../]
[]

[UserObjects]
  [./TemperatureFX]
    type = FXVolumeUserObject
    function = FX_Basis
    variable = Temperature
  [../]
[]

[Executioner]
  type = Transient
  num_steps = 40
  dt = 0.5
  solve_type = PJFNK
  petsc_options_iname = '-pc_type -pc_hypre_type'
  petsc_options_value = 'hypre boomeramg'
  picard_max_its = 30
  nl_rel_tol = 1e-8
  nl_abs_tol = 1e-9
  picard_rel_tol = 1e-8
  picard_abs_tol = 1e-9
[]

[MultiApps]
  [./FXTransferApp]
    type = TransientMultiApp
    input_files = FXVolumeSub.i
  [../]
[]

[Transfers]
  [./TemperatureToSub]
    type = MultiAppFXTransfer
    direction = to_multiapp
    multi_app = FXTransferApp
    this_app_object_name = TemperatureFX
    multi_app_object_name = FX_Basis
  [../]
  [./HeatToMe]
    type = MultiAppFXTransfer
    direction = from_multiapp
    multi_app = FXTransferApp
    this_app_object_name = FX_Basis
    multi_app_object_name = HeatGenerationFX
  [../]
[]

[Postprocessors]
  [./Elapsed_Time]
    type = PerformanceData
    event = ALIVE
    outputs = 'console csv'
  [../]
  [./Iterations:_Linear]
    type = NumLinearIterations
    outputs = 'csv'
  [../]
  [./Iterations:_Nonlinear]
```

```
    type = NumNonlinearIterations
    outputs = 'csv'
  [../]
  [./Iterations:_Picard]
    type = NumPicardIterations
    outputs = 'csv'
  [../]
  [./Temperature:_Average]
    type = ElementAverageValue
    variable = Temperature
  [../]
  [./Temperature:_Peak]
    type = ElementExtremeValue
    value_type = max
    variable = Temperature
  [../]
[]

[Outputs]
  file_base = test
  exodus = true
  [./csv]
    type = CSV
    file_base = csv/test
  []
[]
```

File D.8 – An example MOOSE MultiApp for FE-based volumetric coupling. FXVolumeSub.i 

```
# MultiApp used by FXVolumeMain.i
[Mesh]
  type = GeneratedMesh
  dim = 3
  nx = 100
  ny = 25
  nz = 25
  xmin = -60
  xmax = 60
  ymin = -5
  ymax = 5
  zmin = -5
  zmax = 5
[]

[Variables]
  [./NonCopyTransfersOnlyWorkWithAuxVariables]
  [../]
[]

[AuxVariables]
  [./Heat]
    order = FIRST
    family = LAGRANGE
  [../]
  [./Temperature]
    order = FIRST
    family = LAGRANGE
  [../]
[]

[Kernels]
  [./SoWeHaveToDefineAKernelAndVariableThatDoNothing]
    type = NullKernel
```

```
    variable = NonCopyTransfersOnlyWorkWithAuxVariables
  [../]
[]

[AuxKernels]
  [./ReconstructTemperature]
    type = FunctionSeriesToAux
    function = FX_Basis
    variable = Temperature
  [../]
  [./HeatGeneration]
    type = TREATHeatAux
    in = Temperature
    variable = Heat
    center = '0   0   0'
    total_energy = 19e9
    transient_duration = 10
    max_temperature = 900
    initial_temperature = 290
  [../]
[]

[Functions]
  [./FX_Basis]
    type = FunctionSeries
    series_type = Cartesian
    orders = '5   3   3'
    physical_bounds = '-60 60  -5 5  -5 5'
    x = Legendre
    y = Legendre
    z = Legendre
  [../]
[]

[UserObjects]
  [./HeatGenerationFX]
    type = FXVolumeUserObject
    function = FX_Basis
    variable = Heat
  [../]
[]

[Executioner]
  type = Transient
  num_steps = 40
  dt = 0.5
  solve_type = PJFNK
  petsc_options_iname = '-pc_type -pc_hypre_type'
  petsc_options_value = 'hypre boomeramg'
[]
```

File D.9 – An example MOOSE FE-based interface coupling. FXInterfaceMain.i

```
# This is a simplistic simulation of a fuel rod undergoing basic heat transfer
[Mesh]
  type = AnnularMesh
  rmax = 2.4
  rmin = 0.4
  nr = 30
  nt = 30
  growth_r = 0.95
  boundary_id = '0 1'
  boundary_name = 'inside outside'
[]
```

```
12
13 [MeshModifiers]
14   [./Extruder]
15     type = MeshExtruder
16     extrusion_vector = '0 0 3'
17     num_layers = 10
18     bottom_sideset = 'bottom'
19     top_sideset = 'top'
20   [../]
21 []
22
23 [Variables]
24   [./water_temp]
25   [../]
26 []
27
28 [Kernels]
29   [./diff_water_temp]
30     type = HeatConduction
31     variable = water_temp
32   [../]
33   [./time_diff_water_temp]
34     type = HeatConductionTimeDerivative
35     variable = water_temp
36   [../]
37 []
38
39 [Materials]
40   [./Water]
41     type = GenericConstantMaterial
42     prop_names =  'thermal_conductivity specific_heat density'
43     prop_values = '0.591                4.182         0.9983' # W/(cm K), J/(g K), g/cm^3
44     block = 0
45   [../]
46 []
47
48 [ICs]
49   [./start_water_temp]
50     type = ConstantIC
51     value = 290
52     variable = water_temp
53   [../]
54 []
55
56 [BCs]
57   [./hold_centerline_water_temp]
58     type = FunctionDirichletBC
59     variable = water_temp
60     boundary = outside
61     function = '290'
62   [../]
63   [./heat_from_fuel]
64     type = FXFluxBC
65     variable = water_temp
66     boundary = inside
67     function = FX_Basis_Water
68   [../]
69 []
70
71 [Functions]
72   [./FX_Basis_Water]
73     type = FunctionSeries
74     series_type = AxialShell
75     orders = '2 5'
76     physical_bounds = '0.0 3  0.0 0.0'
77     z = Legendre
```

```
    radial = Fourier
  [../]
[]

[UserObjects]
  [./FXTemperature]
    type = FXBoundaryValueUserObject
    function = FX_Basis_Water
    variable = water_temp
    boundary = inside
    # print_state = true
  [../]
[]

[MultiApps]
  [./FXTransferApp]
    type = TransientMultiApp
    input_files = FXInterfaceSub.i
    sub_cycling = true
    output_sub_cycles = true
  [../]
[]

[Transfers]
  [./TemperatureToSub]
   type = MultiAppFXTransfer
   direction = to_multiapp
   multi_app = FXTransferApp
   this_app_object_name = FXTemperature
   multi_app_object_name = FX_Basis_Fuel
  [../]
  [./HeatToMe]
    type = MultiAppFXTransfer
    direction = from_multiapp
    multi_app = FXTransferApp
    this_app_object_name = FX_Basis_Water
    multi_app_object_name = FXHeat
  [../]
[]

[Preconditioning]
  [./smp]
    type = SMP
    full = true
  [../]
[]

[Executioner]
  type = Transient
  num_steps = 40
  dt = 2.0
  solve_type = PJFNK
  petsc_options_iname = '-pc_type -pc_hypre_type'
  petsc_options_value = 'hypre boomeramg'
  nl_rel_tol = 1e-8
  nl_abs_tol = 1e-9
  picard_max_its = 60
  picard_rel_tol = 1e-8
  picard_abs_tol = 1e-9
[]

[Postprocessors]
  [./average_interface_temperature]
    type = SideAverageValue
    variable = water_temp
    boundary = inside
```

```
  [../]
  [./total_flux]
    type = SideFluxIntegral
    variable = water_temp
    boundary = inside
    diffusivity = 'thermal_conductivity'
  [../]
  [./z_picard_iterations]
    type = NumPicardIterations
    outputs = 'csv'
  [../]
[]

[Outputs]
  exodus = true
  file_base = test_water
  [./csv]
    type = CSV
    file_base = csv/test
  []
[]
```

File D.10 – An example MOOSE MultiApp for FE-based interface coupling. FXInterfaceSub.i 

```
# MultiApp file used by FXInterfaceMain.i
[Mesh]
  type = AnnularMesh
  rmax = 0.4
  rmin = 0.0
  nr = 10
  nt = 30
  growth_r = 0.9
  boundary_id = '0 1'
  boundary_name = 'inside outside'
[]

[MeshModifiers]
  [./Extruder]
    type = MeshExtruder
    extrusion_vector = '0 0 3'
    num_layers = 10
    bottom_sideset = 'bottom'
    top_sideset = 'top'
  [../]
[]

[Variables]
  [./fuel_temp]
  [../]
[]

[Kernels]
  [./diff_fuel_temp]
    type = HeatConduction
    variable = fuel_temp
  [../]
  [./time_diff_fuel_temp]
    type = HeatConductionTimeDerivative
    variable = fuel_temp
  [../]
  [./source_fuel_temp]
    type = BodyForce
    variable = fuel_temp
```

```
    function = '600 * (2 - sqrt((x - 0.2)^2 + (y + 0.2)^2))'
  [../]
[]

[Materials]
  [./Fuel] # Essentially UO2
    type = GenericConstantMaterial
    prop_names =  'thermal_conductivity specific_heat density'
    prop_values = '0.053                0.233         10.5' # W/(cm K), J/(g K), g/cm^3
  [../]
[]

[ICs]
  [./start_fuel_temp]
    type = ConstantIC
    value = 290
    variable = fuel_temp
  [../]
[]

[BCs]
  [./interface_temperature]
    type = FXValueBC
    variable = fuel_temp
    boundary = outside
    function = FX_Basis_Fuel
  [../]
[]

[Functions]
  [./FX_Basis_Fuel]
    type = FunctionSeries
    series_type = AxialShell
    orders = '2 5'
    physical_bounds = '0.0 3.0  0.0 0.0'
    z = Legendre
    radial = Fourier
  [../]
[]

[UserObjects]
  [./FXHeat]
    type = FXBoundaryFluxUserObject
    function = FX_Basis_Fuel
    variable = fuel_temp
    boundary = outside
    diffusivity = thermal_conductivity
    # print_state = true
  [../]
[]

[Preconditioning]
  [./smp]
    type = SMP
    full = true
  [../]
[]

[Executioner]
  type = Transient
  num_steps = 50
  dt = 2.0
  solve_type = PJFNK
  petsc_options_iname = '-pc_type -pc_hypre_type'
  petsc_options_value = 'hypre boomeramg'
[]
```

```
106
107 [Outputs]
108   exodus = true
109   file_base = test_fuel
110 []
```

D.3 MOOSE-Serpent Examples for FE-based Multiphysics Coupling

File D.11 – An example main MOOSE-Serpent input file. water.i

```
# This is a simplistic multiphysics simulation of a single fuel rod in a PWR
# This file simulates heat extraction by the coolant/moderator
[Mesh]
  type = AnnularMesh
  rmin = 0.60579
  rmax = 0.8
  nr = 5
  nt = 30
  growth_r = 0.9
  boundary_id = '0 1'
  boundary_name = 'inside outside'
[]

[MeshModifiers]
  [./Extruder]
    type = MeshExtruder
    extrusion_vector = '0 0 365.76'
    num_layers = 200
    bottom_sideset = 'bottom'
    top_sideset = 'top'
  [../]
  [./Relocator]
   type = Transform
   depends_on = Extruder
   transform = TRANSLATE
   vector_value = '0.23876 0.23876 -182.88'
  [../]
[]

[Variables]
  [./Temperature]
  [../]
[]

[Kernels]
  [./HeatDiff]
    type = HeatConduction
    variable = Temperature
  [../]
  [./HeadDiffTime]
    type = HeatConductionTimeDerivative
    variable = Temperature
  [../]
[]

[ICs]
  [./T_IC]
    type = FunctionIC
    variable = Temperature
    function = '340 - 10 * sqrt((x - 0.23876)^2 + (y - 0.23876)^2)'
  [../]
[]

[BCs]
  [./TemperatureIn]
    type = FXValuePenaltyBC
    variable = Temperature
    boundary = inside
```

```
    function = C_FX_Boundary_Basis
    penalty = 0.2
  [../]
  [./TemperatureOutside]
    type = ConvectiveFluxFunction
    variable = Temperature
    boundary = outside
    T_infinity = '330 + 30 * (z + 182.88) / 365.76'
    coefficient = 3.5
  [../]
[]

[Materials]
  [./Water]
    type = GenericConstantMaterial
    prop_names =  'thermal_conductivity specific_heat density'
    prop_values = '0.591                4.182         0.9983' # W/(cm K), J/(g K), g/cm^3
  [../]
[]

[Functions]
  [./C_FX_Boundary_Basis]
    type = FunctionSeries
    series_type = AxialShell
    orders = '13  7'
    physical_bounds = '-182.88 182.88   0.23876 0.23876'
    z = Legendre
    radial = Fourier
    enable_cache = true
  [../]
[]

[Preconditioning]
  [./smp]
    type = SMP
    full = true
  [../]
[]

[Executioner]
  type = Transient
  num_steps = 25
  dt = 0.5
  solve_type = PJFNK
  petsc_options_iname = '-pc_type -pc_hypre_type'
  petsc_options_value = 'hypre boomeramg'
  nl_rel_tol = 1e-8
  nl_abs_tol = 1e-9
  picard_max_its = 50
  picard_rel_tol = 1e-3
  picard_abs_tol = 5e-1
[]

[MultiApps]
  [./FuelPin]
    type = TransientMultiApp
    input_files = fuel_pin.i
    tolerate_failure = true
    output_sub_cycles = true
    execute_on = TIMESTEP_END
  [../]
[]

[Postprocessors]
  [./Elapsed_Time]
    type = PerformanceData
```

```
    event = ALIVE
    outputs = 'console csv'
  [../]
  [./Iterations:_Linear]
    type = NumLinearIterations
    outputs = 'csv'
  [../]
  [./Iterations:_Nonlinear]
    type = NumNonlinearIterations
    outputs = 'csv'
  [../]
  [./Iterations:_Picard]
    type = NumPicardIterations
    outputs = 'csv'
  [../]
  [./Flux:_Total]
    type = SideFluxIntegral
    diffusivity = 'thermal_conductivity'
    boundary = rmin
    variable = Temperature
  [../]
[]

[UserObjects]
  [./C_TemperatureBoundaryFX]
    type = FXBoundaryFluxUserObject
    function = C_FX_Boundary_Basis
    diffusivity = 'thermal_conductivity'
    boundary = inside
    variable = Temperature
  [../]
[]

[Transfers]
  [./HeatFluxToFuelPin]
    type = MultiAppFXTransfer
    direction = to_multiapp
    multi_app = FuelPin
    multi_app_object_name = F_FX_Boundary_Basis
    this_app_object_name = C_TemperatureBoundaryFX
  [../]
  [./TemperatureToMe]
    type = MultiAppFXTransfer
    direction = from_multiapp
    multi_app = FuelPin
    multi_app_object_name = F_TemperatureBoundaryFX
    this_app_object_name = C_FX_Boundary_Basis
  [../]
[]

[Outputs]
  file_base = water
  exodus = true
  [./csv]
    type = CSV
    file_base = csv/water
  []
[]
```

File D.12 – An example MOOSE-Serpent MultiApp input file. fuel_pin.i

```
# This MultipApp is used by water.i
# This file simulates heat conduction in the fuel rod
[Mesh]
```

```
  type = AnnularMesh
  rmax = 0.60579
  rmin = 0.0
  nr = 16
  nt = 20
  growth_r = 0.9
[]

[MeshModifiers]
  [./Extruder]
    type = MeshExtruder
    extrusion_vector = '0 0 365.76'
    num_layers = 200
    bottom_sideset = 'bottom'
    top_sideset = 'top'
  [../]
  [./Relocator]
   type = Transform
   depends_on = Extruder
   transform = TRANSLATE
   vector_value = '0.23876 0.23876 -182.88'
  [../]
[]

[Variables]
  [./Temperature]
    order = FIRST
    family = LAGRANGE
  [../]
[]

[Kernels]
  [./HeatDiff]
    type = HeatConduction
    variable = Temperature
  [../]
  [./HeadDiffTime]
    type = HeatConductionTimeDerivative
    variable = Temperature
  [../]
  [./HeatIn]
    type = BodyForce
    variable = Temperature
    function = F_FX_Volume_Basis
  [../]
  # [./source_fuel_temp]
  #   type = BodyForce
  #   variable = Temperature
  #   function = '600 * cos(z / 160)^0.25'
  # [../]
[]

[ICs]
  [./T_IC]
    type = ConstantIC
    variable = Temperature
    value = 330
  [../]
[]

[BCs]
  [./HeatOut]
    type = FXFluxBC
    variable = Temperature
    boundary = rmax
    function = F_FX_Boundary_Basis
```

```
  [../]
[]

[AuxVariables]
  [./Heat]
    order = FIRST
    family = LAGRANGE
  [../]
[]

[AuxKernels]
  [./HeatToAux]
    type = FunctionSeriesToAux
    function = F_FX_Volume_Basis
    variable = Heat
  [../]
[]

[Materials]
  [./Fuel] # Essentially UO2
    type = GenericConstantMaterial
    prop_names =  'thermal_conductivity specific_heat density'
    prop_values = '0.053                0.233         10.5' # W/(cm K), J/(g K), g/cm^3
  [../]
[]

[Functions]
  [./F_FX_Boundary_Basis]
    type = FunctionSeries
    series_type = AxialShell
    orders = '13  7'
    physical_bounds = '-182.88 182.88   0.23876 0.23876'
    z = Legendre
    radial = Fourier
    enable_cache = true
  [../]
  [./F_FX_Volume_Basis]
    type = FunctionSeries
    series_type = CylindricalDuo
    orders = '11  5'
    physical_bounds = '-182.88 182.88   0.23876 0.23876   0.60579'
    z = Legendre
    disc = Zernike
    enable_cache = true
  [../]
[]

[Executioner]
  type = Transient
  num_steps = 25
  dt = 0.5
  solve_type = PJFNK
  petsc_options_iname = '-pc_type -pc_hypre_type'
  petsc_options_value = 'hypre boomeramg'
  nl_rel_tol = 1e-8
  nl_abs_tol = 1e-9
  picard_max_its = 50
  picard_rel_tol = 1e-3
  picard_abs_tol = 5e-1
[]

[Postprocessors]
  [./Iterations:_Linear]
    type = NumLinearIterations
    outputs = 'csv'
  [../]
```

```
  [./Iterations:_Nonlinear]
    type = NumNonlinearIterations
    outputs = 'csv'
  [../]
  [./Iterations:_Picard]
    type = NumPicardIterations
    outputs = 'csv'
  [../]
  [./Temperature:_Average]
    type = ElementAverageValue
    variable = Temperature
  [../]
  [./Temperature:_Peak]
    type = ElementExtremeValue
    value_type = max
    variable = Temperature
  [../]
  [./Flux:_Total]
    type = SideFluxIntegral
    diffusivity = 'thermal_conductivity'
    boundary = rmax
    variable = Temperature
  [../]
[]

[MultiApps]
  [./Serpent]
    type = TransientMultiApp
    input_files = serpent.i
    sub_cycling = true
    execute_on = TIMESTEP_END
  [../]
[]

[UserObjects]
  [./F_TemperatureVolumeFX]
    type = FXVolumeUserObject
    function = F_FX_Volume_Basis
    variable = Temperature
  [../]
  [./F_TemperatureBoundaryFX]
    type = FXBoundaryValueUserObject
    function = F_FX_Boundary_Basis
    boundary = rmax
    variable = Temperature
  [../]
[]

[Transfers]
  [./TemperatureToSerpent]
    type = MultiAppFXTransfer
    direction = to_multiapp
    multi_app = Serpent
    multi_app_object_name = SerpentExec
    this_app_object_name = F_TemperatureVolumeFX
  [../]
  [./SerpentHeatToMe]
    type = MultiAppFXTransfer
    direction = from_multiapp
    multi_app = Serpent
    multi_app_object_name = SerpentExec
    this_app_object_name = F_FX_Volume_Basis
  [../]
[]

[Outputs]
```

```
  file_base = fuel_pin
  exodus = true
  [./csv]
    type = CSV
    file_base = csv/fuel_pin
  []
[]
```

File D.13 – An example MOOSE-Serpent SerpentExecutioner input file. serpent.i

```
# This MultiApp is used by fuel_pin.i
# The file is used to configure the multiphysics interface in Serpent to produce the heat generation
# term from the fission power distribution
[Mesh]
  type = GeneratedMesh
  dim = 1
[]

[Variables]
  [./dummy]
  [../]
[]

[Executioner]
  type = SerpentExecutioner
  name = SerpentExec
  series_type = CylindricalDuo
  orders = '11  5'
  physical_bounds = '-182.88 182.88   0.23876 0.23876   0.60579'
  z = Legendre
  disc = Zernike
  serpent_fission_power_outermost_material = 'cool'
  serpent_otf_material = 'fuel3'
  serpent_temperature_file = 'temps'
  serpent_input = 'basic.sss'
  average_power_level_const = 71.094 # 30,000 W per pin (3 MW per 10x10 assembly in a ~2400 MWt core)
  serpent_initial_constant_fe_value = 330
  export_relaxation_type = windowed
  export_relaxation_value = 4
  import_relaxation_type = windowed
  import_relaxation_value = 60
[]

[Problem]
  kernel_coverage_check = false
  solve = false
[]
```

File D.14 – An example of the modified standard Serpent input file for MOOSE-Serpent coupling. basic.sss.moose

```
set title "Serpent-MOOSE internally-coupled calculation"

include geometry.inp
include materials.inp

% --- Cross section library file path:
set acelib sss_endfb7u.xsdata

% --- Reflective boundary condition in XY, vacuum in Z
```

```
set bc 2 2 1

% --- Neutron population and criticality cycles:
set pop 5000 200 400 1.0
set fsp 1 50

% --- Geometry plots:
%plot 3 1000 1000 0.0
%plot 3 1000 1000 0.0 -0.36703 0.84455 -0.36703 0.84455
%plot 3 1000 1000 0.0 -7.62 7.62 -7.62 7.62
%plot 2 500 1500

% --- Do not generate group constants
set gcu -1

% --- Do not use implicit capture, nxn, or fission
set impl 0 0 0


%% ==== START ==== Autogenerated multiphysics section ==== START ====
%%  Multiphysics interface file
ifc temps.ifc

%%  Multiphysics signaling
set ppid 666
set ccmaxiter 999
set power 1.0

%% ====  END  ==== Autogenerated multiphysics section ====  END  ====
```

File D.15 – A truncated example of the interface file generated by SerpentExecutioner. temps.ifc

```
32 2 fuel3 1 cool
fet.m0.pwr 1 0.60579 5 -182.88 182.88 11 3 0.23876 0.23876
495.311
12.5695
-140.029
0.371579
11.1761
-0.244273
2.01364
0.150076
...
0.016904
```

File D.16 – A truncated example of the Serpent multiphysics FET output. fet.pwr

```
Type                 R_Min         R_Max   R_Order         A_Min         A_Max   A_Order     #_Regions
Cylindrical    0.00000E+00   6.05790E-01         5 -1.82880E+02   1.82880E+02        11         1

Region   Linear_Index   R_Order   R_Rank   A_Order   Coefficient   Relative_Uncertainty
0        0                    0        0         0   3.20380E-14   1.07196E-03
0        1                    0        0         1  -1.36682E-15   4.80534E-02
0        2                    0        0         2  -2.81120E-14   3.09343E-03
0        3                    0        0         3   1.42919E-15   7.62343E-02
0        4                    0        0         4   9.76176E-16   1.26788E-01
0        5                    0        0         5  -9.19046E-17   1.48099E+00
...
0        251                  5        5        11   1.33632E-16   2.70405E+00
```

Appendix E

Chrysalis MooseApp Code

This appendix contains the code developed for the `chrysalis` MooseApp. These include extra `kernels` and such that were used to create the multiphysics effects for the testing simulations. It also includes the code for `SerpentExecutioner` and `SerpentTimeStepper`, the classes used to create a MOOSE-wrapped application. Finally, the makefile and script, for building Serpent as a library along-side MOOSE, are provided. Much of this is also available at `https://github.com/BruceWyxv/chrysalis/tree/SerpentCoupling`.

E.1 AuxKernels

File E.1 – An AuxKernel that provides temperature-dependent heat generation in a cylindrical region, similar to a fuel pin in a nuclear reactor. FuelPinHeatAux.h 

```
#ifndef FUELPINHEATAUX_H
#define FUELPINHEATAUX_H

#include "AuxKernel.h"

// Forward Declarations
class FuelPinHeatAux;

template <>
InputParameters validParams<FuelPinHeatAux>();

/**
 * This AuxKernel implements a heating term that generalizes the behavior of the
 * TREAT reactor during a transient event in a single core element
 *
 * The axial direction of the fuel element is assumed to be in 'x'
 */
class FuelPinHeatAux : public AuxKernel
{
public:
  FuelPinHeatAux(const InputParameters & parameters);

  const Real _element_length;
  const Real _element_length_half;
  const Real _element_widths;
  const Real _element_widths_half;
  const Real _total_volume;
  const Real _zero_power_radius;

protected:
  virtual Real computeValue() override;

  const VariableValue & _in;

  const Point _center;
  const Real _total_energy;
  const Real _max_heating_rate;
  const Real _max_temperature;
  const Real _initial_temperature;
};

#endif // FUELPINHEATAUX_H
```

File E.2 – An AuxKernel that provides temperature-dependent heat generation in a cylindrical region, similar to a fuel pin in a nuclear reactor. FuelPinHeatAux.C 

```
#include "FuelPinHeatAux.h"

#include "math.h"

registerMooseObject("ChrysalisApp", FuelPinHeatAux);

template <>
InputParameters
validParams<FuelPinHeatAux>()
{
```

```
  InputParameters params = validParams<AuxKernel>();

  params.addRequiredCoupledVar("in", "The temperature of the system");

  params.addRequiredParam<Point>("center", "The center of the fuel element");
  params.addRangeCheckedParam<Real>(
      "total_energy",
      "total_energy > 1",
      "The total energy that can be generated during a transient event");
  params.addRangeCheckedParam<Real>("max_temperature",
                                    "max_temperature <= 925",
                                    "The maximum design temperature of the transient");
  params.addRangeCheckedParam<Real>(
      "initial_temperature",
      "initial_temperature >= 200",
      "The initial temperature (at which heat generation is maximum)");

  return params;
}

FuelPinHeatAux::FuelPinHeatAux(const InputParameters & parameters)
  : AuxKernel(parameters),
    _element_length(5),
  ↳// cm
    _element_length_half(_element_length / 2.0),
  ↳// cm
    _element_widths(2),
  ↳// cm
    _element_widths_half(_element_widths / 2.0),
  ↳// cm
    _total_volume(M_PI * _element_widths_half * _element_widths_half * _element_length),
  ↳// cm^3
    _zero_power_radius(1.5 * _element_widths),
    _in(coupledValue("in")),
    _center(getParam<Point>("center")),
    _total_energy(getParam<Real>("total_energy")),
    _max_heating_rate(_total_energy / _total_volume),
    _max_temperature(getParam<Real>("max_temperature")),
    _initial_temperature(getParam<Real>("initial_temperature"))
{
}

Real
FuelPinHeatAux::computeValue()
{
  const Point & point = isNodal() ? *_current_node : _q_point[_qp];
  const Real x = point(0) - _center(0);
  const Real y = point(1) - _center(1);
  const Real z = abs(point(2) - _center(2));

  const Real axial_factor = cos(M_PI * pow(z / _element_length_half, 4) / 3);
  const Real radial_factor = 1 - sqrt(x * x + y * y) / _zero_power_radius;
  const Real neutronic_heating =
      exp(-2.30258509299 * (_in[_qp] - _initial_temperature) /
          (_max_temperature - _initial_temperature)); // 10% heat generation at _max_temperature

  return _max_heating_rate * neutronic_heating * axial_factor * radial_factor;
}
```

File E.3 – An AuxKernel that actually does nothing, but is needed from time to time to prevent MOOSE from worrying about an AuxVariable used for caching or other purposes.

KeepItTheSameAux.h

```
#ifndef KEEPITTHESAMEAUX_H
```

```
#define KEEPITTHESAMEAUX_H

#include "AuxKernel.h"

// Forward Declarations
class KeepItTheSameAux;

template <>
InputParameters validParams<KeepItTheSameAux>();

class KeepItTheSameAux : public AuxKernel
{
public:
  KeepItTheSameAux(const InputParameters & parameters);

protected:
  virtual Real computeValue() override;
};

#endif // KEEPITTHESAMEAUX_H
```

File E.4 – An AuxKernel that actually does nothing, but is needed from time to time to prevent MOOSE from worrying about an AuxVariable used for caching or other purposes. KeepItTheSameAux.C

```
#include "KeepItTheSameAux.h"

registerMooseObject("ChrysalisApp", KeepItTheSameAux);

template <>
InputParameters
validParams<KeepItTheSameAux>()
{
  InputParameters params = validParams<AuxKernel>();

  return params;
}

KeepItTheSameAux::KeepItTheSameAux(const InputParameters & parameters) : AuxKernel(
  ↳parameters)
{
  // Nothing here
}

Real
KeepItTheSameAux::computeValue()
{
  return _u[_qp];
}
```

File E.5 – An AuxKernel that provides temperature-dependent heat generation in a Cartesian region, similar to a fuel element in TREAT. TREATHeatAux.h

```
#ifndef TREATHEATAUX_H
#define TREATHEATAUX_H

#include "AuxKernel.h"

// Forward Declarations
class TREATHeatAux;

template <>
```

```
InputParameters validParams<TREATHeatAux>();

/**
 * This AuxKernel implements a heating term that generalizes the behavior of the
 * TREAT reactor during a transient event in a single core element
 *
 * The axial direction of the fuel element is assumed to be in 'x'
 */
class TREATHeatAux : public AuxKernel
{
public:
  TREATHeatAux(const InputParameters & parameters);

  const Real _element_length;
  const Real _element_length_half;
  const Real _element_widths;
  const Real _element_widths_half;
  const unsigned int _number_of_elements;
  const Real _total_volume;
  const Real _zero_power_radius;

protected:
  virtual Real computeValue() override;

  const VariableValue & _in;

  const Point _center;
  const Real _total_energy;
  const Real _transient_duration;
  const Real _front_edge;
  const Real _max_heating_rate;
  const Real _max_temperature;
  const Real _initial_temperature;
};

#endif // TREATHEATAUX_H
```

File E.6 – An AuxKernel that provides temperature-dependent heat generation in a Cartesian region, similar to a fuel element in TREAT. TREATHeatAux.C

```
#include "TREATHeatAux.h"

#include "math.h"

registerMooseObject("ChrysalisApp", TREATHeatAux);

template <>
InputParameters
validParams<TREATHeatAux>()
{
  InputParameters params = validParams<AuxKernel>();

  params.addRequiredCoupledVar("in", "The temperature of the system");

  params.addRequiredParam<Point>("center", "The center of the fuel element");
  params.addRangeCheckedParam<Real>(
      "total_energy",
      "total_energy > 1",
      "The total energy that can be generated during a transient event");
  params.addRangeCheckedParam<Real>(
      "transient_duration", "transient_duration > 0.25", "The duration of the transient
  ↳event");
  params.addRangeCheckedParam<Real>("max_temperature",
                                    "max_temperature <= 925",
```

```
                                    "The maximum design temperature of the transient");
  params.addRangeCheckedParam<Real>(
      "initial_temperature",
      "initial_temperature >= 200",
      "The initial temperature (at which heat generation is maximum)");

  return params;
}

TREATHeatAux::TREATHeatAux(const InputParameters & parameters)
  : AuxKernel(parameters),
    _element_length(120),                          // cm
    _element_length_half(_element_length / 2.0), // cm
    _element_widths(10),                           // cm
    _element_widths_half(_element_widths / 2.0), // cm
    _number_of_elements(19 * 19),
    _total_volume(_number_of_elements * _element_widths * _element_widths *
                  _element_length), // cm^3
    _zero_power_radius(5 * sqrt(_element_widths * _element_widths)),
    _in(coupledValue("in")),
    _center(getParam<Point>("center")),
    _total_energy(getParam<Real>("total_energy")),
    _transient_duration(getParam<Real>("transient_duration")),
    _front_edge(_transient_duration / 3.0),
    _max_heating_rate(_total_energy / (_total_volume * _transient_duration)),
    _max_temperature(getParam<Real>("max_temperature")),
    _initial_temperature(getParam<Real>("initial_temperature"))
{
}

Real
TREATHeatAux::computeValue()
{
  const Point & point = isNodal() ? *_current_node : _q_point[_qp];
  const Real x = point(0) - _center(0);
  const Real y = point(1) - _center(1) + _element_widths_half;
  const Real z = point(2) - _center(2) + _element_widths_half;

  const Real axial_factor = cos(M_PI * pow(x / _element_length_half, 4) / 3);
  const Real radial_factor = 1 - sqrt(y * y + z * z) / _zero_power_radius;
  const Real neutronic_heating =
      exp(-2.30258509299 * (_in[_qp] - _initial_temperature) /
          (_max_temperature - _initial_temperature)); // 10% heat generation at _max_temperature
  const Real ramping =
      (_t <= _front_edge) ? (_t / _front_edge) : ((_t < _transient_duration) ? 1.0 : 1e-4);

  return _max_heating_rate * ramping * neutronic_heating * axial_factor * radial_factor;
}
```

E.2 Executioners

File E.7 – The Executioner for transferring data with Serpent via FEs by reading and writing coefficient data in the interface files. SerpentExecutioner.h

```h
#ifndef SERPENTEXECUTIONER_H
#define SERPENTEXECUTIONER_H

#include "FunctionInterface.h"

#include "FXExecutioner.h"

class Function;
class SerpentExecutioner;
class SerpentTimeStepper;

template <>
InputParameters validParams<SerpentExecutioner>();

/**
 * This class is responsible for orchestrating the interaction with Serpent by:
 *   1) Ensuring that the Serpent input file exists
 *   2) Generating the power/density multiphysics input files
 *   3) Parsing the fission power density output file from each iteration
 */
class SerpentExecutioner : public FXExecutioner, FunctionInterface
{
public:
  SerpentExecutioner(const InputParameters & parameters);

  // Overrides from FXExecutioner
  virtual Real getNormalization(const std::vector<Real> & _coefficients) const override;
  virtual std::size_t mapIndexingMooseToExternal(const std::size_t mooose_fe_index) const
  override;

protected:
  /**
   * Writes the coefficients to the Serpent multiphysics input files
   */
  virtual void exportCoefficients(const FixedSizeArray<Real> & out_coefficients) override
  ;

  /**
   * Extracts the coefficients generated from Serpent and loads them into the provided array
   */
  virtual void importCoefficients(FixedSizeArray<Real> & array_to_fill) override;

  /**
   * Sets the default TimeStepper to SerpentTimeStepper if needed, then calls Transient::init()
   */
  virtual void init() override;

  /// Name of the multiphysics interface file for density
  const std::string _serpent_interface_density_file_name;

  /// Name of the multiphysics interface file for temperatures
  const std::string _serpent_interface_temperature_file_name;

  /// The initial constant value that will be fed into Serpent
  const Real _serpent_initial_constant_fe_value;

  /// The name of the material in Serpent that will have its cross-sections adjusted on-the-fly
  /// using the exported FE
  const std::string _serpent_otf_material;

```

```
  /// The name of the material in Serpent that is the outermost layer of a pin definition, used to
  /// identify which pins should be scored for the fission power density FET
  const std::string _serpent_fission_power_outermost_material;

  /// Name of the main Serpent input file name
  const std::string _serpent_input_template_file_name;

  /// Name of the multiphysics interface file for fission power density
  const std::string _serpent_interface_fission_power_density_file_name;

  /// Keeps all the files by creating files with the following format: file_name_#S_#P, where #S is
  /// the current step and #P is the number of Picard iterations at time step #P
  const bool _keep_files;

  /// A boolean that specifies where to request the power density FETs
  const bool _request_fission_power_in_density_file;

  /// A boolean that defines whether the power level is an average or scaling
  const bool _is_power_level_an_average;

  /// A boolean that defines whether the power level is constant or time-varying
  const bool _is_power_level_time_varying;

  /// A fixed value that defines the power level
  const Real _const_power_level;

  /// A Function that defines a power level
  Function * _function_power_level;

  /**
   * Writes the main interface file for Serpent
   */
  void generateSerpentInterfaceFile(const std::string & interface_base,
                                    const FixedSizeArray<Real> & out_coefficients) const;

  /**
   * Writes the main input file for Serpent
   */
  void generateSerpentInputFile(const std::string & interface_base) const;

  /**
   * Get the name of the output file that contains the fission power density FET from Serpent
   */
  std::string getFissionPowerDensityFileName(bool generate_history_filename) const;

  /**
   * Returns the time stepper as a SerpentTimeStepper reference
   */
  const SerpentTimeStepper & getMySerpentTimeStepper() const;

  /**
   * Get the string that corresponds to the keep_files file name tracking component
   */
  std::string getTrackingFileNameComponent() const;
};

#endif // SERPENTEXECUTIONER_H
```

File E.8 – The Executioner for transferring data with Serpent via FEs by reading and writing coefficient data in the interface files. SerpentExecutioner.C

```
#include <atomic>
#include <fstream>
#include <iostream>
```

```
4
5  #include "Function.h"
6
7  #include "SerpentExecutioner.h"
8  #include "SerpentTimeStepper.h"
9
10 // Include the Serpent header file
11 #include "header.h"
12 #define STR_HELPER(x) #x
13 #define STR(x) STR_HELPER(x)
14
15 registerMooseObject("ChrysalisApp", SerpentExecutioner);
16
17 template <>
18 InputParameters
19 validParams<SerpentExecutioner>()
20 {
21   InputParameters params = validParams<FXExecutioner>();
22
23   params += validParams<MutableCoefficientsInterface>();
24
25   params.addClassDescription("Executioner for coupling to the Serpent Reactor Physics MC
     code");
26   params.set<std::string>("_object_name", "SerpentExecutioner");
27
28   /*
29    * Files for interfacing
30    */
31   params.addParam<std::string>("serpent_input",
32                                "Name of the main Serpent input file to be used as a
     template for "
33                                "creating the multiphysics interface. Namely, the
     communication "
34                                "settings and the \"ifc ...\" lines will be appended to a
     duplicate "
35                                "file using the name \"'serpent_input'.moose\".");
36   params.addParam<std::string>("serpent_density_file",
37                                "",
38                                "This file will be generated by SerpentExecutioner, and is
     the name "
39                                "of the multiphysics file that Serpent expects to find the
     density "
40                                "coefficients in. This corresponds to the line \"ifc "
41                                "'file_name'.ifc\" in the Serpent input, with an interface
     type of "
42                                "'" STR(IFC_TYPE_FET_DENSITY) "'.");
43   params.addParam<std::string>("serpent_temperature_file",
44                                "",
45                                "This file will be generated by SerpentExecutioner, and is
     the name "
46                                "of the multiphysics file that Serpent expects to find the
     "
47                                "temperature values in. This corresponds to the line \"ifc
     "
48                                "'file_name'.ifc\" in the Serpent input, with an interface
     type of "
49                                "'" STR(IFC_TYPE_FET_TEMP) "'.");
50   params.addParam<std::string>("serpent_fission_power_file",
51                                "fet",
52                                "The name of the file to which Serpent should write the
     fission "
53                                "power FETs as \"'file_name'.pwr\".");
54   params.addParamNamesToGroup("serpent_input serpent_density_file
     serpent_temperature_file "
55                               "serpent_fission_power_file",
56                               "Interface Files");
```

```
57
58    /*
59     * Serpent interface materials
60     */
61    params.addRequiredParam<std::string>(
62        "serpent_otf_material",
63        "The name of the Serpent material that will be affected on-the-fly by the exported FE.");
64    params.addRequiredParam<std::string>("serpent_fission_power_outermost_material",
65                                         "The outermost material of a pin structure in Serpent, used "
66                                         "to identify the pin(s) where the FET power scoring should "
67                                         "be performed.");
68    params.addParamNamesToGroup("serpent_otf_material serpent_fission_power_outermost_material",
69                                "Serpent Interface Materials");
70
71    /*
72     * Scaling of the fission power density
73     */
74    params.addParam<Real>(
75        "average_power_level_const",
76        "Average power to which the fission power density FETs should be normalized.");
77    params.addParam<Real>("scale_power_level_const",
78                          "Factor by which the imported Serpent FET should be scaled.");
79    params.addParam<FunctionName>("average_power_level_function",
80                                  "A function describing the time-dependent average power to which "
81                                  "the fission power density FETs should be normalized.");
82    params.addParam<FunctionName>("scale_power_level_function",
83                                  "A function describing the time-dependent value by which the "
84                                  "fission power density FETs should be scaled.");
85    params.addParamNamesToGroup("average_power_level_const scale_power_level_const "
86                                "average_power_level_function scale_power_level_function",
87                                "Power Scaling");
88
89    /*
90     * Misc parameters
91     */
92    params.addParam<bool>(
93        "keep_files", false, "Keep all the files generated for manual inspection later.");
94    params.addParam<Real>(
95        "serpent_initial_constant_fe_value",
96        "The value (density fraction or temperature in K) that the initial FE to Serpent should use. "
97        "In not provided, this will default to 1.0 for density fraction or 290 K for temperature.");
98    params.addParamNamesToGroup("keep_files serpent_initial_constant_fe_value", "Misc");
99
100   /*
101    * Parallel processing options
102    */
103 #ifdef SERPENT_OPENMP_AVAILABLE
104   params.addParam<int>("serpent_omp_threads",
105                        -1,
106                        "The number of OpenMP threads with which to run Serpent. A value of '-1' "
107                        "will default to the number of threads used for this MultiApp. NOTE: this "
108                        "value will be ignored if SerpentTimeStepper is explicitly defined.");
109   params.addParamNamesToGroup("serpent_omp_threads", "Serpent Execution Options");
110 #endif // SERPENT_OPENMP_AVAILABLE
111
```

```
112   /*
113    * Hide meaningless parameters from the user in this context of wraping Serpent
114    */
115   params.suppressParameter<unsigned int>("picard_max_its");
116   params.suppressParameter<Real>("picard_rel_tol");
117   params.suppressParameter<Real>("picard_abs_tol");
118   params.suppressParameter<Real>("relaxation_factor");
119   params.suppressParameter<std::vector<std::string>>("relaxed_variables");
120
121   return params;
122 }
123
124 SerpentExecutioner::SerpentExecutioner(const InputParameters & parameters)
125   : FXExecutioner(parameters),
126     FunctionInterface(this),
127     _serpent_interface_density_file_name(getParam<std::string>("serpent_density_file")),
128     _serpent_interface_temperature_file_name(getParam<std::string>("
  ↳serpent_temperature_file")),
129     _serpent_initial_constant_fe_value(
130         isParamValid("serpent_initial_constant_fe_value")
131             ? getParam<Real>("serpent_initial_constant_fe_value")
132             : (!_serpent_interface_density_file_name.empty() ? 1.0 : 290.0)),
133     _serpent_otf_material(getParam<std::string>("serpent_otf_material")),
134     _serpent_fission_power_outermost_material(
135         getParam<std::string>("serpent_fission_power_outermost_material")),
136     _serpent_input_template_file_name(getParam<std::string>("serpent_input")),
137     _serpent_interface_fission_power_density_file_name(
138         getParam<std::string>("serpent_fission_power_file")),
139     _keep_files(""),
140     _request_fission_power_in_density_file(!_serpent_interface_density_file_name.empty())
  ↳,
141     _is_power_level_an_average(isParamValid("average_power_level_const") ||
142                                isParamValid("average_power_level_function")),
143     _is_power_level_time_varying(isParamValid("average_power_level_function") ||
144                                  isParamValid("scale_power_level_function")),
145     _const_power_level(_is_power_level_time_varying
146                            ? -1.0
147                            : (_is_power_level_an_average
148                                   ? getParam<Real>("average_power_level_const")
149                                   : (isParamValid("scale_power_level_const")
150                                          ? getParam<Real>("scale_power_level_const")
151                                          : 1.0)))
152 {
153   /*
154    * Ensure sanity for the interface file names
155    */
156   // Ensure that a Serpent interface file was specified
157   if (_serpent_interface_density_file_name.empty() &&
158       _serpent_interface_temperature_file_name.empty())
159     mooseError("Either \"serpent_density_file\" or \"serpent_temperature_file\" must be
  ↳set!");
160   // An error if both interface files are defined
161   if (!_serpent_interface_density_file_name.empty() &&
162       !_serpent_interface_temperature_file_name.empty())
163     mooseError("Sorry, only one FE type can currently be exported. Please select only '
  ↳density' or "
164                "'temperature'.");
165
166   /*
167    * Ensure sanity of the power levels
168    */
169   // If we are using a constant power level then ensure that it is sensible
170   if (_const_power_level <= 0.0 && !_is_power_level_time_varying)
171   {
172     if (_is_power_level_an_average)
173       paramError("average_power_level_const", "Cannot have a negative power level!");
```

```
174     else
175       paramError("scale_power_level_const", "Cannot have a negative power level!");
176   }
177
178   /*
179    * Ensure that we don't have multiple power level specifications
180    */
181   // Check to see if both constant options are defined. If both are defined then we will assume that
182   // the average form is preferred and flag the scale definition.
183   if (isParamValid("average_power_level_const") && isParamValid("scale_power_level_const"))
184     paramError("scale_power_level_const",
185                "Both averaging and scaling power levels constants are defined but only one can be "
186                "used. Please remove or comment-out the parameter that is not to be used.");
187   // Check to see if both function options are defined. If both are defined then we will assume that
188   // the average form is preferred and flag the scale definition.
189   if (isParamValid("average_power_level_function") && isParamValid("scale_power_level_function"))
190     paramError("scale_power_level_function",
191                "Both averaging and scaling power levels functions are defined but only one can be "
192                "used. Please remove or comment-out the parameter that is not to be used.");
193   // Check to see if at least one constant and function are both defined. If both are defined then
194   // we will assume that the function-based form is the preferred option and flag the constant.
195   if (_const_power_level > 0.0 && _is_power_level_time_varying)
196   {
197     if (isParamValid("average_power_level_const"))
198     {
199       if (isParamValid("average_power_level_function"))
200         paramError("average_power_level_const",
201                    "Both constant and function-based averaging power levels are defined but only "
202                    "one can be used. Please remove of comment-out the parameter that is not to be "
203                    "used.");
204       else // isParamValid("scale_power_level_function")
205         paramError("average_power_level_const",
206                    "Both averaging constant and scaling function-based power levels are defined "
207                    "but only one can be used. Please remove of comment-out the parameter that is "
208                    "not to be used.");
209     }
210     else // isParamValid("scale_power_level_const")
211     {
212       if (isParamValid("average_power_level_function"))
213         paramError("scale_power_level_const",
214                    "Both scaling constant and averaging function-based power levels are defined "
215                    "but only one can be used. Please remove of comment-out the parameter that is "
216                    "not to be used.");
217       else // isParamValid("scale_power_level_function")
218         paramError("scale_power_level_const",
219                    "Both constant and function-based scaling power levels are defined but only one "
220                    "can be used. Please remove of comment-out the parameter that is not to be "
221                    "used.");
222     }
223   }
224
225   // Generate the initial Serpent input files
```

```
226   const std::string interface_base =
227       (_request_fission_power_in_density_file ? _serpent_interface_density_file_name
228                                               : _serpent_interface_temperature_file_name) +
229       ".ifc";
230   generateSerpentInputFile(interface_base);
231   FixedSizeArray<Real> seed_coefficients(getCoefficients().size(), 0.0);
232   seed_coefficients[0] = _serpent_initial_constant_fe_value;
233   generateSerpentInterfaceFile(interface_base, seed_coefficients);
234 }
235
236 void
237 SerpentExecutioner::init()
238 {
239   /*
240    * Get the function pointer if needed.
241    *
242    * This can't be done in the initialization list because Functions haven't been constructed yet.
243    */
244   if (_is_power_level_time_varying)
245   {
246     if (_is_power_level_an_average)
247       _function_power_level =
248           &getFunctionByName(getParam<FunctionName>("average_power_level_function"));
249     else
250       _function_power_level =
251           &getFunctionByName(getParam<FunctionName>("scale_power_level_function"));
252   }
253
254   /*
255    * Generate the SerpentTimeStepper if the user didn't already define in. Most of the time the
256    * default values for SerpentTimeStepper are sufficient, so defining it here is expected to be the
257    * primary use case.
258    */
259   if (!_time_stepper.get())
260   {
261     /*
262      * Override the default time stepper behavior here, copied from Transient
263      */
264     InputParameters pars = _app.getFactory().getValidParams("SerpentTimeStepper");
265     pars.set<SubProblem *>("_subproblem") = &_problem;
266     pars.set<FEProblemBase *>("_fe_problem_base") = &_problem;
267     pars.set<Transient *>("_executioner") = this;
268     pars.set<bool>("reset_dt") = getParam<bool>("reset_dt");
269
270     /*
271      * Add in the required file options for starting and interacting with Serpent
272      */
273     if (!isParamValid("serpent_input"))
274       mooseError("The template file name for the main Serpent input file was not provided! Please "
275                  "add a value for the parameter 'serpent_input' in the SerpentExecutioner '" +
276                  name() + "'.");
277     pars.set<std::string>("serpent_input") = _serpent_input_template_file_name;
278
279     /*
280      * Add in the parallel processing options for the Serpent executable
281      */
282 #ifdef SERPENT_OPENMP_AVAILABLE
283     if (!_pars.isParamSetByAddParam("serpent_omp_threads"))
284       pars.set<int>("serpent_omp_threads", getParam<int>("serpent_omp_threads"));
285 #endif // SERPENT_OPENMP_AVAILABLE
286
287     /*
288      * Create the TimeStepper as in Transient::init()
```

```
289     */
290     _time_stepper =
291         _app.getFactory().create<TimeStepper>("SerpentTimeStepper", "TimeStepper", pars);
292   }
293   else if (!dynamic_cast<SerpentTimeStepper *>(_time_stepper.get()))
294     mooseError("In SerpentExecutioner '" + name() +
295                "' an explicitly defined TimeStepper must be 'SerpentTimeStepper'");
296   else
297   {
298     SerpentTimeStepper & serpent_stepper = *dynamic_cast<SerpentTimeStepper *>(
  _time_stepper.get());
299
300     if (serpent_stepper.getSerpentInputTemplateFileName().empty())
301       serpent_stepper.setInputFileName(_serpent_input_template_file_name);
302   }
303
304   // Perform the rest of the default init() actions
305   FXExecutioner::init();
306 }
307
308 void
309 SerpentExecutioner::exportCoefficients(const FixedSizeArray<Real> & out_coefficients)
310 {
311   /*
312    * Currently this implementation only supports either temperature or density, not both. This
313    * routine will need changed later if simultaneous exports are supported.
314    */
315   std::string file_name;
316   std::ostringstream formatter;
317   std::ifstream reader;
318   std::ofstream writer;
319
320   const std::string file_base = _request_fission_power_in_density_file
321                                     ? _serpent_interface_density_file_name
322                                     : _serpent_interface_temperature_file_name;
323
324   // Generate the interface file name and then open it for writing
325   const std::string interface_base = file_base + ".ifc";
326
327   generateSerpentInterfaceFile(interface_base, out_coefficients);
328
329   // Create a copy of the interface file if requested
330   if (_keep_files)
331   {
332     reader.open(interface_base);
333
334     if (!reader.is_open())
335     {
336       perror(("Error opening the interface file " + interface_base + "' for reading").
  c_str());
337       mooseError();
338     }
339     else
340     {
341       const std::string duplicate_base = file_base + ".ifc" +
  getTrackingFileNameComponent();
342
343       writer.open(duplicate_base, std::ios::trunc);
344       if (!writer.is_open())
345       {
346         perror(("Error opening the duplicate interface file " + duplicate_base + "' for
  writing")
347                    .c_str());
348         mooseError();
349       }
350
```

```
351       // Duplicate the interface file
352       writer << reader.rdbuf();
353
354       writer.close();
355     }
356
357     reader.close();
358   }
359 }
360
361 void
362 SerpentExecutioner::generateSerpentInputFile(const std::string & interface_base) const
363 {
364   std::ifstream reader;
365   std::ofstream writer;
366
367   // Open the template main input file
368   reader.open(_serpent_input_template_file_name);
369
370   if (!reader.is_open())
371   {
372     perror(
373         ("Error opening the main Serpent template file " + _serpent_input_template_file_name + "'")
374             .c_str());
375     mooseError();
376   }
377   else
378   {
379     // Open the main Serpent input file for writing
380     const std::string main_file_name =
381         SerpentTimeStepper::makeInputFileNameFromTemplate(_serpent_input_template_file_name);
382     writer.open(main_file_name, std::ios::trunc);
383
384     // Write the main input file
385     if (!writer.is_open())
386     {
387       perror(("Error opening the main Serpent input file " + main_file_name + "'").c_str());
388       mooseError();
389     }
390     else
391     {
392       // Copy the body of the template input file
393       writer << reader.rdbuf();
394
395       // Add in the coupling commands
396       writer << "\n\n%% ==== START ==== Autogenerated multiphysics section ==== START ====\n";
397       writer << "%%\tMultiphysics interface file\n";
398       writer << "ifc " << interface_base << "\n";
399       writer << "\n%%\tMultiphysics signaling\n";
400       writer << "set ppid 666\n";
401       writer << "set ccmaxiter 999\n";
402       writer << "set power 1.0\n"; // We handle power ourselves, so do not normalize
403       writer << "\n%% ====  END  ==== Autogenerated multiphysics section ====  END  ====\n";
404     }
405
406     // Close the main input file
407     writer.close();
408   }
409
410   // Close the template input file
411   reader.close();
```

```
412 }
413
414 void
415 SerpentExecutioner::generateSerpentInterfaceFile(
416     const std::string & interface_base, const FixedSizeArray<Real> & out_coefficients)
      ↳const
417 {
418   std::ofstream writer;
419
420   /*
421    * Write the basic Serpent parameters
422    */
423   writer.open(interface_base, std::ios::trunc);
424   if (!writer.is_open())
425   {
426     perror(("Error opening the FE export file '" + interface_base + "'").c_str());
427     mooseError();
428   }
429   else
430   {
431     // The interface type
432     writer << (_request_fission_power_in_density_file ? IFC_TYPE_FET_DENSITY :
      ↳IFC_TYPE_FET_TEMP);
433     // The FE type
434     if (_series_type_name == "Cartesian")
435       writer << " " << FET_TYPE_CARTESIAN;
436     else
437       writer << " " << FET_TYPE_CYLINDRICAL;
438     // The feedback material
439     writer << " " << _serpent_otf_material;
440     /* Generate an fission power density output in the pin identified by the outermost material */
441     writer << " " << YES;
442     writer << " " << _serpent_fission_power_outermost_material;
443     writer << "\n";
444
445     /*
446     * Write the scoring parameters
447     */
448     // The fission power density FET file name
449     writer << getFissionPowerDensityFileName(false);
450     // Duplicate the same FE parameters for both the input and the output?
451     writer << " " << YES;
452     // Write out the geometry-specific FE parameters
453     // clang-format off
454     if (_series_type_name == "Cartesian")
455     {
456       //             min x                       max x                       Legendre order
457       writer << " " << _physical_bounds[0] << " " << _physical_bounds[1] << " " << _orders[0]
458              //       min y                       max y                       Legendre order
459              << " " << _physical_bounds[2] << " " << _physical_bounds[3] << " " << _orders[1]
460              //       min z                       max z                       Legendre order
461              << " " << _physical_bounds[4] << " " << _physical_bounds[5] << " " << _orders[2];
462     }
463     else // CylindricalDuo
464     {
465       //             outer radius                Zernike order
466       writer << " " << _physical_bounds[4] << " " << _orders[1]
467              //       axial bottom                axial top                   Legendre order
468              << " " << _physical_bounds[0] << " " << _physical_bounds[1] << " " << _orders[0]
469              //       axial orientation
470              << " " << (_x.isValid() ? FET_ORIENTATION_X
471                                      : (_y.isValid() ? FET_ORIENTATION_Y : FET_ORIENTATION_Z))
```

```
472               //       axial center 1              axial center 2
473               << " " << _physical_bounds[2] << " " << _physical_bounds[3];
474     }
475     // clang-format on
476     writer << "\n";
477
478     // Check to see if all the coefficients are zero; if so, this is bad
479     bool all_zero = true;
480     for (std::size_t i = 0; i < out_coefficients.size(); ++i)
481       if (out_coefficients[i] != 0.0)
482         all_zero = false;
483
484     // Write out the exported FE coefficients, substituting the first if necessary
485     bool first = all_zero;
486     for (std::size_t i = 0; i < out_coefficients.size(); ++i)
487       writer << (first ? first = false, _serpent_initial_constant_fe_value :
  ↳out_coefficients[i])
488              << "\n";
489   }
490
491   // Close the export file
492   writer.close();
493 }
494
495 std::string
496 SerpentExecutioner::getFissionPowerDensityFileName(bool generate_history_filename) const
497 {
498   std::string name = _serpent_interface_fission_power_density_file_name;
499
500   name.append(SerpentTimeStepper::makeMpiUnique() + ".pwr");
501
502   if (generate_history_filename)
503     name += getTrackingFileNameComponent();
504
505   return name;
506 }
507
508 const SerpentTimeStepper &
509 SerpentExecutioner::getMySerpentTimeStepper() const
510 {
511   return *(dynamic_cast<SerpentTimeStepper *>(_time_stepper.get()));
512 }
513
514 Real
515 SerpentExecutioner::getNormalization(const std::vector<Real> & _coefficients) const
516 {
517   Real multiplier;
518
519   // Calculate the multiplier
520   if (_is_power_level_time_varying)
521     multiplier = _function_power_level->value(_time, Point(0));
522   else
523     multiplier = _const_power_level;
524
525   // Adjust the multiplier if we are generating an average value based on the value of
526   // the zeroth-order coefficient
527   if (_is_power_level_an_average)
528     multiplier /= _coefficients[0];
529
530 #ifndef NDEBUG
531   _console << COLOR_BLUE << "\nIt was calculated that multiplying the fission power
  ↳density FET by "
532            << std::setprecision(6) << multiplier
533            << " will result in a power level with an average value of " << std::
  ↳setprecision(6)
534            << multiplier * _coefficients[0] << "\n"
```

```
535             << COLOR_DEFAULT << std::endl;
536 #endif // NDEBUG
537
538   return multiplier;
539 }
540
541 std::string
542 SerpentExecutioner::getTrackingFileNameComponent() const
543 {
544   std::ostringstream formatter("");
545
546   formatter << "_S" << getLocalTimeStep() << "_P" << getLocalPicardIteration();
547
548   return formatter.str();
549 }
550
551 void
552 SerpentExecutioner::importCoefficients(FixedSizeArray<Real> & array_to_fill)
553 {
554   // Get the FET coefficients from Serpent here
555   const std::string coefficient_file = getFissionPowerDensityFileName(false);
556   std::ifstream reader;
557
558   reader.open(coefficient_file);
559   if (!reader.is_open())
560   {
561     perror(
562         ("Error opening the fission power density output file '" + coefficient_file + "'"
    ↳).c_str());
563     mooseError();
564   }
565   else
566   {
567     long region, first_region = -1;
568     long linear_index;
569     long order_1, order_2, order_3;
570     Real coefficient, relative_uncertainty;
571     std::string line;
572
573     if (getline(reader, line))       // DISCARD: header row
574       if (getline(reader, line))     // DISCARD: parameters row
575         if (getline(reader, line))   // DISCARD: empty line
576           if (getline(reader, line)) // DISCARD: coefficient table headers
577             while (reader)
578             {
579               reader >> region >> linear_index >> order_1 >> order_2 >> order_3 >>
    ↳coefficient >>
580                   relative_uncertainty;
581
582               // Get out of Dodge if something bad has happened
583               if (reader.bad() || reader.eof())
584                 break;
585
586               // This is the first data line read, so set up any required parameters
587               if (first_region < 0)
588               {
589                 // Capture the ID of the first found region
590                 first_region = region;
591
592                 // Ensure the FE data is sane
593                 if (linear_index != 0)
594                   mooseError("The first found coefficient in '%s' is not a zeroth-order "
595                              "coefficient. This is very bad so we will exit now.");
596               }
597               else if (region != first_region)
598                 break; // Only one region is currently supported, so exit if another region is found
```

```
599
600               array_to_fill[linear_index] = relative_uncertainty < 1 ? coefficient : 0.0;
601             }
602
603     if (reader.bad())
604     {
605       perror(("Error reading from the fission power density output file '" +
  ↳coefficient_file + "'")
606                  .c_str());
607
608       if (array_to_fill.size() == 0)
609         mooseError("No coefficients were read from '%s'", coefficient_file);
610       else
611         mooseWarning("Some coefficients may not have been read from '%s'."
612                      "\nAttempting to continue...",
613                      coefficient_file);
614     }
615   }
616
617   reader.close();
618
619   // Create a copy of the FET power file if requested
620   if (_keep_files)
621   {
622     reader.open(coefficient_file);
623
624     if (!reader.is_open())
625     {
626       perror(("Error opening the fission power density output file '" + coefficient_file
  ↳+ "'")
627                  .c_str());
628       mooseError();
629     }
630     else
631     {
632       std::ofstream writer;
633       const std::string duplicate_base = getFissionPowerDensityFileName(true);
634
635       writer.open(duplicate_base, std::ios::trunc);
636       if (!writer.is_open())
637       {
638         perror(("Error opening the duplicate fission power density output file " +
  ↳duplicate_base +
639                 "' for writing")
640                    .c_str());
641         mooseError();
642       }
643
644       // Duplicate the interface file
645       writer << reader.rdbuf();
646
647       writer.close();
648     }
649
650     reader.close();
651   }
652 }
653
654 std::size_t
655 SerpentExecutioner::mapIndexingMooseToExternal(const std::size_t mooose_fe_index) const
656 {
657   // The zeroth coefficients are always the same
658   if (mooose_fe_index == 0)
659     return 0;
660
661   if (_series_type_name == "Cartesian")
```

```
662     return mooose_fe_index; // The Cartesian indexing schema are the same for Serpent and MOOSE
663   else                      // CylindricalDuo
664   {
665     const std::size_t legendre_terms = _orders[0] + 1;
666     const std::size_t zernike_terms = (_orders[1] + 1) * (_orders[1] + 2) / 2;
667     const std::size_t zernike_index = mooose_fe_index % (zernike_terms);
668     const std::size_t legendre_index = (mooose_fe_index - zernike_index) / zernike_terms;
669
670     mooseAssert(zernike_index <= zernike_terms,
671                 "Something bad has happened when mapping the MOOSE FE coefficient "
672                     << mooose_fe_index << " to the Serpent indexing schema. A Zernike
    ↳index of "
673                     << zernike_index << " was calculated, but the highest possible index
    ↳is "
674                     << zernike_terms);
675     mooseAssert(legendre_index <= legendre_terms,
676                 "Something bad has happened when mapping the MOOSE FE coefficient "
677                     << mooose_fe_index << " to the Serpent indexing schema. A Legendre
    ↳index of "
678                     << legendre_index << " was calculated, but the highest possible index
    ↳ is "
679                     << legendre_terms);
680
681     return zernike_index * legendre_terms + legendre_index;
682   }
683 }
```

E.3 InterfaceKernels

File E.9 – An interface kernel that provides a matching flux condition at a shared interface between to libMesh blocks in a single MOOSE MultiApp. InterfaceDiffusion.h

```
/****************************************************************/
/*               DO NOT MODIFY THIS HEADER                      */
/* MOOSE - Multiphysics Object Oriented Simulation Environment  */
/*                                                              */
/*           (c) 2010 Battelle Energy Alliance, LLC             */
/*                   ALL RIGHTS RESERVED                        */
/*                                                              */
/*          Prepared by Battelle Energy Alliance, LLC           */
/*            Under Contract No. DE-AC07-05ID14517              */
/*            With the U. S. Department of Energy               */
/*                                                              */
/*            See COPYRIGHT for full restrictions               */
/****************************************************************/

#ifndef INTERFACEDIFFUSION_H
#define INTERFACEDIFFUSION_H

#include "InterfaceKernel.h"

// Forward Declarations
class InterfaceDiffusion;

template <>
InputParameters validParams<InterfaceDiffusion>();

/**
 * DG kernel for interfacing diffusion between two variables on adjacent blocks
 */
class InterfaceDiffusion : public InterfaceKernel
{
public:
  InterfaceDiffusion(const InputParameters & parameters);

protected:
  virtual Real computeQpResidual(Moose::DGResidualType type);
  virtual Real computeQpJacobian(Moose::DGJacobianType type);

  const bool _use_variable_side_material;
  const bool _use_neighbor_side_material;
  const std::string _variable_diffusivity_input;
  const std::string _neighbor_diffusivity_input;
  const MaterialProperty<Real> * const _variable_diffusion_coefficient_property;
  const MaterialProperty<Real> * const _neighbor_diffusion_coefficient_property;
  Real _variable_diffusion_coefficient;
  Real _neighbor_diffusion_coefficient;
};

#endif
```

File E.10 – An interface kernel that provides a matching flux condition at a shared interface between to libMesh blocks in a single MOOSE MultiApp. InterfaceDiffusion.C

```
#include "InterfaceDiffusion.h"

registerMooseObject("ChrysalisApp", InterfaceDiffusion);

```

```
#include <cmath>

template <>
InputParameters
validParams<InterfaceDiffusion>()
{
  InputParameters params = validParams<InterfaceKernel>();

  params.addParam<bool>("use_variable_side_material",
                        true,
                        "Use a material to get the diffusion coefficient on the variable side.");
  params.addRequiredParam<std::string>(
      "variable_side_diffusivity",
      "The diffusivity property on the variable side that will be used in the "
      "flux computation. This must be the name of a material property if "
      "'use_variable_side_material' is true (default), otherwise this must be "
      "a numeric value.");

  params.addParam<bool>("use_neighbor_side_material",
                        true,
                        "Use a material to get the diffusion coefficient on the neighbor side.");
  params.addParam<std::string>(
      "neighbor_side_diffusivity",
      "The diffusivity property on the neighbor side that will be used in the "
      "flux computation. This should be the name of a material property if "
      "'use_neighbor_side_material' is true (default), otherwise this should "
      "be a numeric value. If left blank then it will default to the parameter "
      "provided for 'variable_side_diffusivity'.");

  return params;
}

InterfaceDiffusion::InterfaceDiffusion(const InputParameters & parameters)
  : InterfaceKernel(parameters),
    _use_variable_side_material(getParam<bool>("use_variable_side_material")),
    _use_neighbor_side_material(getParam<bool>("use_neighbor_side_material")),
    _variable_diffusivity_input(getParam<std::string>("variable_side_diffusivity")),
    _neighbor_diffusivity_input(isParamValid("neighbor_side_diffusivity")
                                    ? getParam<std::string>("neighbor_side_diffusivity")
                                    : (_use_variable_side_material == _use_neighbor_side_material
                                           ? getParam<std::string>("variable_side_diffusivity")
                                           : "")),
    _variable_diffusion_coefficient_property(
        _use_variable_side_material ? &getMaterialProperty<Real>(_variable_diffusivity_input)
                                    : NULL),
    _neighbor_diffusion_coefficient_property(
        _use_neighbor_side_material
            ? &getNeighborMaterialProperty<Real>(_neighbor_diffusivity_input)
            : NULL),
    _variable_diffusion_coefficient(
        _use_variable_side_material ? 0.0 : atof(&_variable_diffusivity_input.front())),
    _neighbor_diffusion_coefficient(
        _use_neighbor_side_material ? 0.0 : atof(&_neighbor_diffusivity_input.front()))
{
  if (!parameters.isParamValid("boundary"))
    mooseError("In order to use the InterfaceDiffusion '",
               name(),
               "', you must specify a boundary where it will live.");

  if (!isParamValid("neighbor_side_diffusivity") &&
      _use_variable_side_material != _use_neighbor_side_material)
```

```
66      mooseError("InterfaceKernel '",
67                 name(),
68                 "': cannot reuse the value from 'variable_side_diffusivity' in "
69                 "'neighbor_side_diffusivity'.\nThis is because the parameters "
70                 "'use_variable_side_material' and 'use_neighbor_side_material' "
71                 "are not identical."
72                 "\nPossible fixes:"
73                 "\n\t1) create an entry for 'neighbor_side_diffusivity' and supply "
74                 "a diffusivity constant manually"
75                 "\n\t2) change 'use_neighbor_side_material' to match "
76                 "'use_variable_side_material'");
77  }
78
79  Real
80  InterfaceDiffusion::computeQpResidual(Moose::DGResidualType type)
81  {
82    if (_use_variable_side_material)
83      _variable_diffusion_coefficient = (*_variable_diffusion_coefficient_property)[_qp];
84    if (_use_neighbor_side_material)
85      _neighbor_diffusion_coefficient = (*_neighbor_diffusion_coefficient_property)[_qp];
86
87    Real r = 0.5 * (-_variable_diffusion_coefficient * _grad_u[_qp] * _normals[_qp] +
88                    -_neighbor_diffusion_coefficient * _grad_neighbor_value[_qp] * _normals
    ↳[_qp]);
89
90    switch (type)
91    {
92      case Moose::Element:
93        r *= _test[_i][_qp];
94        break;
95
96      case Moose::Neighbor:
97        r *= -_test_neighbor[_i][_qp];
98        break;
99    }
100
101   return r;
102 }
103
104 Real
105 InterfaceDiffusion::computeQpJacobian(Moose::DGJacobianType type)
106 {
107   Real jac = 0;
108
109   if (_use_variable_side_material)
110     _variable_diffusion_coefficient = (*_variable_diffusion_coefficient_property)[_qp];
111   if (_use_neighbor_side_material)
112     _neighbor_diffusion_coefficient = (*_neighbor_diffusion_coefficient_property)[_qp];
113
114   switch (type)
115   {
116
117     case Moose::ElementElement:
118       jac -= 0.5 * _variable_diffusion_coefficient * _grad_phi[_j][_qp] * _normals[_qp] *
119              _test[_i][_qp];
120       break;
121
122     case Moose::NeighborNeighbor:
123       jac += 0.5 * _neighbor_diffusion_coefficient * _grad_phi_neighbor[_j][_qp] *
    ↳_normals[_qp] *
124              _test_neighbor[_i][_qp];
125       break;
126
127     case Moose::NeighborElement:
128       jac += 0.5 * _variable_diffusion_coefficient * _grad_phi[_j][_qp] * _normals[_qp] *
129              _test_neighbor[_i][_qp];
```

```
130       break;
131
132     case Moose::ElementNeighbor:
133       jac -= 0.5 * _neighbor_diffusion_coefficient * _grad_phi_neighbor[_j][_qp] * ↲
  ↳_normals[_qp] *
134               _test[_i][_qp];
135       break;
136   }
137
138   return jac;
139 }
```

E.4 Kernels

File E.11 – A Kernel that provides temperature-dependent heat generation in a cylindrical region, similar to a fuel pin in a nuclear reactor. FuelPinHeat.h

```h
#ifndef FUELPINHEAT_H
#define FUELPINHEAT_H

#include "Kernel.h"

// Forward Declarations
class FuelPinHeat;

template <>
InputParameters validParams<FuelPinHeat>();

/**
 * This Kernel implements a heating term that generalizes the behavior of the
 * TREAT reactor during a transient event in a single core element
 *
 * The axial direction of the fuel element is assumed to be in 'x'
 */
class FuelPinHeat : public Kernel
{
public:
  FuelPinHeat(const InputParameters & parameters);

  const Real _element_length;
  const Real _element_length_half;
  const Real _element_widths;
  const Real _element_widths_half;
  const Real _total_volume;
  const Real _zero_power_radius;

protected:
  virtual Real computeQpResidual() override;

  const Point _center;
  const Real _total_energy;
  const Real _max_heating_rate;
  const Real _max_temperature;
  const Real _initial_temperature;
};

#endif // FUELPINHEAT_H
```

File E.12 – A Kernel that provides temperature-dependent heat generation in a cylindrical region, similar to a fuel pin in a nuclear reactor. FuelPinHeat.C

```C
#include "FuelPinHeat.h"

#include "math.h"

registerMooseObject("ChrysalisApp", FuelPinHeat);

template <>
InputParameters
validParams<FuelPinHeat>()
{
  InputParameters params = validParams<Kernel>();

  params.addRequiredParam<Point>("center", "The center of the fuel pin");
  params.addRangeCheckedParam<Real>(
```

```
      "total_energy",
      "total_energy > 1",
      "The total energy that can be generated during a transient event");
  params.addRangeCheckedParam<Real>("max_temperature",
                                    "max_temperature <= 925",
                                    "The maximum design temperature of the transient");
  params.addRangeCheckedParam<Real>(
      "initial_temperature",
      "initial_temperature >= 200",
      "The initial temperature (at which heat generation is maximum)");

  return params;
}

FuelPinHeat::FuelPinHeat(const InputParameters & parameters)
  : Kernel(parameters),
    _element_length(5),
  // cm
    _element_length_half(_element_length / 2.0),
  // cm
    _element_widths(2),
  // cm
    _element_widths_half(_element_widths / 2.0),
  // cm
    _total_volume(M_PI * _element_widths_half * _element_widths_half * _element_length),
  // cm^3
    _zero_power_radius(1.5 * _element_widths),
    _center(getParam<Point>("center")),
    _total_energy(getParam<Real>("total_energy")),
    _max_heating_rate(_total_energy / _total_volume),
    _max_temperature(getParam<Real>("max_temperature")),
    _initial_temperature(getParam<Real>("initial_temperature"))
{
}

Real
FuelPinHeat::computeQpResidual()
{
  const Point & point = _q_point[_qp];
  const Real x = point(0) - _center(0);
  const Real y = point(1) - _center(1);
  const Real z = abs(point(2) - _center(2));

  const Real axial_factor = cos(M_PI * pow(z / _element_length_half, 4) / 3);
  const Real radial_factor = 1 - sqrt(x * x + y * y) / _zero_power_radius;
  const Real neutronic_heating =
      exp(-2.30258509299 * (_u[_qp] - _initial_temperature) /
          (_max_temperature - _initial_temperature)); // 10% heat generation at _max_temperature

  return -_test[_i][_qp] * _max_heating_rate * neutronic_heating * axial_factor *
  radial_factor;
}
```

File E.13 – A Kernel that provides temperature-dependent heat generation in a Cartesian region, similar to a fuel element in TREAT. TREATHeat.h 

```
#ifndef TREATHEAT_H
#define TREATHEAT_H

#include "Kernel.h"

// Forward Declarations
class TREATHeat;

```

```
template <>
InputParameters validParams<TREATHeat>();

/**
 * This Kernel implements a heating term that generalizes the behavior of the
 * TREAT reactor during a transient event in a single core element
 *
 * The axial direction of the fuel element is assumed to be in 'x'
 */
class TREATHeat : public Kernel
{
public:
  TREATHeat(const InputParameters & parameters);

  const Real _element_length;
  const Real _element_length_half;
  const Real _element_widths;
  const Real _element_widths_half;
  const unsigned int _number_of_elements;
  const Real _total_volume;
  const Real _zero_power_radius;

protected:
  virtual Real computeQpResidual() override;

  const Point _center;
  const Real _total_energy;
  const Real _transient_duration;
  const Real _front_edge;
  const Real _max_heating_rate;
  const Real _max_temperature;
  const Real _initial_temperature;
};

#endif // TREATHEAT_H
```

File E.14 – A Kernel that provides temperature-dependent heat generation in a Cartesian region, similar to a fuel element in TREAT. TREATHeat.C

```
#include "TREATHeat.h"

#include "math.h"

registerMooseObject("ChrysalisApp", TREATHeat);

template <>
InputParameters
validParams<TREATHeat>()
{
  InputParameters params = validParams<Kernel>();

  params.addRequiredParam<Point>("center", "The center of the fuel element");
  params.addRangeCheckedParam<Real>(
      "total_energy",
      "total_energy > 1",
      "The total energy that can be generated during a transient event");
  params.addRangeCheckedParam<Real>(
      "transient_duration", "transient_duration > 0.25", "The duration of the transient
  ↳event");
  params.addRangeCheckedParam<Real>("max_temperature",
                                    "max_temperature <= 925",
                                    "The maximum design temperature of the transient");
  params.addRangeCheckedParam<Real>(
      "initial_temperature",
```

```
      "initial_temperature >= 200",
      "The initial temperature (at which heat generation is maximum)");

  return params;
}

TREATHeat::TREATHeat(const InputParameters & parameters)
  : Kernel(parameters),
    _element_length(120),                        // cm
    _element_length_half(_element_length / 2.0), // cm
    _element_widths(10),                         // cm
    _element_widths_half(_element_widths / 2.0), // cm
    _number_of_elements(19 * 19),
    _total_volume(_number_of_elements * _element_widths * _element_widths *
                  _element_length), // cm^3
    _zero_power_radius(5 * sqrt(_element_widths * _element_widths)),
    _center(getParam<Point>("center")),
    _total_energy(getParam<Real>("total_energy")),
    _transient_duration(getParam<Real>("transient_duration")),
    _front_edge(_transient_duration / 3.0),
    _max_heating_rate(_total_energy / (_total_volume * _transient_duration)),
    _max_temperature(getParam<Real>("max_temperature")),
    _initial_temperature(getParam<Real>("initial_temperature"))
{
}

Real
TREATHeat::computeQpResidual()
{
  const Point & point = _q_point[_qp];
  const Real x = point(0) - _center(0);
  const Real y = point(1) - _center(1) + _element_widths_half;
  const Real z = point(2) - _center(2) + _element_widths_half;

  const Real axial_factor = cos(M_PI * pow(x / _element_length_half, 4) / 3);
  const Real radial_factor = 1 - sqrt(y * y + z * z) / _zero_power_radius;
  const Real neutronic_heating =
      exp(-2.30258509299 * (_u[_qp] - _initial_temperature) /
          (_max_temperature - _initial_temperature)); // 10% heat generation at _max_temperature
  const Real ramping =
      (_t <= _front_edge) ? (_t / _front_edge) : ((_t < _transient_duration) ? 1.0 : 1e-4);

  return -_test[_i][_qp] * _max_heating_rate * ramping * neutronic_heating * axial_factor *
         radial_factor;
}
```

E.5 TimeSteppers

File E.15 – The TimeStepper for interacting with Serpent by calling the execution methods and setting internal data flags. SerpentTimeStepper.h

```h
#ifndef SERPENTTIMESTEPPER_H
#define SERPENTTIMESTEPPER_H

#include "TimeStepper.h"

#include "header.h"
#include "locations.h"

class SerpentTimeStepper;

template <>
InputParameters validParams<SerpentTimeStepper>();

/**
 * This class is responsible for communicating with Serpent and starting each iteration's solve
 */
class SerpentTimeStepper : public TimeStepper
{
public:
  SerpentTimeStepper(const InputParameters & parameters);
  virtual ~SerpentTimeStepper();

  /**
   * Gets the name of the main input file for Serpent
   */
  const std::string & getSerpentInputFileName() const;

  /**
   * Gets the name of the main input file for Serpent
   */
  const std::string & getSerpentInputTemplateFileName() const;

  /**
   * Generate a string that is unique for each instance based on any threading and MPI ranks
   */
  static std::string makeMpiUnique();

  /**
   * Generate the name of the file that is to be used as the main input file
   */
  static std::string makeInputFileNameFromTemplate(const std::string & template_name);

  /**
   * Set the input file name if required
   */
  void setInputFileName(const std::string & template_name);

protected:
  /* Overrides from TimeStepper */
  virtual Real computeDT() override;
  virtual Real computeInitialDT() override;
  virtual bool converged() override;
  virtual void init() override;
  virtual void step() override;

protected:
  /// Name of the main Serpent input file name
  std::string _serpent_input_template_file_name;

```

```
60    /// Name of the main Serpent input file name
61    std::string _serpent_input_file_name;
62
63    /// The number of OpenMP threads with which to run Serpent
64    const int _serpent_omp_threads;
65
66  private:
67    /**
68     * Helper to generate the command input arguments for Serpent
69     */
70    void
71    appendCommand(int & argv, std::vector<std::string> & arg_strings, const std::string &
      ↳new_arg);
72
73    /**
74     * The parts of IterateCC() and IterateExternal() that come after SignalExternal()
75     */
76    void serpentPostIterate();
77
78    /**
79     * The parts of IterateCC() and IterateExternal() that come before SignalExternal()
80     */
81    void serpentPreIterate();
82
83    /**
84     * Initializes Serpent
85     */
86    int serpentInit(int argc, char ** argv);
87
88    // Flag for first Serpent step
89    bool _first_serpent_step;
90
91    /// The name of the main Serpent input file
92    std::string _main_input_file_name;
93
94    /// The name of the file in which any POSIX signals from Serpent will be found
95    std::string _signal_input_file_name;
96
97    /// The name of the file to which any POSIX signals to Serpent will be written
98    std::string _signal_output_file_name;
99
100 ////////////////////////////////////////////////////////////////////////////////
101 //   4 lines redacted to maintain compliance with RSICC export control license //
102 ////////////////////////////////////////////////////////////////////////////////
103 };
104
105 #endif // SERPENTTIMESTEPPER_H
```

File E.16 – The TimeStepper for interacting with Serpent by calling the execution methods and setting internal data flags. SerpentTimeStepper.C

```
1   #include <atomic>
2
3   #include "Transient.h"
4
5   #include "SerpentExecutioner.h"
6   #include "SerpentTimeStepper.h"
7
8   registerMooseObject("ChrysalisApp", SerpentTimeStepper);
9
10  template <>
11  InputParameters
12  validParams<SerpentTimeStepper>()
13  {
```

```
14   InputParameters params = validParams<TimeStepper>();
15
16   params.addClassDescription("TimeStepper for running the Serpent Reactor Physics MC code "
17                              "developed at VTT, Finland");
18   params.set<std::string>("_object_name", "SerpentTimeStepper");
19
20   /*
21    * Files for interfacing
22    */
23   params.addParam<std::string>(
24       "serpent_input",
25       "",
26       "Name of the main Serpent input file to be used as a template for creating the multiphysics "
27       "interface. Namely, the communication settings and the \"ifc ...\" lines will be appended to "
28       "a duplicate file using the name \"'serpent_input'.moose\".");
29   params.addParamNamesToGroup("serpent_input", "Interface Files");
30
31   /*
32    * Parallel processing options
33    */
34 #ifdef SERPENT_OPENMP_AVAILABLE
35   params.addParam<int>("serpent_omp_threads",
36                        -1,
37                        "The number of OpenMP threads with which to run Serpent. A value of '-1' "
38                        "will default to the number of threads used for this MultiApp.");
39   params.addParamNamesToGroup("serpent_omp_threads", "Serpent Execution Options");
40 #endif // SERPENT_OPENMP_AVAILABLE
41
42   return params;
43 }
44
45 SerpentTimeStepper::SerpentTimeStepper(const InputParameters & parameters)
46   : TimeStepper(parameters),
47     _serpent_input_template_file_name(getParam<std::string>("serpent_input")),
48     _serpent_input_file_name(makeInputFileNameFromTemplate(_serpent_input_template_file_name)),
49 #ifdef SERPENT_OPENMP_AVAILABLE
50     _serpent_omp_threads(getParam<int>("serpent_omp_threads") > 0
51                          ? getParam<int>("serpent_omp_threads")
52                          : libMesh::n_threads()),
53 #else
54     _serpent_omp_threads(1),
55 #endif // SERPENT_OPENMP_AVAILABLE
56     _first_serpent_step(true)
57 {
58 }
59
60 void
61 SerpentTimeStepper::appendCommand(int & argv,
62                                   std::vector<std::string> & arg_strings,
63                                   const std::string & new_arg)
64 {
65   arg_strings.push_back(new_arg);
66   ++argv;
67 }
68
69 Real
70 SerpentTimeStepper::computeDT()
71 {
72   return 1.0;
73 }
74
```

```
75  Real
76  SerpentTimeStepper::computeInitialDT()
77  {
78    return 1.0;
79  }
80
81  bool
82  SerpentTimeStepper::converged()
83  {
84    /* Assume that Serpent has always converged so that the simulation will continue */
85    return true;
86  }
87
88  void
89  SerpentTimeStepper::init()
90  {
91    int argv = 0;
92    std::vector<char *> argc;
93    std::vector<std::string> arg_strings;
94    std::ostringstream formatter;
95
96    appendCommand(argv, arg_strings, "chrysalis");
97    appendCommand(argv, arg_strings, _serpent_input_file_name);
98
99  #ifdef SERPENT_OPENMP_AVAILABLE
100   if (_serpent_omp_threads > 1)
101   {
102     appendCommand(argv, arg_strings, "-omp");
103
104     formatter.str("");
105     formatter.clear();
106     formatter << _serpent_omp_threads;
107     appendCommand(argv, arg_strings, formatter.str());
108   }
109 #endif // SERPENT_OPENMP_AVAILABLE
110
111   for (auto & string : arg_strings)
112     argc.push_back(&string[0]);
113   serpentInit(argv, &argc[0]);
114
115   SerpentExecutioner * const check = dynamic_cast<SerpentExecutioner *>(&_executioner);
116   if (!check)
117     mooseError("The Executioner using this SerpentTimeStepper is not a SerpentExecutioner
      ↳. It must "
118                "be, so please change it.");
119 }
120
121 const std::string &
122 SerpentTimeStepper::getSerpentInputFileName() const
123 {
124   return _serpent_input_file_name;
125 }
126
127 const std::string &
128 SerpentTimeStepper::getSerpentInputTemplateFileName() const
129 {
130   return _serpent_input_template_file_name;
131 }
132
133 std::string
134 SerpentTimeStepper::makeInputFileNameFromTemplate(const std::string & template_name)
135 {
136   return template_name + ".moose";
137 }
138
139 std::string
```

```
140 SerpentTimeStepper::makeMpiUnique()
141 {
142   int rank;
143   MPI_Comm_rank(MPI_COMM_WORLD, &rank);
144 
145   std::ostringstream formatter("");
146   formatter << ".m" << rank;
147 
148   return formatter.str();
149 }
150 
151 int
152 SerpentTimeStepper::serpentInit(int argc, char ** argv)
153 {
154   _console << COLOR_YELLOW << "Starting Serpent initialization..." << COLOR_DEFAULT <<
    std::endl;
155 
156 ////////////////////////////////////////////////////////////////////////////////
157 // 504 lines redacted to maintain compliance with RSICC export control license //
158 ////////////////////////////////////////////////////////////////////////////////
159 
160   _console << COLOR_YELLOW << "Serpent initialization complete!" << COLOR_DEFAULT << std
    ::endl;
161 
162   return 0;
163 }
164 
165 void
166 SerpentTimeStepper::serpentPostIterate()
167 {
168 ////////////////////////////////////////////////////////////////////////////////
169 //  47 lines redacted to maintain compliance with RSICC export control license //
170 ////////////////////////////////////////////////////////////////////////////////
171 }
172 
173 void
174 SerpentTimeStepper::serpentPreIterate()
175 {
176 ////////////////////////////////////////////////////////////////////////////////
177 //  79 lines redacted to maintain compliance with RSICC export control license //
178 ////////////////////////////////////////////////////////////////////////////////
179 }
180 
181 void
182 SerpentTimeStepper::setInputFileName(const std::string & template_name)
183 {
184   if (_serpent_input_template_file_name.empty())
185   {
186     _serpent_input_template_file_name = template_name;
187     _serpent_input_file_name = makeInputFileNameFromTemplate(template_name);
188   }
189   else
190     mooseError("SerpentTimeStepper: the template input file name has already been set.");
191 }
192 
193 void
194 SerpentTimeStepper::step()
195 {
196   _console << COLOR_YELLOW << "Starting a Serpent run..." << COLOR_DEFAULT << std::endl;
197 
198   if (_first_serpent_step)
199     _first_serpent_step = false;
200   else
201     serpentPreIterate();
202 
203 ////////////////////////////////////////////////////////////////////////////////
```

```
//   8 lines redacted to maintain compliance with RSICC export control license //
////////////////////////////////////////////////////////////////////////////////

  serpentPostIterate();

  _console << COLOR_YELLOW << "Serpent run complete." << COLOR_DEFAULT << std::endl;
}

SerpentTimeStepper::~SerpentTimeStepper()
{
////////////////////////////////////////////////////////////////////////////////
//  15 lines redacted to maintain compliance with RSICC export control license //
////////////////////////////////////////////////////////////////////////////////
}
```

E.6 Build Files

File E.17 – The makefile used to build Serpent as a library to MOOSE. chrysalis.mk 

```
# Build Serpent
################################################################################

# Build options

# Use link-time optimization for improved run speed. Building/linking may take
# longer, but the run-time duration is up to 60% shorter
USE_LTO                 := TRUE

################################################################################

# Ensure a sane build environment, error if otherwise

define MESSAGE_ERROR

================================================================================
SERPENT_HOME not defined, cannot build Serpent module.

Ensure the global module makefile checks for this condition before it triggers
my build.
================================================================================
endef

ifeq ($(SERPENT_HOME),)
    $(error $(MESSAGE_ERROR))
endif

################################################################################

# Run the script to make/update a local copy of the Serpent source code

APPLICATION_DIR         := $(realpath $(APPLICATION_DIR))

LOCAL_SERPENT_DIR       := $(APPLICATION_DIR)/serpent

TRICK_SCRIPT_TO_RUN     := $(shell $(APPLICATION_DIR)/scripts/update_serpent.sh $(
  ↳APPLICATION_DIR)/serpent)

################################################################################

# Define the build environment

NAME                    := serpent
SERPENT_LIB_NAME        := lib$(NAME)-$(METHOD).la
SERPENT_LIB             := $(LOCAL_SERPENT_DIR)/$(SERPENT_LIB_NAME)
SERPENT_SRC             := $(sort $(wildcard $(LOCAL_SERPENT_DIR)/*.c))
SERPENT_OBJ             := $(patsubst %.c, %.$(obj-suffix), $(SERPENT_SRC))
SERPENT_DEPS            := $(patsubst %.$(obj-suffix), %.$(obj-suffix).d, $(SERPENT_OBJ))

SERPENT_CFLAGS          := $(ADDITIONAL_CPPFLAGS) -w -ansi -ffast-math -O3
SERPENT_LDFLAGS         := $(ADDITIONAL_LDFLAGS) -lm

# Add Serpent to this app's dependencies
app_INCLUDES            += -I$(LOCAL_SERPENT_DIR)
app_LIBS                += $(SERPENT_LIB)

# Local source directory
MESSAGE_DIRECTORY       := - The local copy of Serpent is in '$(LOCAL_SERPENT_DIR)'

# MPI
```

```
ifeq ($(findstring mpi,$(libmesh_CC)),)
MESSAGE_MPI         := - Compiling without MPI support
else
MESSAGE_MPI         := - MPI compiler detected! Compiling with MPI support
SERPENT_CFLAGS      += -DMPI
ADDITIONAL_CPPFLAGS += -DSERPENT_MPI_AVAILABLE
endif

# OpenMP
ifeq ($(findstring openmp,$(libmesh_CFLAGS)),)
MESSAGE_OPENMP      := - Compiling for single-thread operations
else
MESSAGE_OPENMP      := - OpenMP libraries found! Compiling with OpenMP support
SERPENT_CFLAGS      += -DOPEN_MP
ADDITIONAL_CPPFLAGS += -DSERPENT_OPENMP_AVAILABLE
endif

# Graphics
ifeq ($(shell ldconfig -p | grep "libgd\."),)
MESSAGE_GRAPHICS    := - Compiling without GD graphics support
SERPENT_CFLAGS      += -DNO_GFX_MODE
else
MESSAGE_GRAPHICS    := - GD graphics library found! Compiling with graphics support
SERPENT_LDFLAGS     += -lgd
endif

# Debug
ifeq ($(METHOD),dbg)
MESSAGE_MODE        := - Debug mode deteted, compiling with debugging options
SERPENT_CFLAGS      += -DDEBUG -g
SERPENT_LDFLAGS     += -g
else
MESSAGE_MODE        := - Compiling in optimized mode
endif

# Link-time optimization
ifeq ($(USE_LTO),TRUE)
MESSAGE_LTO         := - Using LTO, linking '$(SERPENT_LIB_NAME)' may take a minute or
  ↳more
SERPENT_CFLAGS      += -flto
SERPENT_LDFLAGS     += -flto
else
MESSAGE_LTO         := - No LTO, expect slower Serpent runs
endif

################################################################################

# Create a pretty message that prints out when building Serpent

define MESSAGE_NOTIFICATION
========================= SERPENT BUILD CONFIGURATION =========================
  $(MESSAGE_DIRECTORY)
  $(MESSAGE_MPI)
  $(MESSAGE_OPENMP)
  $(MESSAGE_GRAPHICS)
  $(MESSAGE_MODE)
  $(MESSAGE_LTO)
========================= SERPENT BUILD CONFIGURATION =========================
endef
export MESSAGE_NOTIFICATION

################################################################################

# Determine the level of rebuild needed in the Serpent directory

ifneq (,$(findstring Nothing,$(TRICK_SCRIPT_TO_RUN)))
```

```
SERPENT_UPDATES       = 0
else ifneq (,$(findstring Patching,$(TRICK_SCRIPT_TO_RUN)))
# Header files were modified, we need to rebuild the entire source tree
SERPENT_UPDATES             = 2
$(shell rm -f $(SERPENT_LIB); rm -f $(SERPENT_DEPS); rm -f $(SERPENT_OBJ))
else
# Only source files were modified, recompile and rebuild as neccessary
SERPENT_UPDATES             = 1
endif
ifeq (,$(wildcard $(SERPENT_LIB)))
# The library is not found, so rebuild
SERPENT_UPDATES             = 1
endif

###############################################################################

# Create the rule to build the Serpent library

# If this target is hit then at least one Serpent source file is being recompiled, so the library
# will need rebuilt
$(LOCAL_SERPENT_DIR)/%.$(obj-suffix) : $(LOCAL_SERPENT_DIR)/%.c
    @echo "MOOSE Compiling Serpent (in "$(METHOD)" mode) "$<"..."
    @$(libmesh_LIBTOOL) --tag=CC $(LIBTOOLFLAGS) --mode=compile --quiet \
        $(libmesh_CC) $(libmesh_CPPFLAGS) $(SERPENT_CFLAGS) $(libmesh_CFLAGS) -MMD -MP -MF $@.d -MT $@ -c $< -o $@

$(SERPENT_LIB): pre_install_notifications $(SERPENT_OBJ)
    @if [ "$(SERPENT_UPDATES)" -ne 0 ]; then \
        echo "Linking Library "$@"..."; \
        $(libmesh_LIBTOOL) --tag=CC $(LIBTOOLFLAGS) --mode=link --quiet \
            $(libmesh_CC) $(libmesh_CFLAGS) -o $@ $(SERPENT_OBJ) \
            $(libmesh_LDFLAGS) $(SERPENT_LDFLAGS) $(EXTERNAL_FLAGS) \
            -rpath $(LOCAL_SERPENT_DIR);\
        $(libmesh_LIBTOOL) --mode=install --quiet install -c $(SERPENT_LIB) $(LOCAL_SERPENT_DIR); \
    fi

pre_install_notifications:
    @if [ "$(SERPENT_UPDATES)" -eq 0 ]; then \
        echo "No Serpent code modifications detected, leaving "$(SERPENT_LIB_NAME)" as is."; \
        fi
    @echo "$$MESSAGE_NOTIFICATION"

###############################################################################

# Add on the the 'clean' and 'clobber' rules to make sure we handle the local
# Serpent files correctly

clean::
    @rm -f $(SERPENT_SRC)
    @rm -f $(SERPENT_LIB)
    @rm -f $(SERPENT_DEPS)
    @rm -f $(SERPENT_OBJ)

clobber::
    @if [ -d "$(LOCAL_SERPENT_DIR)" ]; then \
        echo Removing directory $(LOCAL_SERPENT_DIR); \
        rm -rf $(LOCAL_SERPENT_DIR); \
    fi

###############################################################################
```

File E.18 – A bash script that copies Serpent to a local directory and modifies the files for compatibility with MOOSE. update_serpent.sh

```
#!/bin/bash
# This script copies the files from ${SERPENT_HOME} to the argument provided,
# defaulting to ./serpent otherwise, them modifies the headers files for use
# within MOOSE

# Array checker function
array_contains () {
    local array="$1[@]"
    local seeking=$2
    for element in "${!array}"; do
        if [[ $element == $seeking ]]; then
            return 3
        fi
    done
    return 0
}

MODIFIED=0
PATCH_HEADER=0

# Ignored source files
IGNORED=("deffcordata.c")

# Acquire/define the directory for the local copy
CURRENT_DIR=$(pwd)
COPY_TO_DIR=${1:-${CURRENT_DIR}/serpent}

# Ensure our two directories are different
if [[ ${COPY_TO_DIR} -ef ${SERPENT_HOME} ]]; then
  printf "Cannot copy when the source and destination directories are the same!\n"
  exit 1
fi

# Ensure the destination directory exists and then copy
mkdir -p ${COPY_TO_DIR}
printf "Copying header files from %s to %s... \n" ${SERPENT_HOME} ${COPY_TO_DIR}
cd ${SERPENT_HOME}
for file in *.h ; do
  # Only copy files that are suspect of being modified
  if [[ ${file} -nt ${COPY_TO_DIR}/${file} ]]; then
    array_contains IGNORED ${file}
    if [[ $? -ne 0 ]]; then
      printf "Skipping ignored file %s\n" ${file}
    else
      printf "%s is newer, copying...\n" ${file}
      cp -p ${file} ${COPY_TO_DIR}
      PATCH_HEADER=1
    fi
  fi
done
printf "Copying source files from %s to %s... \n" ${SERPENT_HOME} ${COPY_TO_DIR}
for file in *.c ; do
  # Only copy files that are suspect of being modified
  if [[ ${file} -nt ${COPY_TO_DIR}/${file} ]]; then
    array_contains IGNORED ${file}
    if [[ $? -ne 0 ]]; then
      printf "Skipping ignored file %s\n" ${file}
    else
      printf "%s is newer, copying...\n" ${file}
      cp -p ${file} ${COPY_TO_DIR}
      MODIFIED=1
    fi
  fi
```

```
done

if [[ "${PATCH_HEADER}" -ne 0 ]]; then
  # Move to the destination directory
  cd ${COPY_TO_DIR}

  printf "Patching header files... \n"
  for file in *.h; do
    # Only patch files that are suspect of being unmodified
    if [[ ! ${file} -nt ${SERPENT_HOME}/${file} ]]; then
      # There are some special things that need to be done for header.h
      if [[ "${file}" == "header.h" ]]; then
        # Remove some definitions already made in MOOSE
        sed -i.old '/GNU_SOURCE/ s;^;//;' header.h

        # Put header.h in an extern "C" block and use the internal definition of _XOPEN_SOURCE
        sed -i.old '1s;^;#undef _XOPEN_SOURCE\n#ifdef __cplusplus\nextern "C"{\n#endif\n\n;' header.h
        printf "#ifdef __cplusplus\n}\n#endif\n\n#undef _XOPEN_SOURCE\n#define _XOPEN_SOURCE 700" >> header.h

        # Prevent multiple declarations of global variables and structs
        start="Global arrays and variables"
        # This shouldn't appear anywere, so the scan should go through to the end-of-file
        end="EOF"
        # These are all the variable types (and the "int mpi*" cases) that need protected
        match="\(double \*\|char \*\|unsigned long\|FILE \*\|int mpi\|struct {\)"
        # Prepend "extern" **only** when compiling in C++ mode. This causes a definition to be
        # created by the C compiler when building the library. Conversely, only a declaration is
        # made to the C++ compiler when including header.h in a C++ file.
        prepend="#ifdef __cplusplus\n""extern""\n#endif\n"
        # Make the magic happen with sed
        sed -i.old "/${start}/,/${end}/{/${match}/s/^/${prepend}/}" header.h
       fi

      # Place an include guard on all header files
      guard=${file//./_}
      guard=${guard^^}
      sed -i.old "1s;^;#ifndef ${guard}\n#define ${guard}\n\n;" ${file}
      printf "\n\n#endif // ${guard}\n" >> ${file}
      rm ${file}.old
    fi
  done
fi

if [[ "${MODIFIED}" -eq 0 && "${PATCH_HEADER}" -eq 0 ]]; then
  printf "Nothing modified.\n"
fi
```

Appendix F

MOOSE Functional Expansion Module Code

This appendix contains the code of the initial MOOSE functional expansion contribution. Essentially, it is commit 128ecd3 to the MOOSE repository on February 15, 2018, stripped down to the source files and other significant contributions. The auxiliary documentation files, build files, and other standardized MOOSE requirements are not included for compactness.

F.1 AuxKernels

File F.1 – An AuxKernel that expands an FE into an AuxVariable. FunctionSeriesToAux.h

```
//* This file is part of the MOOSE framework
//* https://www.mooseframework.org
//*
//* All rights reserved, see COPYRIGHT for full restrictions
//* https://github.com/idaholab/moose/blob/master/COPYRIGHT
//*
//* Licensed under LGPL 2.1, please see LICENSE for details
//* https://www.gnu.org/licenses/lgpl-2.1.html

#ifndef FUNCTIONSERIESTOAUX_H
#define FUNCTIONSERIESTOAUX_H

#include "FunctionAux.h"

class FunctionSeriesToAux;

template <>
InputParameters validParams<FunctionSeriesToAux>();

/**
 * Specialization of FunctionAux that is designed to work specifically with FXs, namely that it is
 * always processed at timestep_begin
 */
class FunctionSeriesToAux : public FunctionAux
{
public:
  FunctionSeriesToAux(const InputParameters & parameters);
};

#endif // FUNCTIONSERIESTOAUX_H
```

File F.2 – An AuxKernel that expands an FE into an AuxVariable. FunctionSeriesToAux.C

```
//* This file is part of the MOOSE framework
//* https://www.mooseframework.org
//*
//* All rights reserved, see COPYRIGHT for full restrictions
//* https://github.com/idaholab/moose/blob/master/COPYRIGHT
//*
//* Licensed under LGPL 2.1, please see LICENSE for details
//* https://www.gnu.org/licenses/lgpl-2.1.html

#include "FunctionSeries.h"
#include "FunctionSeriesToAux.h"

template <>
InputParameters
validParams<FunctionSeriesToAux>()
{
  InputParameters params = validParams<FunctionAux>();

  params.addClassDescription("AuxKernel to convert a functional expansion"
                             " (Functions object, type = FunctionSeries) to an AuxVariable");

  // Force this AuxKernel to execute at "timestep_begin"
  params.set<ExecFlagEnum>("execute_on", true) = EXEC_TIMESTEP_BEGIN;
```

```
24   // Don't let the user change the execution time
25   params.suppressParameter<ExecFlagEnum>("execute_on");
26
27   return params;
28 }
29
30 FunctionSeriesToAux::FunctionSeriesToAux(const InputParameters & parameters)
31   : FunctionAux(parameters)
32 {
33   FunctionSeries::checkAndConvertFunction(_func, getParam<std::string>("_moose_base"),
     name());
34 }
```

F.2 BCs

File F.3 – A boundary condition that expands an FE as an interface flux. FXFluxBC.h 

```
//* This file is part of the MOOSE framework
//* https://www.mooseframework.org
//*
//* All rights reserved, see COPYRIGHT for full restrictions
//* https://github.com/idaholab/moose/blob/master/COPYRIGHT
//*
//* Licensed under LGPL 2.1, please see LICENSE for details
//* https://www.gnu.org/licenses/lgpl-2.1.html

#ifndef FXFLUXBC_H
#define FXFLUXBC_H

#include "FunctionNeumannBC.h"

class FXFluxBC;

template <>
InputParameters validParams<FXFluxBC>();

/**
 * Defines an FX-based BC that strongly encourages the gradients to match
 */
class FXFluxBC : public FunctionNeumannBC
{
public:
  FXFluxBC(const InputParameters & parameters);
};

#endif // FXFLUXBC_H
```

File F.4 – A boundary condition that expands an FE as an interface flux. FXFluxBC.C 

```
//* This file is part of the MOOSE framework
//* https://www.mooseframework.org
//*
//* All rights reserved, see COPYRIGHT for full restrictions
//* https://github.com/idaholab/moose/blob/master/COPYRIGHT
//*
//* Licensed under LGPL 2.1, please see LICENSE for details
//* https://www.gnu.org/licenses/lgpl-2.1.html

#include "FXFluxBC.h"
#include "FunctionSeries.h"

template <>
InputParameters
validParams<FXFluxBC>()
{
  InputParameters params = validParams<FunctionNeumannBC>();

  params.addClassDescription(
      "Sets a flux boundary condition, evaluated using a FunctionSeries instance. This does not "
      "fix the flux, but rather 'strongly encourages' flux agreement by penalizing the differences "
      "through contributions to the residual.");

  return params;
}
```

```
26
27 FXFluxBC::FXFluxBC(const InputParameters & parameters) : FunctionNeumannBC(parameters)
28 {
29   FunctionSeries & fe_basis =
30       FunctionSeries::checkAndConvertFunction(_func, getParam<std::string>("_moose_base")
     ↳, name());
31
32   fe_basis.useCache(true);
33 }
```

File F.5 – A boundary condition that expands an FE as an interface value. FXValueBC.h

```
1  //* This file is part of the MOOSE framework
2  //* https://www.mooseframework.org
3  //*
4  //* All rights reserved, see COPYRIGHT for full restrictions
5  //* https://github.com/idaholab/moose/blob/master/COPYRIGHT
6  //*
7  //* Licensed under LGPL 2.1, please see LICENSE for details
8  //* https://www.gnu.org/licenses/lgpl-2.1.html
9
10 #ifndef FXVALUEBC_H
11 #define FXVALUEBC_H
12
13 #include "FunctionDirichletBC.h"
14
15 class FXValueBC;
16
17 template <>
18 InputParameters validParams<FXValueBC>();
19
20 /**
21  * Defines an FX-based boundary condition that forces the values to match
22  */
23 class FXValueBC : public FunctionDirichletBC
24 {
25 public:
26   FXValueBC(const InputParameters & parameters);
27 };
28
29 #endif // FXVALUEBC_H
```

File F.6 – A boundary condition that expands an FE as an interface value. FXValueBC.C

```
1  //* This file is part of the MOOSE framework
2  //* https://www.mooseframework.org
3  //*
4  //* All rights reserved, see COPYRIGHT for full restrictions
5  //* https://github.com/idaholab/moose/blob/master/COPYRIGHT
6  //*
7  //* Licensed under LGPL 2.1, please see LICENSE for details
8  //* https://www.gnu.org/licenses/lgpl-2.1.html
9
10 #include "FXValueBC.h"
11 #include "FunctionSeries.h"
12
13 template <>
14 InputParameters
15 validParams<FXValueBC>()
16 {
17   InputParameters params = validParams<FunctionDirichletBC>();
18
```

```
  params.addClassDescription(
      "Imposes a fixed value boundary condition, evaluated using a FunctionSeries
  ↳instance.");

  return params;
}

FXValueBC::FXValueBC(const InputParameters & parameters) : FunctionDirichletBC(parameters
  ↳)
{
  FunctionSeries & fe_basis =
      FunctionSeries::checkAndConvertFunction(_func, getParam<std::string>("_moose_base")
  ↳, name());

  fe_basis.useCache(true);
}
```

File F.7 – A boundary condition that expands an FE at an interface, but only penalizes differences in the solution. FXValuePenaltyBC.h

```
//* This file is part of the MOOSE framework
//* https://www.mooseframework.org
//*
//* All rights reserved, see COPYRIGHT for full restrictions
//* https://github.com/idaholab/moose/blob/master/COPYRIGHT
//*
//* Licensed under LGPL 2.1, please see LICENSE for details
//* https://www.gnu.org/licenses/lgpl-2.1.html

#ifndef FXVALUEPENALTYBC_H
#define FXVALUEPENALTYBC_H

#include "FunctionPenaltyDirichletBC.h"

class FXValuePenaltyBC;

template <>
InputParameters validParams<FXValuePenaltyBC>();

/**
 * Defines an FX-based BC that strongly encourages the values to match
 */
class FXValuePenaltyBC : public FunctionPenaltyDirichletBC
{
public:
  FXValuePenaltyBC(const InputParameters & parameters);
};

#endif // FXVALUEPENALTYBC_H
```

File F.8 – A boundary condition that expands an FE at an interface, but only penalizes differences in the solution. FXValuePenaltyBC.C

```
//* This file is part of the MOOSE framework
//* https://www.mooseframework.org
//*
//* All rights reserved, see COPYRIGHT for full restrictions
//* https://github.com/idaholab/moose/blob/master/COPYRIGHT
//*
//* Licensed under LGPL 2.1, please see LICENSE for details
//* https://www.gnu.org/licenses/lgpl-2.1.html

#include "FXValuePenaltyBC.h"
```

```
11 #include "FunctionSeries.h"
12
13 template <>
14 InputParameters
15 validParams<FXValuePenaltyBC>()
16 {
17   InputParameters params = validParams<FunctionPenaltyDirichletBC>();
18
19   params.addClassDescription(
20       "Sets a value boundary condition, evaluated using a FunctionSeries instance. This
     does not "
21       "fix the value, but rather 'strongly encourages'  value agreement by penalizing the
     "
22       "differences through contributions to  the residual.");
23
24   return params;
25 }
26
27 FXValuePenaltyBC::FXValuePenaltyBC(const InputParameters & parameters)
28   : FunctionPenaltyDirichletBC(parameters)
29 {
30   FunctionSeries & fe_basis =
31       FunctionSeries::checkAndConvertFunction(_func, getParam<std::string>("_moose_base")
     , name());
32
33   fe_basis.useCache(true);
34 }
```

F.3 Coefficients

File F.9 – Interface that provides methods for working with coefficient values that will change throughout a simulation.

MutableCoefficientsInterface.h

```
//* This file is part of the MOOSE framework
//* https://www.mooseframework.org
//*
//* All rights reserved, see COPYRIGHT for full restrictions
//* https://github.com/idaholab/moose/blob/master/COPYRIGHT
//*
//* Licensed under LGPL 2.1, please see LICENSE for details
//* https://www.gnu.org/licenses/lgpl-2.1.html

#ifndef MUTABLECOEFFICIENTSINTERFACE_H
#define MUTABLECOEFFICIENTSINTERFACE_H

#include <vector>

#include "InputParameters.h"
#include "MooseTypes.h"
#include "Restartable.h"

class MutableCoefficientsInterface;

template <>
InputParameters validParams<MutableCoefficientsInterface>();

/**
 * This class is designed to provide a uniform interface for any class that uses an array of
 * coefficients for any of its operations. In particular, the MultiAppMutableCoefficientsTransfer
 * mechanism transfers coefficients using this interface. Any derived class of
 * MutableCoefficientsInterface can easily be used in any MultiAppMutableCoefficientsTransfer-
 * based transfer.
 */
class MutableCoefficientsInterface
{
public:
  MutableCoefficientsInterface(const MooseObject * moose_object,
                               Restartable * restartable,
                               const InputParameters & parameters);

  // Coefficient access
  /**
   * Get the value of the coefficient at the corresponding index
   */
  Real operator[](std::size_t index) const;
  /**
   * Get a reference to the characteristics array
   */
  const std::vector<std::size_t> & getCharacteristics() const;
  /**
   * Get a reference to the vector of coefficients
   */
  const std::vector<Real> & getCoefficients() const;
  /**
   * Get a formatted string of the coefficients
   */
  std::string getCoefficientsTable() const;

  // Current state
  /**
   * Get the size, aka number of coefficients
```

```
59    */
60   std::size_t getSize() const;
61   /**
62    * Checks to see if another instance is compatible
63    */
64   bool isCompatibleWith(const MutableCoefficientsInterface & other) const;
65   /**
66    * Returns true if the size of the coefficient array is fixed and enforced
67    */
68   bool isSizeEnforced() const;
69   /**
70    * Toggle whether the size of the coefficient array can be changed
71    */
72   void enforceSize(bool enforce);
73
74   // Mutable aspect
75   /**
76    * Import the coefficients from another instance
77    */
78   void importCoefficients(const MutableCoefficientsInterface & other);
79   /**
80    * Resize the array, using the value for fill if the new size is larger
81    */
82   void resize(std::size_t size, Real fill = 0.0, bool fill_out_to_size = true);
83   /**
84    * Sets the characteristics array
85    */
86   void setCharacteristics(const std::vector<std::size_t> & new_characteristics);
87   /**
88    * Set the coefficients using a copy operation
89    */
90   void setCoefficients(const std::vector<Real> & new_coefficients);
91   /**
92    * Set the coefficients using a move operation (only works with temp objects)
93    */
94   void setCoefficients(std::vector<Real> && dropin_coefficients);
95
96   /**
97    * Friend operator to easily print out the array of coefficients
98    */
99   friend std::ostream & operator<<(std::ostream & stream, const
     ↳MutableCoefficientsInterface & me);
100
101 protected:
102   /**
103    * Called when the coefficients have been changed
104    */
105   virtual void coefficientsChanged(){};
106
107   /// An array of integer characteristics that can be used to check compatibility
108   std::vector<std::size_t> & _characteristics;
109
110   /// The coefficient array
111   std::vector<Real> & _coefficients;
112
113   /// Boolean that locks or allows resizing of the coefficient array
114   bool _enforce_size;
115
116   /// Boolean to flag if the coefficients should be printed when set
117   const bool _print_coefficients;
118
119 private:
120   /// MooseObject instance of `this` to provide access to `_console`
121   const ConsoleStream & _console;
122 };
123
```

```
124 #endif // MUTABLECOEFFICIENTSINTERFACE_H
```

F.4 Functions

File F.10 – A MOOSE function that exposes the functional expansion capabilities of a CompositeSeriesBasisInterface instance. FunctionSeries.h

```
//* This file is part of the MOOSE framework
//* https://www.mooseframework.org
//*
//* All rights reserved, see COPYRIGHT for full restrictions
//* https://github.com/idaholab/moose/blob/master/COPYRIGHT
//*
//* Licensed under LGPL 2.1, please see LICENSE for details
//* https://www.gnu.org/licenses/lgpl-2.1.html

#ifndef FUNCTIONSERIES_H
#define FUNCTIONSERIES_H

#include "MutableCoefficientsFunctionInterface.h"
#include "CompositeSeriesBasisInterface.h"

class FunctionSeries;

template <>
InputParameters validParams<FunctionSeries>();

/**
 * This class uses implementations of CompositeSeriesBasisInterface to generate a function based on
 * convolved function series. Its inheritance tree includes MutableCoefficientsInterface, which
 * enables easy MultiApp transfers of coefficients.
 */
class FunctionSeries : public MutableCoefficientsFunctionInterface
{
public:
  FunctionSeries(const InputParameters & parameters);

  /**
   * Static function to cast a Function to SeriesFunction
   */
  static FunctionSeries & checkAndConvertFunction(Function & function,
                                                  const std::string & typeName,
                                                  const std::string & objectName);

  // Override from MemoizedFunctionInterface
  virtual Real evaluateValue(Real t, const Point & p) override;

  /**
   * Expand the function series at the current location and with the current coefficients
   */
  Real expand();

  /**
   * Expand the function using the provided coefficients at the current location
   */
  Real expand(const std::vector<Real> & coefficients);

  /**
   * Returns the number of terms (coefficients) in the underlying function series
   */
  std::size_t getNumberOfTerms() const;

  /**
   * Returns the volume of evaluation in the functional series standardized space
   */
  Real getStandardizedFunctionVolume() const;

```

```
61    /**
62     * Returns a vector of the functional orders in the underlying functional series
63     */
64    const std::vector<std::size_t> & getOrders() const;
65
66    /**
67     * Returns a vector of the orthogonally-evaluated functional series at the current location
68     */
69    const std::vector<Real> & getOrthonormal();
70
71    /**
72     * Returns a vector of the standardly-evaluated functional series at the current location
73     */
74    const std::vector<Real> & getStandard();
75
76    /**
77     * Returns true if the provided point is within the set physical boundaries
78     */
79    bool isInPhysicalBounds(const Point & point) const;
80
81    /**
82     * Set the current evaluation location
83     */
84    void setLocation(const Point & point);
85
86    /**
87     * Returns a tabularized text stream of the currently stored coefficients
88     */
89    friend std::ostream & operator<<(std::ostream & stream, const FunctionSeries & me);
90
91  protected:
92    /// The vector holding the orders of each single series
93    const std::vector<std::size_t> _orders;
94
95    /// The physical bounds of the function series
96    const std::vector<Real> _physical_bounds;
97
98    /// Stores a pointer to the functional series object
99    std::unique_ptr<CompositeSeriesBasisInterface> _series_type;
100
101   /// Stores the name of the current functional series type
102   const MooseEnum & _series_type_name;
103
104   /*
105    * Enumerations of the possible series types for the different spatial expansions. Not all of
106    * these will be provided for any one series.
107    */
108   /// Stores the name of the single function series to use in the x direction
109   const MooseEnum & _x;
110   /// Stores the name of the single function series to use in the y direction
111   const MooseEnum & _y;
112   /// Stores the name of the single function series to use in the z direction
113   const MooseEnum & _z;
114   /// Stores the name of the single function series to use for a unit disc
115   const MooseEnum & _disc;
116
117 private:
118   /**
119    * Static function to convert an array of `unsigned int` to `std::size_t`. The MOOSE parser has
120    * issues reading a list of integers in as `std::size_t` (unsigned long), so this workaround is
121    * required in order to set `_orders` in the constructor initializer list.
122    */
123   static std::vector<std::size_t> convertOrders(const std::vector<unsigned int> & orders)
      ↳;
124 };
125
```

```
126 #endif
```

File F.11 – A MOOSE function that exposes the functional expansion capabilities of a CompositeSeriesBasisInterface instance. FunctionSeries.C 

```
1  //* This file is part of the MOOSE framework
2  //* https://www.mooseframework.org
3  //*
4  //* All rights reserved, see COPYRIGHT for full restrictions
5  //* https://github.com/idaholab/moose/blob/master/COPYRIGHT
6  //*
7  //* Licensed under LGPL 2.1, please see LICENSE for details
8  //* https://www.gnu.org/licenses/lgpl-2.1.html
9
10 #include <numeric> // Provides accumulate()
11
12 #include "FunctionalBasisInterface.h" // Provides _domain_options
13 #include "FunctionSeries.h"
14 #include "Cartesian.h"
15 #include "CylindricalDuo.h"
16
17 template <>
18 InputParameters
19 validParams<FunctionSeries>()
20 {
21   InputParameters params = validParams<MutableCoefficientsFunctionInterface>();
22
23   params.addClassDescription("This function uses a convolution of functional"
24                              " series (functional expansion or FX) to create a"
25                              " 1D, 2D, or 3D function");
26
27   // The available composite series types.
28   //   Cartesian:      1D, 2D, or 3D, depending on which of x, y, and z are present
29   //   CylindricalDuo: planar disc expansion and axial expansion
30   MooseEnum series_types("Cartesian CylindricalDuo");
31   MooseEnum single_series_types_1D("Legendre");
32   MooseEnum single_series_types_2D("Zernike");
33
34   params.addRequiredParam<MooseEnum>(
35       "series_type", series_types, "The type of function series to construct.");
36
37   /*
38    * This needs to use `unsigned int` instead of `std::size_t` because otherwise
39    * MOOSE errors at runtime
40    */
41   params.addRequiredParam<std::vector<unsigned int>>("orders",
42                                                      "The order of each series. These must be "
43                                                      "defined as \"x y z\" for Cartesian, and \"z "
44                                                      "disc\" for CylindricalDuo.");
45
46   params.addParam<std::vector<Real>>("physical_bounds",
47                                      "The physical bounds of the function series. These must be "
48                                      "defined as \"x_min x_max y_min y_max z_min z_max\" for "
49                                      "Cartesian, and \"axial_min axial_max disc_center1 "
50                                      "disc_center2 radius\" for CylindricalDuo");
51
52   params.addParam<MooseEnum>("x", single_series_types_1D, "The series to use for the x-direction.");
53   params.addParam<MooseEnum>("y", single_series_types_1D, "The series to use for the y-direction.");
```

```
54    params.addParam<MooseEnum>("z", single_series_types_1D, "The series to use for the z-direction.");
55
56    params.addParam<MooseEnum>("disc",
57                               single_series_types_2D,
58                               "The series to use for the disc. Its direction is determined by "
59                               "orthogonality to the declared direction of the axis.");
60    return params;
61  }
62
63  FunctionSeries::FunctionSeries(const InputParameters & parameters)
64    : MutableCoefficientsFunctionInterface(this, this, parameters),
65      _orders(convertOrders(getParam<std::vector<unsigned int>>("orders"))),
66      _physical_bounds(getParam<std::vector<Real>>("physical_bounds")),
67      _series_type_name(getParam<MooseEnum>("series_type")),
68      _x(getParam<MooseEnum>("x")),
69      _y(getParam<MooseEnum>("y")),
70      _z(getParam<MooseEnum>("z")),
71      _disc(getParam<MooseEnum>("disc"))
72  {
73    std::vector<MooseEnum> domains;
74    std::vector<MooseEnum> types;
75
76    if (_series_type_name == "Cartesian")
77    {
78      /*
79      * For Cartesian series, at least one of 'x', 'y', and 'z' must be specified.
80      *
81      * The individual series are always stored in x, y, z order (independent of the order in which
82      * they appear in the input file). Hence, the 'orders' and 'physical_bounds' vectors must always
83      * be specified in x, y, z order.
84      */
85      if (isParamValid("x"))
86      {
87        domains.push_back(FunctionalBasisInterface::_domain_options = "x");
88        types.push_back(_x);
89      }
90      if (isParamValid("y"))
91      {
92        domains.push_back(FunctionalBasisInterface::_domain_options = "y");
93        types.push_back(_y);
94      }
95      if (isParamValid("z"))
96      {
97        domains.push_back(FunctionalBasisInterface::_domain_options = "z");
98        types.push_back(_z);
99      }
100     if (types.size() == 0)
101       mooseError("Must specify one of 'x', 'y', or 'z' for 'Cartesian' series!");
102     _series_type = libmesh_make_unique<Cartesian>(domains, _orders, types, name());
103   }
104   else if (_series_type_name == "CylindricalDuo")
105   {
106     /*
107     * CylindricalDuo represents a disc-axial expansion, where the disc is described by a single
108     * series, such as Zernike (as opposed to a series individually representing r and a second
109     * series independently representing theta. For CylindricalDuo series, the series are always
110     * stored in the axial, planar order, independent of which order the series appear in the input
111     * file. Therefore, the _orders and _physical_bounds vectors must always appear in axial, planar
112     * order. The first entry in _domains is interpreted as the axial direction, and the following
113     * two as the planar.
114     */
115     if (isParamValid("x"))
116     {
117       domains = {FunctionalBasisInterface::_domain_options = "x",
```

```
118                 FunctionalBasisInterface::_domain_options = "y",
119                 FunctionalBasisInterface::_domain_options = "z"};
120       types.push_back(_x);
121     }
122     if (isParamValid("y"))
123     {
124       domains = {FunctionalBasisInterface::_domain_options = "y",
125                  FunctionalBasisInterface::_domain_options = "x",
126                  FunctionalBasisInterface::_domain_options = "z"};
127       types.push_back(_y);
128     }
129     if (isParamValid("z"))
130     {
131       domains = {FunctionalBasisInterface::_domain_options = "z",
132                  FunctionalBasisInterface::_domain_options = "x",
133                  FunctionalBasisInterface::_domain_options = "y"};
134       types.push_back(_z);
135     }
136
137     if (types.size() == 0)
138       mooseError("Must specify one of 'x', 'y', or 'z' for 'CylindricalDuo' series!");
139
140     if (types.size() > 1)
141       mooseError("Cannot specify more than one of 'x', 'y', or 'z' for 'CylindricalDuo'
  ↳series!");
142
143     types.push_back(_disc);
144     _series_type = libmesh_make_unique<CylindricalDuo>(domains, _orders, types, name());
145   }
146   else
147     mooseError("Unknown functional series type \"", _series_type_name, "\"");
148
149   // Set the physical bounds of each of the single series if defined
150   if (isParamValid("physical_bounds"))
151     _series_type->setPhysicalBounds(_physical_bounds);
152
153   // Resize the coefficient array as needed
154   enforceSize(false), resize(getNumberOfTerms(), 0.0), enforceSize(true);
155   setCharacteristics(_orders);
156 }
157
158 FunctionSeries &
159 FunctionSeries::checkAndConvertFunction(Function & function,
160                                         const std::string & typeName,
161                                         const std::string & objectName)
162 {
163   FunctionSeries * test = dynamic_cast<FunctionSeries *>(&function);
164   if (!test)
165     ::mooseError("In ",
166                  typeName,
167                  "-type object \"",
168                  objectName,
169                  "\": the named Function \"",
170                  function.name(),
171                  "\" must be a FunctionSeries-type object.");
172
173   return *test;
174 }
175
176 Real
177 FunctionSeries::getStandardizedFunctionVolume() const
178 {
179   return _series_type->getStandardizedFunctionVolume();
180 }
181
182 std::size_t
```

```
183 FunctionSeries::getNumberOfTerms() const
184 {
185   return _series_type->getNumberOfTerms();
186 }
187
188 const std::vector<size_t> &
189 FunctionSeries::getOrders() const
190 {
191   return _orders;
192 }
193
194 /*
195  * getAllOrthonormal() is defined in the FunctionalBasisInterface, which calls the pure virtual
196  * evaluateOrthonormal() method of the CompositeSeriesBasisInterface class, which then calls the
197  * getAllOrthonormal() method of each of the single series.
198  */
199 const std::vector<Real> &
200 FunctionSeries::getOrthonormal()
201 {
202   return _series_type->getAllOrthonormal();
203 }
204
205 /*
206  * getAllStandard() is defined in the FunctionalBasisInterface, which calls the pure virtual
207  * evaluateStandard() method of the CompositeSeriesBasisInterface class, which then calls the
208  * getAllStandard() method of each of the single series.
209  */
210 const std::vector<Real> &
211 FunctionSeries::getStandard()
212 {
213   return _series_type->getAllStandard();
214 }
215
216 /*
217  * isInPhysicalBounds() is a pure virtual method of the FunctionalBasisInterface that is defined in
218  * the CompositeSeriesBasisInterface class because it is agnostic to the underlying types of the
219  * single series.
220  */
221 bool
222 FunctionSeries::isInPhysicalBounds(const Point & point) const
223 {
224   return _series_type->isInPhysicalBounds(point);
225 }
226
227 void
228 FunctionSeries::setLocation(const Point & point)
229 {
230   _series_type->setLocation(point);
231 }
232
233 Real
234 FunctionSeries::evaluateValue(Real, const Point & point)
235 {
236   // Check that the point is within the physical bounds of the series
237   if (!isInPhysicalBounds(point))
238     return 0.0;
239
240   // Set the location at which to evaluate the series
241   setLocation(point);
242
243   return expand();
244 }
245
246 Real
247 FunctionSeries::expand()
248 {
```

```
249   return expand(_coefficients);
250 }
251
252 Real
253 FunctionSeries::expand(const std::vector<Real> & coefficients)
254 {
255   // Evaluate all of the terms in the series
256   const std::vector<Real> & terms = getStandard();
257
258   return std::inner_product(terms.begin(), terms.end(), coefficients.begin(), 0.0);
259 }
260
261 std::ostream &
262 operator<<(std::ostream & stream, const FunctionSeries & me)
263 {
264   stream << "\n\n"
265          << "FunctionSeries: " << me.name() << "\n"
266          << "         Terms: " << me.getNumberOfTerms() << "\n";
267   me._series_type->formatCoefficients(stream, me._coefficients);
268   stream << "\n\n";
269
270   return stream;
271 }
272
273 std::vector<std::size_t>
274 FunctionSeries::convertOrders(const std::vector<unsigned int> & orders)
275 {
276   return std::vector<std::size_t>(orders.begin(), orders.end());
277 }
```

File F.12 – An interface that provides memoization capabilities to a MOOSE function class. MemoizedFunctionInterface.h

```
1  //* This file is part of the MOOSE framework
2  //* https://www.mooseframework.org
3  //*
4  //* All rights reserved, see COPYRIGHT for full restrictions
5  //* https://github.com/idaholab/moose/blob/master/COPYRIGHT
6  //*
7  //* Licensed under LGPL 2.1, please see LICENSE for details
8  //* https://www.gnu.org/licenses/lgpl-2.1.html
9
10 #ifndef MEMOIZEFUNCTIONINTERFACE_H
11 #define MEMOIZEFUNCTIONINTERFACE_H
12
13 #include <map>
14
15 #include "Function.h"
16
17 #include "Hashing.h"
18
19 class MemoizedFunctionInterface;
20
21 template <>
22 InputParameters validParams<MemoizedFunctionInterface>();
23
24 /**
25  * Implementation of Function that memoizes (caches) former evaluations in an unordered map using a
26  * hash of the evaluation locations as the key. The purpose is to allow for quick evaluation of a
27  * complex function that may be reevaluated multiple times without changing the actual outputs.
28  */
29 class MemoizedFunctionInterface : public Function
30 {
31 public:
```

```
  MemoizedFunctionInterface(const InputParameters & parameters);

  // Override from MeshChangedInterface
  virtual void meshChanged() override;

  /**
   * Enable/disable the cache
   */
  void useCache(bool use);

  // Make this implementation of Function::value() final so derived classes cannot bypass the
  // memoization functionality it implements. Instead, deriving classes should implement
  // evaluateValue().
  virtual Real value(Real time, const Point & point) final;

protected:
  /**
   * Used in derived classes, equivalent to Function::value()
   */
  virtual Real evaluateValue(Real time, const Point & point) = 0;

  /**
   * Called by derived classes to invalidate the cache, perhaps due to a state change
   */
  void invalidateCache();

private:
  /// Cached evaluations for each point
  std::unordered_map<hashing::HashValue, Real> _cache;

  /// Stores the time evaluation of the cache
  Real _current_time;

  /// Flag for whether to cache values
  bool _enable_cache;

  /// Flag for whether changes in time invalidate the cache
  bool _respect_time;
};

#endif // MEMOIZEFUNCTIONINTERFACE_H
```

File F.13 – An extension of MutableCoefficientsInterface that provides specific tools for functions that work with mutable coefficients.

MutableCoefficientsFunctionInterface.h

```
//* This file is part of the MOOSE framework
//* https://www.mooseframework.org
//*
//* All rights reserved, see COPYRIGHT for full restrictions
//* https://github.com/idaholab/moose/blob/master/COPYRIGHT
//*
//* Licensed under LGPL 2.1, please see LICENSE for details
//* https://www.gnu.org/licenses/lgpl-2.1.html

#ifndef MUTABLECOEFFICIENTSFUNCTIONINTERFACE_H
#define MUTABLECOEFFICIENTSFUNCTIONINTERFACE_H

#include "FunctionInterface.h"

#include "MemoizedFunctionInterface.h"
#include "MutableCoefficientsInterface.h"

```

```
18 class MutableCoefficientsFunctionInterface;
19
20 template <>
21 InputParameters validParams<MutableCoefficientsFunctionInterface>();
22
23 /**
24  * Interface for a type of functions using coefficients that may be changed before or after a solve
25  */
26 class MutableCoefficientsFunctionInterface : public MemoizedFunctionInterface,
27                                              protected FunctionInterface,
28                                              public MutableCoefficientsInterface
29 {
30 public:
31   MutableCoefficientsFunctionInterface(const MooseObject * moose_object,
32                                        Restartable * restartable,
33                                        const InputParameters & parameters);
34
35 protected:
36   // Override from MemoizedFunctionInterface
37   virtual void coefficientsChanged() override;
38 };
39
40 #endif // MUTABLECOEFFICIENTSFUNCTIONINTERFACE_H
```

F.5 Series

File F.14 – A CompositeSeriesBasisInterface implementation that provides an FE in Cartesian geometries. Cartesian.h

```
//* This file is part of the MOOSE framework
//* https://www.mooseframework.org
//*
//* All rights reserved, see COPYRIGHT for full restrictions
//* https://github.com/idaholab/moose/blob/master/COPYRIGHT
//*
//* Licensed under LGPL 2.1, please see LICENSE for details
//* https://www.gnu.org/licenses/lgpl-2.1.html

#ifndef CARTESIAN_H
#define CARTESIAN_H

#include "CompositeSeriesBasisInterface.h"

/**
 * This class constructs a functional expansion using a separate series for each Cartesian
 * dimension. 1D, 2D, and 3D domains are supported.
 */
class Cartesian final : public CompositeSeriesBasisInterface
{
public:
  Cartesian(const std::string & who_is_using_me);
  Cartesian(const std::vector<MooseEnum> & domain,
            const std::vector<std::size_t> & order,
            const std::vector<MooseEnum> & series_types,
            const std::string & who_is_using_me);

  // Overrides from FunctionalBasisInterface
  virtual void setPhysicalBounds(const std::vector<Real> & bounds) final;
};

#endif // CARTESIAN_H
```

File F.15 – A CompositeSeriesBasisInterface implementation that provides an FE in Cartesian geometries. Cartesian.C

```
//* This file is part of the MOOSE framework
//* https://www.mooseframework.org
//*
//* All rights reserved, see COPYRIGHT for full restrictions
//* https://github.com/idaholab/moose/blob/master/COPYRIGHT
//*
//* Licensed under LGPL 2.1, please see LICENSE for details
//* https://www.gnu.org/licenses/lgpl-2.1.html

#include "Cartesian.h"
#include "Legendre.h"

Cartesian::Cartesian(const std::string & who_is_using_me)
  : CompositeSeriesBasisInterface(who_is_using_me)
{
}

Cartesian::Cartesian(const std::vector<MooseEnum> & domains,
                     const std::vector<std::size_t> & orders,
                     const std::vector<MooseEnum> & series_types,
                     const std::string & who_is_using_me)
  : CompositeSeriesBasisInterface(orders, series_types, who_is_using_me)
```

```
{
  // Initialize the pointers to each of the single series
  for (std::size_t i = 0; i < _series_types.size(); ++i)
    if (_series_types[i] == "Legendre")
    {
      std::vector<MooseEnum> local_domain = {domains[i]};
      std::vector<std::size_t> local_order = {orders[i]};
      _series.push_back(libmesh_make_unique<Legendre>(local_domain, local_order));
    }
    else
      mooseError("Cartesian: No other linear series implemented except Legendre!");

  /*
   * Set the _number_of_terms for the composite series by looping over each of the single series.
   * This also initializes _basis_evaluation with zero values and the appropriate length.
   */
  setNumberOfTerms();
}

void
Cartesian::setPhysicalBounds(const std::vector<Real> & bounds)
{
  // Each single series is assumed to be a function of a single variable so that it has two bounds
  if (bounds.size() != _series_types.size() * 2)
    mooseError("Cartesian: Mismatch between the physical bounds provided and the number of series "
               "in the functional basis!");

  // Update the _physical_bounds of each of the single series
  unsigned int j = 0;
  for (std::size_t i = 0; i < _series_types.size(); ++i, j += 2)
    _series[i]->setPhysicalBounds({bounds[j], bounds[j + 1]});
}
```

File F.16 – An interface that provides a generalization for constructing a multivariate FE. CompositeSeriesBasisInterface.h

```
//* This file is part of the MOOSE framework
//* https://www.mooseframework.org
//*
//* All rights reserved, see COPYRIGHT for full restrictions
//* https://github.com/idaholab/moose/blob/master/COPYRIGHT
//*
//* Licensed under LGPL 2.1, please see LICENSE for details
//* https://www.gnu.org/licenses/lgpl-2.1.html

#ifndef COMPOSITESERIESBASISINTERFACE_H
#define COMPOSITESERIESBASISINTERFACE_H

#include <iomanip>

#include "FunctionalBasisInterface.h"

class SingleSeriesBasisInterface;

/**
 * This class is the basis for constructing a composite---or convolved---functional series by
 * combining multiple other series together. Nonseparability is currently assumed.
 */
class CompositeSeriesBasisInterface : public FunctionalBasisInterface
{
public:
  CompositeSeriesBasisInterface(const std::string & who_is_using_me);
  CompositeSeriesBasisInterface(const std::vector<std::size_t> & orders,
```

```
                                std::vector<MooseEnum> series_types,
                                const std::string & who_is_using_me);
  virtual ~CompositeSeriesBasisInterface();

  // Disable move and copy operations
  CompositeSeriesBasisInterface(const CompositeSeriesBasisInterface &) = delete;
  CompositeSeriesBasisInterface(CompositeSeriesBasisInterface &&) = delete;
  void operator=(const CompositeSeriesBasisInterface &) = delete;
  CompositeSeriesBasisInterface & operator=(CompositeSeriesBasisInterface &&) = delete;

  // Overrides from FunctionalBasisInterface
  virtual const std::vector<Real> & getStandardizedFunctionLimits() const final;
  virtual Real getStandardizedFunctionVolume() const final;
  virtual bool isCacheInvalid() const final;
  virtual bool isInPhysicalBounds(const Point & point) const final;
  virtual void setLocation(const Point & p) final;
  // This definition must be with CSBI because it has to loop over each of the single series.
  virtual void setOrder(const std::vector<std::size_t> & orders) final;

  /**
   * Get the function limits by looping over each of the single series
   */
  std::vector<Real> combineStandardizedFunctionLimits() const;

  /**
   * Initialize the number of terms in the composite series by looping over the single series
   */
  void setNumberOfTerms();

  /**
   * Appends a tabulated form of the coefficients to the stream
   */
  virtual void formatCoefficients(std::ostream & stream,
                                  const std::vector<Real> & coefficients) const;

protected:
  // Overrides from FunctionalBasisInterface
  virtual void evaluateOrthonormal() final;
  virtual void evaluateStandard() final;

  /**
   * Evaluates the values of _basis_evaluation for either evaluateOrthonormal() or
   * evaluateStandard()
   */
  void evaluateSeries(const std::vector<std::vector<Real>> &
  ↳single_series_basis_evaluations);

  /// The series types in this composite series
  std::vector<MooseEnum> _series_types;

  /// A pointer to the single series type (one for each entry in _domains)
  std::vector<std::unique_ptr<SingleSeriesBasisInterface>> _series;

  /// The name of the MooseObject that is using this class
  const std::string & _who_is_using_me;

private:
  /// The previous point at which the series was evaluated
  Point _previous_point;

  // Hide from subclasses (everything can be done by CSBI) to prevent BAD things from happening
  using FunctionalBasisInterface::_is_cache_invalid;
  using FunctionalBasisInterface::clearBasisEvaluation;
};

#endif // COMPOSITESERIESBASISINTERFACE_H
```

File F.17 – An interface that provides a generalization for constructing a multivariate FE. CompositeSeriesBasisInterface.C

```
1  //* This file is part of the MOOSE framework
2  //* https://www.mooseframework.org
3  //*
4  //* All rights reserved, see COPYRIGHT for full restrictions
5  //* https://github.com/idaholab/moose/blob/master/COPYRIGHT
6  //*
7  //* Licensed under LGPL 2.1, please see LICENSE for details
8  //* https://www.gnu.org/licenses/lgpl-2.1.html
9
10 #include "CompositeSeriesBasisInterface.h"
11 #include "Legendre.h"
12
13 /*
14  * Default constructor creates a functional basis with one term. In order for the _series member to
15  * be initialized, we initialized it with a Legendre series.
16  */
17 CompositeSeriesBasisInterface::CompositeSeriesBasisInterface(const std::string &
      ↳who_is_using_me)
18   : FunctionalBasisInterface(1), _who_is_using_me(who_is_using_me)
19 {
20   _series.push_back(libmesh_make_unique<Legendre>());
21 }
22
23 /*
24  * The non-default constructor is where we actually loop over the series_types and initialize
25  * pointers to those members. Because we won't know the number of terms until the end of the body
26  * of the constructor, we need to call the default FunctionalBasisInterface constructor.
27  */
28 CompositeSeriesBasisInterface::CompositeSeriesBasisInterface(
29     const std::vector<std::size_t> & orders,
30     std::vector<MooseEnum> series_types,
31     const std::string & who_is_using_me)
32   : FunctionalBasisInterface(), _series_types(series_types), _who_is_using_me(
      ↳who_is_using_me)
33 {
34   if (orders.size() != _series_types.size())
35     mooseError(_who_is_using_me,
36                " calling CSBI::CSBI(...): Incorrect number of 'orders' specified for "
37                "'FunctionSeries'! Check that 'orders' is the correct length and no
      ↳invalid "
38                "enumerations are specified for the series.");
39 }
40
41 void
42 CompositeSeriesBasisInterface::evaluateOrthonormal()
43 {
44   /*
45    * Evaluate the orthonormal versions of each of the single series, and collect the results before
46    * passing them to evaluateSeries, where they will be multiplied together correctly and stored in
47    * the composite series basis evaluation.
48    */
49   std::vector<std::vector<Real>> single_series_basis_evaluation;
50   for (auto & series : _series)
51     single_series_basis_evaluation.push_back(series->getAllOrthonormal());
52
53   evaluateSeries(single_series_basis_evaluation);
54 }
55
56 void
```

```
57  CompositeSeriesBasisInterface::evaluateSeries(
58      const std::vector<std::vector<Real>> & single_series_basis_evaluations)
59  {
60    /*
61     * Appropriate number of loops based on 1-D, 2-D, or 3-D to multiply the basis evaluations of the
62     * single series together to form the basis evaluation of the entire composite series.
63     */
64    Real f1, f2, f3;
65    std::size_t term = 0;
66
67    if (single_series_basis_evaluations.size() == 1)
68      for (std::size_t i = 0; i < _series[0]->getNumberOfTerms(); ++i, ++term)
69        save(term, single_series_basis_evaluations[0][i]);
70
71    if (single_series_basis_evaluations.size() == 2)
72      for (std::size_t i = 0; i < _series[0]->getNumberOfTerms(); ++i)
73      {
74        f1 = single_series_basis_evaluations[0][i];
75        for (std::size_t j = 0; j < _series[1]->getNumberOfTerms(); ++j, ++term)
76        {
77          f2 = single_series_basis_evaluations[1][j];
78          save(term, f1 * f2);
79        }
80      }
81
82    if (single_series_basis_evaluations.size() == 3)
83      for (std::size_t i = 0; i < _series[0]->getNumberOfTerms(); ++i)
84      {
85        f1 = single_series_basis_evaluations[0][i];
86        for (std::size_t j = 0; j < _series[1]->getNumberOfTerms(); ++j)
87        {
88          f2 = single_series_basis_evaluations[1][j];
89          for (std::size_t k = 0; k < _series[2]->getNumberOfTerms(); ++k, ++term)
90          {
91            f3 = single_series_basis_evaluations[2][k];
92            save(term, f1 * f2 * f3);
93          }
94        }
95      }
96  }
97
98  void
99  CompositeSeriesBasisInterface::evaluateStandard()
100 {
101   /*
102    * Evaluate the standard versions of each of the single series, and collect the results before
103    * passing them to evaluateSeries, where they will be multiplied together correctly and stored in
104    * the composite series basis evaluation.
105    */
106   std::vector<std::vector<Real>> single_series_basis_evaluation;
107   for (auto & series : _series)
108     single_series_basis_evaluation.push_back(series->getAllStandard());
109
110   evaluateSeries(single_series_basis_evaluation);
111 }
112
113 const std::vector<Real> &
114 CompositeSeriesBasisInterface::getStandardizedFunctionLimits() const
115 {
116   static const std::vector<Real> function_limits = combineStandardizedFunctionLimits();
117
118   return function_limits;
119 }
120
121 Real
122 CompositeSeriesBasisInterface::getStandardizedFunctionVolume() const
```

```
123 {
124   Real function_volume = 1.0;
125
126   for (auto & series : _series)
127     function_volume *= series->getStandardizedFunctionVolume();
128
129   return function_volume;
130 }
131
132 std::vector<Real>
133 CompositeSeriesBasisInterface::combineStandardizedFunctionLimits() const
134 {
135   std::vector<Real> function_limits;
136
137   for (auto & series : _series)
138   {
139     std::vector<Real> local_limits = series->getStandardizedFunctionLimits();
140     for (auto & limit : local_limits)
141       function_limits.push_back(limit);
142   }
143
144   return function_limits;
145 }
146
147 void
148 CompositeSeriesBasisInterface::formatCoefficients(std::ostream & stream,
149                                                   const std::vector<Real> & coefficients)
    const
150 {
151   // clang-format off
152   std::ostringstream formatted, domains, orders;
153   std::size_t term = 0;
154
155   stream <<               "---------------- Coefficients ----------------\n"
156          <<               "               == Subindices ==\n";
157
158   if (_series_types.size() == 1)
159   {
160     orders <<             "         Orders: " << std::setw(3) << _series[0]->getOrder(0)
    << "\n";
161     domains <<            " == Index  ==    " << std::setw(3) << _series[0]->_domains[0]
162             <<                                  "             === Value ===\n"
163             <<            "-------------------------------------------------\n";
164
165     for (std::size_t i = 0; i < _series[0]->getNumberOfTerms(); ++i, ++term)
166       formatted <<        "    " << std::setw(4) << term
167                 <<                  "         " << std::setw(3) << i
168                 <<                                "             " << std::setw(12) <<
    coefficients[term] << "\n";
169   }
170   else if (_series_types.size() == 2)
171   {
172     orders <<             "         Orders: " << std::setw(3) << _series[0]->getOrder(0)
173            <<                                  " " << std::setw(3) << _series[1]->getOrder
    (0) << "\n";
174     domains <<            " == Index  ==    " << std::setw(3) << _series[0]->_domains[0]
175             <<                                  " " << std::setw(3) << _series[1]->_domains
    [0]
176             <<                                    "          === Value ===\n"
177             <<            "-------------------------------------------------\n";
178
179     for (std::size_t i = 0; i < _series[0]->getNumberOfTerms(); ++i)
180     {
181       for (std::size_t j = 0; j < _series[1]->getNumberOfTerms(); ++j, ++term)
182         formatted <<      "    " << std::setw(4) << term
183                   <<                "         " << std::setw(3) << i
```

```
184                   <<                              " " << std::setw(3) << j
185                   <<                                  "          " << std::setw(12) << coefficients[term] << "\n";
186       }
187     }
188     else if (_series_types.size() == 3)
189     {
190       orders <<              "          Orders: " << std::setw(3) << _series[0]->getOrder(0)
191              <<                                  " " << std::setw(3) << _series[1]->getOrder(0)
192              <<                                      " " << std::setw(3) << _series[2]->getOrder(0) << "\n";
193       domains <<             " == Index  ==     " << std::setw(3) << _series[0]->_domains[0]
194               <<                                 " " << std::setw(3) << _series[1]->_domains[0]
195               <<                                     " " << std::setw(3) << _series[2]->_domains[0]
196               <<                                         "     === Value ===\n"
197               <<             "-------------------------------------------\n";
198
199       for (std::size_t i = 0; i < _series[0]->getNumberOfTerms(); ++i)
200       {
201         for (std::size_t j = 0; j < _series[1]->getNumberOfTerms(); ++j)
202         {
203           for (std::size_t k = 0; k < _series[2]->getNumberOfTerms(); ++k, ++term)
204             formatted <<    "    " << std::setw(4) << term
205                       <<            "          " << std::setw(3) << i
206                       <<                          " " << std::setw(3) << j
207                       <<                              " " << std::setw(3) << k
208                       <<                                  "     " << std::setw(12) << coefficients[term] << "\n";
209         }
210       }
211     }
212     // clang-format on
213
214     stream << orders.str() << domains.str() << formatted.str();
215   }
216
217   bool
218   CompositeSeriesBasisInterface::isCacheInvalid() const
219   {
220     /*
221      * If any one of the single series have an invalid cache, then we need to re-evaluate the entire
222      * composite series because the terms are multiplied.
223      */
224     for (auto & series : _series)
225       if (series->isCacheInvalid())
226         return true;
227
228     return false;
229   }
230
231   bool
232   CompositeSeriesBasisInterface::isInPhysicalBounds(const Point & point) const
233   {
234     /*
235      * A point is in the physical bounds of the composite series if it is in the physical bounds of
236      * each of the single series
237      */
238     for (auto & series : _series)
239       if (!series->isInPhysicalBounds(point))
240         return false;
241
242     return true;
243   }
```

```
244
245 void
246 CompositeSeriesBasisInterface::setNumberOfTerms()
247 {
248   unsigned int number_of_terms = 1;
249
250   // Accumulate the number of terms for each series
251   for (auto & series : _series)
252     number_of_terms *= series->getNumberOfTerms();
253
254   _number_of_terms = number_of_terms;
255
256   /*
257    * The length of the _basis_evaluation depends on the number of terms, so we need to clear the
258    * entries because the number of terms in the composite series may have changed.
259    */
260   clearBasisEvaluation(_number_of_terms);
261 }
262
263 void
264 CompositeSeriesBasisInterface::setOrder(const std::vector<std::size_t> & orders)
265 {
266   // One order must be specified for each single series
267   if (orders.size() != _series.size())
268     mooseError(_who_is_using_me,
269                " calling CSBI::setOrder(): Mismatch between the orders provided and the
  ↳number of "
270                "series in the functional basis!");
271
272   // Update the orders of each of the single series
273   for (std::size_t i = 0; i < _series.size(); ++i)
274     _series[i]->setOrder({orders[i]});
275
276   /*
277    * After changing the order of each single series, we need to recompute the number of terms by
278    * looping over those single series. This also clears the basis evaluation of the composite
279    * series.
280    */
281   setNumberOfTerms();
282 }
283
284 void
285 CompositeSeriesBasisInterface::setLocation(const Point & point)
286 {
287   // Return if this point is the same as the last at which the composite series was evaluated
288   if (point.absolute_fuzzy_equals(_previous_point))
289     return;
290
291   // Set the location of each of the single series
292   for (auto & series : _series)
293     series->setLocation(point);
294
295   // Store the previous point
296   _previous_point = point;
297 }
298
299 CompositeSeriesBasisInterface::~CompositeSeriesBasisInterface() {}
```

File F.18 – A CompositeSeriesBasisInterface implementation that provides an FE in cylindrical geometries. CylindricalDuo.h

```
1 //* This file is part of the MOOSE framework
2 //* https://www.mooseframework.org
3 //*
```

```
//* All rights reserved, see COPYRIGHT for full restrictions
//* https://github.com/idaholab/moose/blob/master/COPYRIGHT
//*
//* Licensed under LGPL 2.1, please see LICENSE for details
//* https://www.gnu.org/licenses/lgpl-2.1.html

#ifndef CYLINDRICALDUO_H
#define CYLINDRICALDUO_H

#include "CompositeSeriesBasisInterface.h"

/**
 * This class constructs a functional expansion in cylindrical space using a 1D series for the axial
 * direction and a 2D disc series for (r, t).
 */
class CylindricalDuo final : public CompositeSeriesBasisInterface
{
public:
  CylindricalDuo(const std::string & who_is_using_me);
  CylindricalDuo(const std::vector<MooseEnum> & domain,
                 const std::vector<std::size_t> & order,
                 const std::vector<MooseEnum> & series_types,
                 const std::string & who_is_using_me);

  // Virtual overrides
  virtual void setPhysicalBounds(const std::vector<Real> & bounds) final;
};

#endif // CYLINDRICALDUO_H
```

File F.19 – A CompositeSeriesBasisInterface implementation that provides an FE in cylindrical geometries. CylindricalDuo.C

```
//* This file is part of the MOOSE framework
//* https://www.mooseframework.org
//*
//* All rights reserved, see COPYRIGHT for full restrictions
//* https://github.com/idaholab/moose/blob/master/COPYRIGHT
//*
//* Licensed under LGPL 2.1, please see LICENSE for details
//* https://www.gnu.org/licenses/lgpl-2.1.html

#include "CylindricalDuo.h"
#include "Legendre.h"
#include "Zernike.h"

CylindricalDuo::CylindricalDuo(const std::string & who_is_using_me)
  : CompositeSeriesBasisInterface(who_is_using_me)
{
}

CylindricalDuo::CylindricalDuo(const std::vector<MooseEnum> & domains,
                               const std::vector<std::size_t> & orders,
                               const std::vector<MooseEnum> & series_types,
                               const std::string & who_is_using_me)
  : CompositeSeriesBasisInterface(orders, series_types, who_is_using_me)
{
  // Initialize the pointer to the axial series
  if (_series_types[0] == "Legendre")
  {
    std::vector<MooseEnum> local_domain = {domains[0]};
    std::vector<std::size_t> local_order = {orders[0]};
    _series.push_back(libmesh_make_unique<Legendre>(local_domain, local_order));
  }
```

```
  else
    mooseError("CylindricalDuo: No other linear series implemented except Legendre!");

  // Initialize the pointer to the disc series
  if (_series_types[1] == "Zernike")
  {
    std::vector<MooseEnum> local_domain = {domains[1], domains[2]};
    std::vector<std::size_t> local_order = {orders[1]};
    _series.push_back(libmesh_make_unique<Zernike>(local_domain, local_order));
  }
  else
    mooseError("CylindricalDuo: No other disc series implemented except Zernike!");

  /*
   * Set the _number_of_terms for the composite series by looping over each of the single series.
   * This also initializes _basis_evaluation with zero values and the appropriate length.
   */
  setNumberOfTerms();
}

void
CylindricalDuo::setPhysicalBounds(const std::vector<Real> & bounds)
{
  // The axial direction will have two bounds, the disc three
  if (bounds.size() != 5)
    mooseError("CylindricalDuo: Must provide 3 physical bounds: axial_min axial_max
  ↳disc_center1 "
               "disc_center2 radius");

  _series[0]->setPhysicalBounds({bounds[0], bounds[1]});
  _series[1]->setPhysicalBounds({bounds[2], bounds[3], bounds[4]});
}
```

File F.20 – An interface that provides a base for constructing function series. FunctionalBasisInterface.h

```
//* This file is part of the MOOSE framework
//* https://www.mooseframework.org
//*
//* All rights reserved, see COPYRIGHT for full restrictions
//* https://github.com/idaholab/moose/blob/master/COPYRIGHT
//*
//* Licensed under LGPL 2.1, please see LICENSE for details
//* https://www.gnu.org/licenses/lgpl-2.1.html

#ifndef FUNCTIONALBASISINTERFACE_H
#define FUNCTIONALBASISINTERFACE_H

#include "MooseEnum.h"
#include "MooseError.h"
#include "MooseTypes.h"

// Shortened typename
class FunctionalBasisInterface;

/**
 * This class provides the basis for any custom functional basis, and is the parent class of both
 * SingleSeriesBasisInterface and CompositeSeriesBasisInterface
 */
class FunctionalBasisInterface
{
public:
  FunctionalBasisInterface();
  FunctionalBasisInterface(const unsigned int number_of_terms);
```

```
29
30  /**
31   * Returns the current evaluation at the given index
32   */
33  Real operator[](std::size_t index) const;
34
35  /**
36   * Returns an array reference containing the value of each orthonormalized term
37   */
38  const std::vector<Real> & getAllOrthonormal();
39
40  /**
41   * Returns an array reference containing the value of each standardized term
42   */
43  const std::vector<Real> & getAllStandard();
44
45  /**
46   * Returns the number of terms in the series
47   */
48  std::size_t getNumberOfTerms() const;
49
50  /**
51   * Gets the last term of the orthonormalized functional basis
52   */
53  Real getOrthonormal();
54
55  /**
56   * Gets the sum of all terms in the orthonormalized functional basis
57   */
58  Real getOrthonormalSeriesSum();
59
60  /**
61   * Gets the #_order-th term of the standardized functional basis
62   */
63  Real getStandard();
64
65  /**
66   * Evaluates the sum of all terms in the standardized functional basis up to #_order
67   */
68  Real getStandardSeriesSum();
69
70  /**
71   * Returns a vector of the lower and upper bounds of the standardized functional space
72   */
73  virtual const std::vector<Real> & getStandardizedFunctionLimits() const = 0;
74
75  /**
76   * Returns the volume within the standardized function local_limits
77   */
78  virtual Real getStandardizedFunctionVolume() const = 0;
79
80  /**
81   * Returns true if the current evaluation is orthonormalized
82   */
83  bool isOrthonormal() const;
84
85  /**
86   * Returns true if the current evaluation is standardized
87   */
88  bool isStandard() const;
89
90  /**
91   * Whether the cached values correspond to the current point
92   */
93  virtual bool isCacheInvalid() const = 0;
94
```

```
95    /**
96     * Determines if the point provided is in within the physical bounds
97     */
98    virtual bool isInPhysicalBounds(const Point & point) const = 0;
99
100   /**
101    * Set the location that will be used by the series to compute values
102    */
103   virtual void setLocation(const Point & point) = 0;
104
105   /**
106    * Set the order of the series
107    */
108   virtual void setOrder(const std::vector<std::size_t> & orders) = 0;
109
110   /**
111    * Sets the bounds of the series
112    */
113   virtual void setPhysicalBounds(const std::vector<Real> & bounds) = 0;
114
115   /// An enumeration of the domains available to each functional series
116   static MooseEnum _domain_options;
117
118 protected:
119   /**
120    * Set all entries of the basis evaluation to zero.
121    */
122   virtual void clearBasisEvaluation(const unsigned int & number_of_terms);
123
124   /**
125    * Evaluate the orthonormal form of the functional basis
126    */
127   virtual void evaluateOrthonormal() = 0;
128
129   /**
130    * Evaluate the standardized form of the functional basis
131    */
132   virtual void evaluateStandard() = 0;
133
134   /**
135    * Helper function to load a value from #_series
136    */
137   Real load(std::size_t index) const;
138
139   /**
140    * Helper function to store a value in #_series
141    */
142   void save(std::size_t index, Real value);
143
144   /// The number of terms in the series
145   unsigned int _number_of_terms;
146
147   /// indicates if the evaluated values correspond to the current location
148   bool _is_cache_invalid;
149
150 private:
151   /// Stores the values of the basis evaluation
152   std::vector<Real> _basis_evaluation;
153
154   /// Indicates whether the current evaluation is standardized or orthonormalized
155   bool _is_orthonormal;
156 };
157
158 #endif // FUNCTIONALBASISINTERFACE_H
```

File F.21 – An interface that provides a base for constructing function series. FunctionalBasisInterface.C

```
1  //* This file is part of the MOOSE framework
2  //* https://www.mooseframework.org
3  //*
4  //* All rights reserved, see COPYRIGHT for full restrictions
5  //* https://github.com/idaholab/moose/blob/master/COPYRIGHT
6  //*
7  //* Licensed under LGPL 2.1, please see LICENSE for details
8  //* https://www.gnu.org/licenses/lgpl-2.1.html
9
10 #include "FunctionalBasisInterface.h"
11
12 MooseEnum FunctionalBasisInterface::_domain_options("x=0 y=1 z=2");
13
14 /*
15  * The default constructor is used to initialize a series before knowing the number of terms in the
16  * series. This is called from the CSBI, and in the body of the CSBI constructor, setNumberOfTerms()
17  * is used to after-the-fact perform the same initializations that would be done with the
18  * non-default constructor.
19  */
20 FunctionalBasisInterface::FunctionalBasisInterface()
21   : _is_cache_invalid(true), _is_orthonormal(false)
22 {
23 }
24
25 /*
26  * The non-default constructor should be used to initialize a series if the number of terms is
27  * known, such as with a single series.
28  */
29 FunctionalBasisInterface::FunctionalBasisInterface(const unsigned int number_of_terms)
30   : _number_of_terms(number_of_terms),
31     _is_cache_invalid(true),
32     _basis_evaluation(_number_of_terms, 0.0),
33     _is_orthonormal(false)
34 {
35   _basis_evaluation.shrink_to_fit();
36 }
37
38 Real FunctionalBasisInterface::operator[](std::size_t index) const
39 {
40   return (index > _basis_evaluation.size() ? 0.0 : _basis_evaluation[index]);
41 }
42
43 bool
44 FunctionalBasisInterface::isOrthonormal() const
45 {
46   return _is_orthonormal;
47 }
48
49 bool
50 FunctionalBasisInterface::isStandard() const
51 {
52   return !_is_orthonormal;
53 }
54
55 const std::vector<Real> &
56 FunctionalBasisInterface::getAllOrthonormal()
57 {
58   if (isStandard() || isCacheInvalid())
59   {
60     clearBasisEvaluation(_number_of_terms);
61
62     evaluateOrthonormal();
63
```

```
64      _is_orthonormal = true;
65      _is_cache_invalid = false;
66    }
67
68    return _basis_evaluation;
69  }
70
71  const std::vector<Real> &
72  FunctionalBasisInterface::getAllStandard()
73  {
74    if (isOrthonormal() || isCacheInvalid())
75    {
76      clearBasisEvaluation(_number_of_terms);
77
78      evaluateStandard();
79
80      _is_orthonormal = false;
81      _is_cache_invalid = false;
82    }
83
84    return _basis_evaluation;
85  }
86
87  std::size_t
88  FunctionalBasisInterface::getNumberOfTerms() const
89  {
90    return _number_of_terms;
91  }
92
93  Real
94  FunctionalBasisInterface::getOrthonormal()
95  {
96    // Use getAllOrthonormal() which will lazily evaluate the series as needed
97    return getAllOrthonormal().back();
98  }
99
100 Real
101 FunctionalBasisInterface::getOrthonormalSeriesSum()
102 {
103   Real sum = 0.0;
104
105   // Use getAllOrthonormal() which will lazily evaluate the series as needed
106   for (auto term : getAllOrthonormal())
107     sum += term;
108
109   return sum;
110 }
111
112 Real
113 FunctionalBasisInterface::getStandard()
114 {
115   // Use getAllStandard() which will lazily evaluate the series as needed
116   return getAllStandard().back();
117 }
118
119 Real
120 FunctionalBasisInterface::getStandardSeriesSum()
121 {
122   Real sum = 0.0;
123
124   // Use getAllStandard() which will lazily evaluate the series as needed
125   for (auto term : getAllStandard())
126     sum += term;
127
128   return sum;
129 }
```

```
130
131 Real
132 FunctionalBasisInterface::load(std::size_t index) const
133 {
134   return _basis_evaluation[index];
135 }
136
137 void
138 FunctionalBasisInterface::save(std::size_t index, Real value)
139 {
140   _basis_evaluation[index] = value;
141 }
142
143 void
144 FunctionalBasisInterface::clearBasisEvaluation(const unsigned int & number_of_terms)
145 {
146   _basis_evaluation.assign(number_of_terms, 0.0);
147   _basis_evaluation.shrink_to_fit();
148 }
```

File F.22 – A SingleSeriesBasisInterface implementation that provides a 1D Legendre polynomial series.

Legendre.h 

```
1  //* This file is part of the MOOSE framework
2  //* https://www.mooseframework.org
3  //*
4  //* All rights reserved, see COPYRIGHT for full restrictions
5  //* https://github.com/idaholab/moose/blob/master/COPYRIGHT
6  //*
7  //* Licensed under LGPL 2.1, please see LICENSE for details
8  //* https://www.gnu.org/licenses/lgpl-2.1.html
9
10 #ifndef LEGENDRE_H
11 #define LEGENDRE_H
12
13 #include "SingleSeriesBasisInterface.h"
14
15 /**
16  * This class provides the algorithms and properties of the Legendre polynomial series.
17  */
18 class Legendre final : public SingleSeriesBasisInterface
19 {
20 public:
21   Legendre();
22   Legendre(const std::vector<MooseEnum> & domain, const std::vector<std::size_t> & order)
    ↳;
23
24   // Overrides from FunctionalBasisInterface
25   virtual Real getStandardizedFunctionVolume() const override;
26   virtual bool isInPhysicalBounds(const Point & point) const override;
27
28   // Overrides from SingleSeriesBasisInterface
29   virtual std::size_t
30   calculatedNumberOfTermsBasedOnOrder(const std::vector<std::size_t> & order) const
    ↳override;
31   virtual const std::vector<Real> & getStandardizedFunctionLimits() const override;
32
33 protected:
34   // Overrides from FunctionalBasisInterface
35   virtual void evaluateOrthonormal() override;
36   virtual void evaluateStandard() override;
37
38   // Overrides from SingleSeriesBasisInterface
39   virtual void checkPhysicalBounds(const std::vector<Real> & bounds) const override;
```

```
  virtual std::vector<Real>
  getStandardizedLocation(const std::vector<Real> & location) const override;
};

#endif // LEGENDRE_H
```

File F.23 – A SingleSeriesBasisInterface implementation that provides a 1D Legendre polynomial series. Legendre.C 

```
//* This file is part of the MOOSE framework
//* https://www.mooseframework.org
//*
//* All rights reserved, see COPYRIGHT for full restrictions
//* https://github.com/idaholab/moose/blob/master/COPYRIGHT
//*
//* Licensed under LGPL 2.1, please see LICENSE for details
//* https://www.gnu.org/licenses/lgpl-2.1.html

#include "Legendre.h"

/**
 * The highest order of Legendre polynomials calculated directly instead of via the recurrence
 * relation
 */
#define MAX_DIRECT_CALCULATION_LEGENDRE 12

Legendre::Legendre() : SingleSeriesBasisInterface() {}

Legendre::Legendre(const std::vector<MooseEnum> & domain, const std::vector<std::size_t>
  & order)
  : SingleSeriesBasisInterface(domain, order, calculatedNumberOfTermsBasedOnOrder(order))
{
}

std::size_t
Legendre::calculatedNumberOfTermsBasedOnOrder(const std::vector<std::size_t> & order)
  const
{
  return order[0] + 1;
}

void
Legendre::checkPhysicalBounds(const std::vector<Real> & bounds) const
{
  // Each Legendre series should have a min and max bound
  if (bounds.size() != 2)
    mooseError("Legend: Invalid number of bounds specified for single series!");
}

void
Legendre::evaluateOrthonormal()
{
  std::size_t k;
  const Real & x = _standardized_location[0];
  const Real x2 = x * x;

  /*
   * Use direct formula to efficiently evaluate the polynomials for n <= 12
   *
   * The performance benefit diminishes for higher n. It is expected that the cost of the direct
   * calculation nears that of the recurrence relation in the neighborhood of n == 15, although this
   * theory is untested due to only implementing the direct calculations up to n == 12.
   *
   * If you want to calculate the higher-order Legendre Coefficients and code them in then be my
```

```
54    * guest.
55    */
56   // clang-format off
57   switch (_orders[0])
58   {
59     default:
60     case MAX_DIRECT_CALCULATION_LEGENDRE:  /* 12 */
61       save(12, ((((((676039 * x2 - 1939938) * x2 + 2078505) * x2 - 1021020) * x2 +
     ↳225225) * x2 - 18018) * x2 + 231) / 1024
62                * 12.5);
63       libmesh_fallthrough();
64
65     case 11:
66       save(11, (((((88179 * x2 - 230945) * x2 + 218790) * x2 - 90090) * x2 + 15015) * x2
     ↳- 693) * x / 256
67                * 11.5);
68       libmesh_fallthrough();
69
70     case 10:
71       save(10, (((((46189 * x2 - 109395) * x2 + 90090) * x2 - 30030) * x2 + 3465) * x2 -
     ↳63) / 256
72                * 10.5);
73       libmesh_fallthrough();
74
75     case 9:
76       save(9, ((((12155 * x2 - 25740) * x2 + 18018) * x2 - 4620) * x2 + 315) * x / 128
77               * 9.5);
78       libmesh_fallthrough();
79
80     case 8:
81       save(8, ((((6435 * x2 - 12012) * x2 + 6930) * x2 - 1260) * x2 + 35) / 128
82               * 8.5);
83       libmesh_fallthrough();
84
85     case 7:
86       save(7, (((429 * x2 - 693) * x2 + 315) * x2 - 35) * x / 16
87               * 7.5);
88       libmesh_fallthrough();
89
90     case 6:
91       save(6, (((231 * x2 - 315) * x2 + 105) * x2 - 5) / 16
92               * 6.5);
93       libmesh_fallthrough();
94
95     case 5:
96       save(5, ((63 * x2 - 70) * x2 + 15) * x / 8
97               * 5.5);
98       libmesh_fallthrough();
99
100    case 4:
101      save(4, ((35 * x2 - 30) * x2 + 3) / 8
102              * 4.5);
103      libmesh_fallthrough();
104
105    case 3:
106      save(3, (5 * x2 - 3) * x / 2
107              * 3.5);
108      libmesh_fallthrough();
109
110    case 2:
111      save(2, (3 * x2 - 1) / 2
112              * 2.5);
113      libmesh_fallthrough();
114
115    case 1:
116      save(1, x
```

```
117               * 1.5);
118       libmesh_fallthrough();
119
120     case 0:
121       save(0, 1
122               * 0.5);
123   }
124   // clang-format on
125
126   /*
127    * Evaluate any remaining polynomials.
128    *
129    * The original recurrence relation is:
130    *       (2 * k - 1) * x * L_(k-1) - (k - 1) * L_(k-2)
131    * L_k = ----------------------------------------------
132    *                       k
133    *
134    * However, for FXs we are using a the orthonormalized version of the polynomials, so each
135    * polynomial L_k is multiplied by:
136    *    (2 * k + 1)
137    *    -----------      essentially:    k + 0.5
138    *        2
139    * Reversing this in the previous polynomials and implementing for the current polynomial results
140    * in the orthonormalized recurrence:
141    *       (2 * k + 1)   /                 (k - 1)            \
142    * L_k = ----------- * | x * L_(k-1) - ----------- * L_(k-2) |
143    *           k         \               (2 * k - 3)          /
144    *
145    * The options are 1) to use this form, or 2) to not apply the orthonormalization at first, and
146    * then loop through all the values in a second loop and then apply the orthonormalization.
147    */
148   for (k = MAX_DIRECT_CALCULATION_LEGENDRE + 1; k <= _orders[0]; ++k)
149     save(k, ((k + k + 1) / Real(k)) * (x * load(k - 1) - ((k - 1) / (k + k - 3.0)) * load
    ↳(k - 2)));
150 }
151
152 void
153 Legendre::evaluateStandard()
154 {
155   std::size_t k;
156   const Real x = _standardized_location[0];
157   const Real x2 = x * x;
158
159   /*
160    * Use direct formula to efficiently evaluate the polynomials for n <= 12
161    *
162    * The performance benefit diminishes for higher n. It is expected that the cost of the direct
163    * calculation nears that of the recurrence relation in the neighborhood of n == 15, although this
164    * theory is untested due to only implementing the direct calculations up to n == 12.
165    *
166    * If you want to calculate the higher-order Legendre Coefficients and
167    * code them in then be my guest.
168    */
169   // clang-format off
170   switch (_orders[0])
171   {
172     default:
173     case MAX_DIRECT_CALCULATION_LEGENDRE:  /* 12 */
174       save(12, ((((((676039 * x2 - 1939938) * x2 + 2078505) * x2 - 1021020) * x2 +
    ↳225225) * x2 - 18018) * x2 + 231) / 1024);
175       libmesh_fallthrough();
176
177     case 11:
178       save(11, (((((88179 * x2 - 230945) * x2 + 218790) * x2 - 90090) * x2 + 15015) * x2
    ↳- 693) * x / 256);
179       libmesh_fallthrough();
```

```
180
181     case 10:
182       save(10, (((((46189 * x2 - 109395) * x2 + 90090) * x2 - 30030) * x2 + 3465) * x2 -
    ↳63) / 256);
183       libmesh_fallthrough();
184
185     case 9:
186       save(9, ((((12155 * x2 - 25740) * x2 + 18018) * x2 - 4620) * x2 + 315) * x / 128);
187       libmesh_fallthrough();
188
189     case 8:
190       save(8, ((((6435 * x2 - 12012) * x2 + 6930) * x2 - 1260) * x2 + 35) / 128);
191       libmesh_fallthrough();
192
193     case 7:
194       save(7, (((429 * x2 - 693) * x2 + 315) * x2 - 35) * x / 16);
195       libmesh_fallthrough();
196
197     case 6:
198       save(6, (((231 * x2 - 315) * x2 + 105) * x2 - 5) / 16);
199       libmesh_fallthrough();
200
201     case 5:
202       save(5, ((63 * x2 - 70) * x2 + 15) * x / 8);
203       libmesh_fallthrough();
204
205     case 4:
206       save(4, ((35 * x2 - 30) * x2 + 3) / 8);
207       libmesh_fallthrough();
208
209     case 3:
210       save(3, (5 * x2 - 3) * x / 2);
211       libmesh_fallthrough();
212
213     case 2:
214       save(2, (3 * x2 - 1) / 2);
215       libmesh_fallthrough();
216
217     case 1:
218       save(1, x);
219       libmesh_fallthrough();
220
221     case 0:
222       save(0, 1);
223   }
224   // clang-format on
225
226   /*
227    * Evaluate any remaining polynomials.
228    * The recurrence relation is:
229    *       (2 * k - 1) * x * L_(k-1) - (k - 1) * L_(k-2)
230    * L_k = ---------------------------------------------
231    *                       k
232    */
233   for (k = MAX_DIRECT_CALCULATION_LEGENDRE + 1; k <= _orders[0]; ++k)
234     save(k, (((2 * k - 1) * x * load(k - 1)) - ((k - 1) * load(k - 2))) / Real(k));
235 }
236
237 const std::vector<Real> &
238 Legendre::getStandardizedFunctionLimits() const
239 {
240   // Lazily instantiate the function limits array
241   static const std::vector<Real> standardizedFunctionLimits = {-1, 1};
242
243   return standardizedFunctionLimits;
244 }
```

```
245
246 Real
247 Legendre::getStandardizedFunctionVolume() const
248 {
249   return 2.0; // Span of [-1, 1]
250 }
251
252 std::vector<Real>
253 Legendre::getStandardizedLocation(const std::vector<Real> & location) const
254 {
255   const Real difference = location[0] - _physical_bounds[0];
256   const Real span = _physical_bounds[1] - _physical_bounds[0];
257
258   // Convert to [0, 1] (assuming that location[0] is within _physical_bounds)
259   const Real ratio = difference / span;
260
261   // Legendre space is [-1, 1]
262   return {ratio * 2 - 1};
263 }
264
265 bool
266 Legendre::isInPhysicalBounds(const Point & point) const
267 {
268   std::vector<Real> location = extractLocationFromPoint(point);
269
270   if (location[0] < _physical_bounds[0] || _physical_bounds[1] < location[0])
271     return false;
272   else
273     return true;
274 }
```

File F.24 – An interface that provides a generalization for constructing a single function series. SingleSeriesBasisInterface.h

```
1  //* This file is part of the MOOSE framework
2  //* https://www.mooseframework.org
3  //*
4  //* All rights reserved, see COPYRIGHT for full restrictions
5  //* https://github.com/idaholab/moose/blob/master/COPYRIGHT
6  //*
7  //* Licensed under LGPL 2.1, please see LICENSE for details
8  //* https://www.gnu.org/licenses/lgpl-2.1.html
9
10 #ifndef SINGLESERIESBASISINTERFACE_H
11 #define SINGLESERIESBASISINTERFACE_H
12
13 #include "FunctionalBasisInterface.h"
14
15 /**
16  * This class is a simple wrapper around FunctionalBasisInterface, and intended for use by any
17  * single functional series like Legendre, Zernike, etc...
18  */
19 class SingleSeriesBasisInterface : public FunctionalBasisInterface
20 {
21 public:
22   SingleSeriesBasisInterface();
23   SingleSeriesBasisInterface(const std::vector<MooseEnum> & domains,
24                              const std::vector<std::size_t> & orders,
25                              const unsigned int number_of_terms);
26   virtual ~SingleSeriesBasisInterface();
27
28   // Disable move and copy operations
29   SingleSeriesBasisInterface(const SingleSeriesBasisInterface &) = delete;
30   SingleSeriesBasisInterface(SingleSeriesBasisInterface &&) = delete;
```

```
  void operator=(const SingleSeriesBasisInterface &) = delete;
  SingleSeriesBasisInterface & operator=(SingleSeriesBasisInterface &&) = delete;

  // Overrides from FunctionalBasisInterface
  virtual bool isCacheInvalid() const final;
  virtual void setLocation(const Point & point) final;
  virtual void setOrder(const std::vector<std::size_t> & orders) final;
  virtual void setPhysicalBounds(const std::vector<Real> & bounds) final;

  /**
   * Returns the number of terms in the single series given a value for the order
   */
  virtual std::size_t
  calculatedNumberOfTermsBasedOnOrder(const std::vector<std::size_t> & order) const = 0;

  /**
   * Standardize the location according to the requirements of the underlying basis, which may
   * actually convert the Cartesian coordinates into a more suitable system. The second version
   * exists simply to return the value.
   */
  virtual std::vector<Real> getStandardizedLocation(const std::vector<Real> & location)
  const = 0;

  /**
   * Returns the order of the particular domain index
   */
  std::size_t getOrder(std::size_t domain) const;

  /// An ordered list of the x, y, and/or z domains needed by the functional basis to convert a point
  /// to a standardized location
  const std::vector<MooseEnum> _domains;

protected:
  /**
   * Checks the physical bounds according to the actual implementation
   */
  virtual void checkPhysicalBounds(const std::vector<Real> & bounds) const = 0;

  /**
   * Convert a spatial point to a location that the series will use to determine the value at which
   * to evaluate the series
   */
  std::vector<Real> extractLocationFromPoint(const Point & point) const;

  /// The order of the series
  std::vector<std::size_t> _orders;

  /// The physical bounds of the series
  std::vector<Real> _physical_bounds;

  /// The standardized location of evaluation
  std::vector<Real> _standardized_location;

private:
  /// Flag for if the physical bounds are specified for this series
  bool _are_physical_bounds_specified;

  /// The domain locations of the current evaluation. This is private so that derived classes will be
  /// required to use #_standardized_location, essentially forcing location-awareness compliance
  std::vector<Real> _location;

  // Hide from subclasses to prevent BAD things from happening
  using FunctionalBasisInterface::_is_cache_invalid;
  using FunctionalBasisInterface::clearBasisEvaluation;
};

```

```
96 #endif // SINGLESERIESBASISINTERFACE_H
```

File F.25 – An interface that provides a generalization for constructing a single function series. SingleSeriesBasisInterface.C

```
1  //* This file is part of the MOOSE framework
2  //* https://www.mooseframework.org
3  //*
4  //* All rights reserved, see COPYRIGHT for full restrictions
5  //* https://github.com/idaholab/moose/blob/master/COPYRIGHT
6  //*
7  //* Licensed under LGPL 2.1, please see LICENSE for details
8  //* https://www.gnu.org/licenses/lgpl-2.1.html
9
10 #include "SingleSeriesBasisInterface.h"
11
12 /*
13  * The default constructor for a single series creates a single-term functional basis of zeroth
14  * order. Although the _physical_bounds flag is set to false anyways, we need to assign some value
15  * here in the constructor so that the order of member variables in the include file doesn't give an
16  * error. The same holds for _standardized_location.
17  */
18 SingleSeriesBasisInterface::SingleSeriesBasisInterface()
19   : FunctionalBasisInterface(1),
20     _domains({_domain_options = "x"}),
21     _orders({0}),
22     _physical_bounds(2, 0.0),
23     _standardized_location({0.0}),
24     _are_physical_bounds_specified(false),
25     _location(_domains.size(), 0.0)
26 {
27 }
28
29 SingleSeriesBasisInterface::SingleSeriesBasisInterface(const std::vector<MooseEnum> &
    domains,
30                                                        const std::vector<std::size_t> &
    orders,
31                                                        const unsigned int number_of_terms
    )
32   : FunctionalBasisInterface(number_of_terms),
33     _domains(domains),
34     _orders(orders),
35     _physical_bounds(2, 0.0),
36     _standardized_location(1, 0.0),
37     _are_physical_bounds_specified(false),
38     _location(_domains.size(), 0.0)
39 {
40 }
41
42 bool
43 SingleSeriesBasisInterface::isCacheInvalid() const
44 {
45   return _is_cache_invalid;
46 }
47
48 void
49 SingleSeriesBasisInterface::setOrder(const std::vector<std::size_t> & orders)
50 {
51   if (orders.size() != _orders.size())
52     mooseError("SSBI: Invalid 'orders' use in setOrder()!");
53
54   /*
55    * Do nothing if the order isn't changed. Note that this only compares the first value - it is
56    * assumed that a single series only needs to be described using a single order.
```

```
57    */
58   if (orders[0] == _orders[0])
59     return;
60
61   _orders = orders;
62
63   // Set the new number of terms in the single series
64   _number_of_terms = calculatedNumberOfTermsBasedOnOrder(_orders);
65
66   // Zero the basis evaluation
67   clearBasisEvaluation(_number_of_terms);
68   _is_cache_invalid = true;
69 }
70
71 void
72 SingleSeriesBasisInterface::setPhysicalBounds(const std::vector<Real> & bounds)
73 {
74   // Use the concrete implementation to check the validity of the bounds
75   checkPhysicalBounds(bounds);
76
77   _physical_bounds = bounds;
78   _are_physical_bounds_specified = true;
79
80   /*
81    * Once the physical bounds have been changed, the normalization of a point will change, so the
82    * cached values will also be incorrect
83    */
84   _is_cache_invalid = true;
85 }
86
87 void
88 SingleSeriesBasisInterface::setLocation(const Point & point)
89 {
90   std::vector<Real> oldLocation(_location);
91
92   // Update the physical-space location
93   _location = extractLocationFromPoint(point);
94
95   // Standardize the location if standardized bounds exist
96   if (_are_physical_bounds_specified)
97     _standardized_location = getStandardizedLocation(_location);
98   else
99     _standardized_location = _location;
100
101  // Once the location is changed, the cached values correspond to an old location
102  for (std::size_t i = 0; !_is_cache_invalid && (i < _location.size()); ++i)
103    if (oldLocation[i] != _location[i])
104      _is_cache_invalid = true;
105 }
106
107 std::vector<Real>
108 SingleSeriesBasisInterface::extractLocationFromPoint(const Point & point) const
109 {
110   std::vector<Real> location(_domains.size());
111
112   // Update the locations as specified by _domain
113   for (std::size_t index = 0; index < _domains.size(); ++index)
114     location[index] = point(_domains[index]);
115   return location;
116 }
117
118 std::size_t
119 SingleSeriesBasisInterface::getOrder(std::size_t domain) const
120 {
121   return domain < _orders.size() ? _orders[domain] : -1;
122 }
```

```
123
124 SingleSeriesBasisInterface::~SingleSeriesBasisInterface() {}
```

File F.26 – A SingleSeriesBasisInterface implementation that provides a 2D Zernike polynomial series. Zernike.h 

```
1  //* This file is part of the MOOSE framework
2  //* https://www.mooseframework.org
3  //*
4  //* All rights reserved, see COPYRIGHT for full restrictions
5  //* https://github.com/idaholab/moose/blob/master/COPYRIGHT
6  //*
7  //* Licensed under LGPL 2.1, please see LICENSE for details
8  //* https://www.gnu.org/licenses/lgpl-2.1.html
9
10 #ifndef ZERNIKE_H
11 #define ZERNIKE_H
12
13 #include "SingleSeriesBasisInterface.h"
14
15 /**
16  * This class provides the algorithms and properties of the Zernike polynomial series.
17  */
18 class Zernike : public SingleSeriesBasisInterface
19 {
20 public:
21   Zernike();
22
23   Zernike(const std::vector<MooseEnum> & domain, const std::vector<std::size_t> & order);
24
25   // Overrides from FunctionalBasisInterface
26   virtual Real getStandardizedFunctionVolume() const override;
27   virtual bool isInPhysicalBounds(const Point & point) const override;
28
29   // Overrides from SingleSeriesBasisInterface
30   virtual std::size_t
31   calculatedNumberOfTermsBasedOnOrder(const std::vector<std::size_t> & order) const
     override;
32   virtual const std::vector<Real> & getStandardizedFunctionLimits() const override;
33
34 protected:
35   // Overrides from FunctionalBasisInterface
36   virtual void evaluateOrthonormal() override;
37   virtual void evaluateStandard() override;
38
39   // Overrides from SingleSeriesBasisInterface
40   virtual void checkPhysicalBounds(const std::vector<Real> & bounds) const override;
41   virtual std::vector<Real>
42   getStandardizedLocation(const std::vector<Real> & location) const override;
43
44   /**
45    * Helper function used by evaluateOrthonormal() and evaluateStandard(). It
46    * uses the evaluated value array of the zero and positive rank terms to:
47    *  1) fill out the negative rank terms
48    *  2) apply the azimuthal components to all terms
49    */
50   void fillOutNegativeRankAndApplyAzimuthalComponent();
51
52   /**
53    * Maps the double order/rank idices to a single linear index
54    */
55   std::size_t simpleDoubleToSingle(std::size_t n, long m) const;
56
57   /// Stores the recurrence evaluations for the negative rank azimuthal terms
```

```
  std::vector<Real> _negative_azimuthal_components;

  /// Stores the recurrence evaluations for the positive rank azimuthal terms
  std::vector<Real> _positive_azimuthal_components;
};

#endif // ZERNIKE_H
```

File F.27 – A SingleSeriesBasisInterface implementation that provides a 2D Zernike polynomial series. Zernike.C

```
//* This file is part of the MOOSE framework
//* https://www.mooseframework.org
//*
//* All rights reserved, see COPYRIGHT for full restrictions
//* https://github.com/idaholab/moose/blob/master/COPYRIGHT
//*
//* Licensed under LGPL 2.1, please see LICENSE for details
//* https://www.gnu.org/licenses/lgpl-2.1.html

#include "MooseUtils.h"

#include "Zernike.h"

/**
 * The higherst order of Zernike polynomials calculated directly instead of via the recurrence
 * relation
 */
#define MAX_DIRECT_CALCULATION_ZERNIKE 10

Zernike::Zernike() : SingleSeriesBasisInterface() {}

Zernike::Zernike(const std::vector<MooseEnum> & domain, const std::vector<std::size_t> &
  ↳order)
  : SingleSeriesBasisInterface(domain, order, calculatedNumberOfTermsBasedOnOrder(order))
{
}

std::size_t
Zernike::calculatedNumberOfTermsBasedOnOrder(const std::vector<std::size_t> & order)
  ↳const
{
  return ((order[0] + 1) * (order[0] + 2)) / 2;
}

void
Zernike::checkPhysicalBounds(const std::vector<Real> & bounds) const
{
  /*
   * Each single series is assumed to be a function of a single coordinate, which should only have
   * two bounds.
   */
  if (bounds.size() != 3)
    mooseError("Zernike: Invalid number of bounds specified for single series!");
}

// clang-format off
void
Zernike::evaluateOrthonormal()
{
  std::size_t n;
  long j, q;
  Real H1, H2, H3;
  const Real & rho = _standardized_location[0];
```

```
52   const Real rho2 = rho * rho;
53   const Real rho4 = rho2 * rho2;
54
55   if (MooseUtils::absoluteFuzzyEqual(rho, 0.0))
56   {
57     for (n = 0; n <= _orders[0]; n += 2)
58     {
59       j = simpleDoubleToSingle(n, 0);
60
61       if ((n / 2) % 2 != 0)
62         save(j, -1 * (n + 1) / M_PI);
63       else
64         save(j, 1 * (n + 1) / M_PI);
65     }
66
67     return;
68   }
69
70   switch (_orders[0])
71   {
72     default:
73     case MAX_DIRECT_CALCULATION_ZERNIKE:  /* 10 */
74       save(65, rho4 * rho4 * rho2
75                * 22 / M_PI);
76       save(64, (10 * rho2 - 9) * rho4 * rho4
77                * 22 / M_PI);
78       save(63, ((45 * rho2 - 72) * rho2 + 28) * rho4 * rho2
79                * 22 / M_PI);
80       save(62, (((120 * rho2 - 252) * rho2 + 168) * rho2- 35) * rho4
81                * 22 / M_PI);
82       save(61, ((((210 * rho2 - 504) * rho2 + 420) * rho2 - 140) * rho2 + 15) * rho2
83                * 22 / M_PI);
84       save(60, (((((252 * rho2 - 630) * rho2 + 560) * rho2 - 210) * rho2 + 30) * rho2 -
     ↳1)
85                * 11 / M_PI);
86       libmesh_fallthrough();
87
88     case 9:
89       save(54, rho4 * rho4 * rho
90                * 20 / M_PI);
91       save(53, (9 * rho2 - 8) * rho4 * rho2 * rho
92                * 20 / M_PI);
93       save(52, ((36 * rho2 - 56) * rho2 + 21) * rho4 * rho
94                * 20 / M_PI);
95       save(51, (((84 * rho2 - 168) * rho2 + 105) * rho2 - 20) * rho2 * rho
96                * 20 / M_PI);
97       save(50, ((((126 * rho2 - 280) * rho2 + 210) * rho2 - 60) * rho2 + 5) * rho
98                * 20 / M_PI);
99       libmesh_fallthrough();
100
101    case 8:
102      save(44, rho4 * rho4
103               * 18 / M_PI);
104      save(43, (8 * rho2 - 7) * rho4 * rho2
105               * 18 / M_PI);
106      save(42, ((28 * rho2 - 42) * rho2 + 15) * rho4
107               * 18 / M_PI);
108      save(41, (((56 * rho2 - 105) * rho2 + 60) * rho2 - 10) * rho2
109               * 18 / M_PI);
110      save(40, ((((70 * rho2 - 140) * rho2 + 90) * rho2 - 20) * rho2 + 1)
111               * 9 / M_PI);
112      libmesh_fallthrough();
113
114    case 7:
115      save(35, rho4 * rho2 * rho
116               * 16 / M_PI);
```

```
117       save(34, (7 * rho2 - 6) * rho4 * rho
118                * 16 / M_PI);
119       save(33, ((21 * rho2 - 30) * rho2 + 10) * rho2 * rho
120                * 16 / M_PI);
121       save(32, (((35 * rho2 - 60) * rho2 + 30) * rho2 - 4) * rho
122                * 16 / M_PI);
123       libmesh_fallthrough();
124
125     case 6:
126       save(27, rho4 * rho2
127                * 14 / M_PI);
128       save(26, (6 * rho2 - 5) * rho4
129                * 14 / M_PI);
130       save(25, ((15 * rho2 - 20) * rho2 + 6) * rho2
131                * 14 / M_PI);
132       save(24, (((20 * rho2 - 30) * rho2 + 12) * rho2 - 1)
133                * 7 / M_PI);
134       libmesh_fallthrough();
135
136     case 5:
137       save(20, rho4 * rho
138                * 12 / M_PI);
139       save(19, (5 * rho2 - 4) * rho2 * rho
140                * 12 / M_PI);
141       save(18, ((10 * rho2 - 12) * rho2 + 3) * rho
142                * 12 / M_PI);
143       libmesh_fallthrough();
144
145     case 4:
146       save(14, rho4
147                * 10 / M_PI);
148       save(13, (4 * rho2 - 3) * rho2
149                * 10 / M_PI);
150       save(12, ((6 * rho2 - 6) * rho2 + 1)
151                * 5 / M_PI);
152       libmesh_fallthrough();
153
154     case 3:
155       save(9, rho2 * rho
156               * 8 / M_PI);
157       save(8, (3 * rho2 - 2) * rho
158               * 8 / M_PI);
159       libmesh_fallthrough();
160
161     case 2:
162       save(5, rho2
163               * 6 / M_PI);
164       save(4, (2 * rho2 - 1)
165               * 3 / M_PI);
166       libmesh_fallthrough();
167
168     case 1:
169       save(2, rho
170               * 4 / M_PI);
171       libmesh_fallthrough();
172
173     case 0:
174       save(0, 1
175               * 1 / M_PI);
176   }
177
178   for (n = MAX_DIRECT_CALCULATION_ZERNIKE + 1; n <= _orders[0]; ++n)
179   {
180     j = simpleDoubleToSingle(n, n);
181     save(j, pow(rho, n)
182             * (n + n + 2) / M_PI);
```

```
183
184     j--;
185     save(j, n * load(j + 1) - (n + 1) * load(j - (n + n)));
186
187     for (q = n; q >= 4; q -= 2)
188     {
189       H3 = (-4 * (q - 2) * (q - 3)) / ((n + q - 2) * (n - q + 4.0));
190       H2 = (H3 * (n + q) * (n - q + 2)) / (4.0 * (q - 1)) + (q - 2);
191       H1 = q * (q - 1) / 2 - q * H2 + (H3 * (n + q + 2) * (n - q)) / 8.0;
192       j--;
193       if (q == 4)
194         save(j, (H1 * load(j + 2) + (H2 + H3 / rho2) * load(j + 1))
195                 * 0.5);
196       else
197         save(j, H1 * load(j + 2) + (H2 + H3 / rho2) * load(j + 1));
198     }
199   }
200
201   fillOutNegativeRankAndApplyAzimuthalComponent();
202 }
203 // clang-format on
204
205 void
206 Zernike::evaluateStandard()
207 {
208   std::size_t n;
209   long j, q;
210   Real H1, H2, H3;
211   const Real & rho = _standardized_location[0];
212   const Real rho2 = rho * rho;
213   const Real rho4 = rho2 * rho2;
214
215   if (MooseUtils::absoluteFuzzyLessEqual(rho, 0))
216   {
217     for (n = 0; n <= _orders[0]; n += 2)
218     {
219       j = simpleDoubleToSingle(n, 0);
220
221       if ((n / 2) % 2 != 0)
222         save(j, -1);
223       else
224         save(j, 1);
225     }
226
227     return;
228   }
229
230   switch (_orders[0])
231   {
232     default:
233     case MAX_DIRECT_CALCULATION_ZERNIKE: /* 10 */
234       save(65, rho4 * rho4 * rho2);
235       save(64, (10 * rho2 - 9) * rho4 * rho4);
236       save(63, ((45 * rho2 - 72) * rho2 + 28) * rho4 * rho2);
237       save(62, (((120 * rho2 - 252) * rho2 + 168) * rho2 - 35) * rho4);
238       save(61, ((((210 * rho2 - 504) * rho2 + 420) * rho2 - 140) * rho2 + 15) * rho2);
239       save(60, ((((252 * rho2 - 630) * rho2 + 560) * rho2 - 210) * rho2 + 30) * rho2 - 1)↲
  ↳;
240       libmesh_fallthrough();
241
242     case 9:
243       save(54, rho4 * rho4 * rho);
244       save(53, (9 * rho2 - 8) * rho4 * rho2 * rho);
245       save(52, ((36 * rho2 - 56) * rho2 + 21) * rho4 * rho);
246       save(51, (((84 * rho2 - 168) * rho2 + 105) * rho2 - 20) * rho2 * rho);
247       save(50, ((((126 * rho2 - 280) * rho2 + 210) * rho2 - 60) * rho2 + 5) * rho);
```

```
248       libmesh_fallthrough();
249
250     case 8:
251       save(44, rho4 * rho4);
252       save(43, (8 * rho2 - 7) * rho4 * rho2);
253       save(42, ((28 * rho2 - 42) * rho2 + 15) * rho4);
254       save(41, (((56 * rho2 - 105) * rho2 + 60) * rho2 - 10) * rho2);
255       save(40, (((70 * rho2 - 140) * rho2 + 90) * rho2 - 20) * rho2 + 1);
256       libmesh_fallthrough();
257
258     case 7:
259       save(35, rho4 * rho2 * rho);
260       save(34, (7 * rho2 - 6) * rho4 * rho);
261       save(33, ((21 * rho2 - 30) * rho2 + 10) * rho2 * rho);
262       save(32, (((35 * rho2 - 60) * rho2 + 30) * rho2 - 4) * rho);
263       libmesh_fallthrough();
264
265     case 6:
266       save(27, rho4 * rho2);
267       save(26, (6 * rho2 - 5) * rho4);
268       save(25, ((15 * rho2 - 20) * rho2 + 6) * rho2);
269       save(24, ((20 * rho2 - 30) * rho2 + 12) * rho2 - 1);
270       libmesh_fallthrough();
271
272     case 5:
273       save(20, rho4 * rho);
274       save(19, (5 * rho2 - 4) * rho2 * rho);
275       save(18, ((10 * rho2 - 12) * rho2 + 3) * rho);
276       libmesh_fallthrough();
277
278     case 4:
279       save(14, rho4);
280       save(13, (4 * rho2 - 3) * rho2);
281       save(12, (6 * rho2 - 6) * rho2 + 1);
282       libmesh_fallthrough();
283
284     case 3:
285       save(9, rho2 * rho);
286       save(8, (3 * rho2 - 2) * rho);
287       libmesh_fallthrough();
288
289     case 2:
290       save(5, rho2);
291       save(4, 2 * rho2 - 1);
292       libmesh_fallthrough();
293
294     case 1:
295       save(2, rho);
296       libmesh_fallthrough();
297
298     case 0:
299       save(0, 1);
300   }
301
302   for (n = MAX_DIRECT_CALCULATION_ZERNIKE + 1; n <= _orders[0]; ++n)
303   {
304     j = simpleDoubleToSingle(n, n);
305     save(j, pow(rho, n));
306
307     j--;
308     save(j, n * load(j + 1) - (n - 1) * load(j - (n + n)));
309
310     for (q = n; q >= 4; q -= 2)
311     {
312       H3 = (-4 * (q - 2) * (q - 3)) / ((n + q - 2) * (n - q + 4.0));
313       H2 = (H3 * (n + q) * (n - q + 2)) / (4.0 * (q - 1)) + (q - 2);
```

```
      H1 = q * (q - 1) / 2 - q * H2 + (H3 * (n + q + 2) * (n - q)) / 8.0;
      j--;
      save(j, H1 * load(j + 2) + (H2 + H3 / rho2) * load(j + 1));
    }
  }

  fillOutNegativeRankAndApplyAzimuthalComponent();
}

void
Zernike::fillOutNegativeRankAndApplyAzimuthalComponent()
{
  std::size_t n;
  long j, m, q, a;
  const Real & phi = _standardized_location[1];

  j = 0;
  for (n = 1; n <= _orders[0]; ++n)
  {
    j += n;
    for (m = 0, q = a = n; m < q; ++m, --q, a -= 2)
    {
      save(j + m, load(j + q) * sin(a * phi));
      save(j + q, load(j + q) * cos(a * phi));
    }
  }
}

const std::vector<Real> &
Zernike::getStandardizedFunctionLimits() const
{
  // Lazily instantiate the function limits array
  static const std::vector<Real> standardizedFunctionLimits = {0, 1, -M_PI, M_PI};

  return standardizedFunctionLimits;
}

Real
Zernike::getStandardizedFunctionVolume() const
{
  return M_PI; // The area of a unit disc is pi
}

std::vector<Real>
Zernike::getStandardizedLocation(const std::vector<Real> & location) const
{
  // Get the offset corresponding to the 'x' direction
  const Real offset1 = location[0] - _physical_bounds[0];
  // Get the offset corresponding to the 'y' direction
  const Real offset2 = location[1] - _physical_bounds[1];
  // Get the user-provided radius bound
  const Real & radius = _physical_bounds[2];
  // Covert to a radis and normalize
  const Real standardizedRadius = sqrt(offset1 * offset1 + offset2 * offset2) / radius;
  // Get the angle
  const Real theta = atan2(offset2, offset1);

  return {standardizedRadius, theta};
}

bool
Zernike::isInPhysicalBounds(const Point & point) const
{
  /*
   * Because Zernike polynomials live in RZ space, the easiest approach to check
   * this is to convert the physical location into a standardized location, then
```

```
380     * check against the radius and theta bounds.
381     */
382    const std::vector<Real> location = extractLocationFromPoint(point);
383    const std::vector<Real> standardized_location = getStandardizedLocation(location);
384
385    /*
386     * The radius (standardized_location[0]) is always positive, so only check
387     * against the maximum radius (1). The theta components should always be in
388     * bounds.
389     */
390    if (standardized_location[0] > 1.0)
391      return false;
392    else
393      return true;
394  }
395
396  std::size_t
397  Zernike::simpleDoubleToSingle(std::size_t n, long m) const
398  {
399    return (n * (n + 2) + m) / 2;
400  }
```

F.6 Transfers

File F.28 – A MOOSE transfer that operates by transferring FE coefficients between supported implementors of MutableCoefficientsInterface.

MultiAppFXTransfer.h 

```
1  //* This file is part of the MOOSE framework
2  //* https://www.mooseframework.org
3  //*
4  //* All rights reserved, see COPYRIGHT for full restrictions
5  //* https://github.com/idaholab/moose/blob/master/COPYRIGHT
6  //*
7  //* Licensed under LGPL 2.1, please see LICENSE for details
8  //* https://www.gnu.org/licenses/lgpl-2.1.html
9
10 #ifndef MULTIAPPMUTABLECOEFFICIENTSTRANSFER_H
11 #define MULTIAPPMUTABLECOEFFICIENTSTRANSFER_H
12
13 #include "MultiAppTransfer.h"
14
15 #include "MutableCoefficientsInterface.h"
16
17 class MultiAppFXTransfer;
18
19 template <>
20 InputParameters validParams<MultiAppFXTransfer>();
21
22 /**
23  * Transfers mutable coefficient arrays between supported object types
24  */
25 class MultiAppFXTransfer : public MultiAppTransfer
26 {
27 public:
28   MultiAppFXTransfer(const InputParameters & parameters);
29
30   // Overrides from MultiAppTransfer
31   virtual void execute() override;
32   virtual void initialSetup() override;
33
34 protected:
35   /// Name of the MutableCoefficientsInterface-derived object in the creating app
36   const std::string _this_app_object_name;
37
38   /// Name of the MutableCoefficientsInterface-derived object in the MultiApp
39   const std::string _multi_app_object_name;
40
41 private:
42   /**
43    * Gets a MutableCoefficientsInterface-based Function, intented for use via function pointer
44    */
45   MutableCoefficientsInterface & getMutableCoefficientsFunction(FEProblemBase & base,
46                                                                 const std::string &
   ↳object_name,
47                                                                 THREAD_ID thread);
48
49   /**
50    * Gets a MutableCoefficientsInterface-based UserObject, intended for use via function pointer
51    */
52   MutableCoefficientsInterface & getMutableCoefficientsUserOject(FEProblemBase & base,
53                                                                  const std::string &
   ↳object_name,
54                                                                  THREAD_ID thread);
55
56   /**
```

```
   * Function pointer typedef for functions used to find, convert, and return the appropriate
   * MutableCoefficientsInterface object from an FEProblemBase.
   */
  typedef MutableCoefficientsInterface & (MultiAppFXTransfer::*GetProblemObject)(
      FEProblemBase & base, const std::string & object_name, THREAD_ID thread);

  /**
   * Searches an FEProblemBase for a MutableCoefficientsInterface-based object and returns a
   * function pointer to the matched function type.
   */
  virtual GetProblemObject scanProblemBaseForObject(FEProblemBase & base,
                                                    const std::string & object_name,
                                                    const std::string & app_name);

  /// Function pointer for grabbing the MultiApp object
  GetProblemObject getMultiAppObject;

  /// Function pointer for grabbing the SubApp object
  GetProblemObject getSubAppObject;
};

#endif // MULTIAPPMUTABLECOEFFICIENTSTRANSFER_H
```

File F.29 – A MOOSE transfer that operates by transferring FE coefficients between supported implementors of MutableCoefficientsInterface.

MultiAppFXTransfer.C

```
//* This file is part of the MOOSE framework
//* https://www.mooseframework.org
//*
//* All rights reserved, see COPYRIGHT for full restrictions
//* https://github.com/idaholab/moose/blob/master/COPYRIGHT
//*
//* Licensed under LGPL 2.1, please see LICENSE for details
//* https://www.gnu.org/licenses/lgpl-2.1.html

#include "FEProblemBase.h"
#include "Function.h"
#include "MultiApp.h"
#include "UserObject.h"

#include "MultiAppFXTransfer.h"

template <>
InputParameters
validParams<MultiAppFXTransfer>()
{
  InputParameters params = validParams<MultiAppTransfer>();

  params.addClassDescription("Transfers coefficient arrays between objects that are derived from "
                             "MutableCoefficientsInterface; currently includes the following "
                             "types: FunctionSeries, FXBoundaryUserObject, and FXVolumeUserObject");

  params.addRequiredParam<std::string>(
      "this_app_object_name",
      "Name of the MutableCoefficientsInterface-derived object in this app (LocalApp).");

  params.addRequiredParam<std::string>(
      "multi_app_object_name",
      "Name of the MutableCoefficientsInterface-derived object in the MultiApp.");

```

```
35    return params;
36  }
37
38  MultiAppFXTransfer::MultiAppFXTransfer(const InputParameters & parameters)
39    : MultiAppTransfer(parameters),
40      _this_app_object_name(getParam<std::string>("this_app_object_name")),
41      _multi_app_object_name(getParam<std::string>("multi_app_object_name")),
42      getMultiAppObject(NULL),
43      getSubAppObject(NULL)
44  {
45  }
46
47  void
48  MultiAppFXTransfer::initialSetup()
49  {
50    // Search for the _this_app_object_name in the LocalApp
51    getMultiAppObject =
52        scanProblemBaseForObject(_multi_app->problemBase(), _this_app_object_name, "
    ↳MultiApp");
53    if (getMultiAppObject == NULL)
54      mooseError(
55          "Transfer '", name(), "': Cannot find object '", _multi_app_object_name, "' in
    ↳MultiApp");
56
57    // Search for the _multi_app_object_name in each of the MultiApps
58    for (std::size_t i = 0; i < _multi_app->numGlobalApps(); ++i)
59      if (_multi_app->hasLocalApp(i))
60      {
61        if (i == 0) // First time through, assign without checking against previous values
62          getSubAppObject = scanProblemBaseForObject(
63              _multi_app->appProblemBase(i), _multi_app_object_name, _multi_app->name());
64        else if (getSubAppObject != scanProblemBaseForObject(_multi_app->appProblemBase(i),
65                                                             _multi_app_object_name,
66                                                             _multi_app->name()))
67          mooseError("The name '",
68                     _multi_app_object_name,
69                     "' is assigned to two different object types. Please modify your input
    ↳ file and "
70                     "try again.");
71      }
72    if (getSubAppObject == NULL)
73      mooseError(
74          "Transfer '", name(), "': Cannot find object '", _multi_app_object_name, "' in
    ↳SubApp");
75  }
76
77  MultiAppFXTransfer::GetProblemObject
78  MultiAppFXTransfer::scanProblemBaseForObject(FEProblemBase & base,
79                                               const std::string & object_name,
80                                               const std::string & app_name)
81  {
82    /*
83     * For now we are only considering Functions and UserObjects, as they are the only types currently
84     * implemented with MutableCoefficientsInterface. Others may be added later.
85     *
86     * Functions:
87     *   FunctionSeries
88     *
89     * UserObjects:
90     *   FXBoundaryUserObject (via FXBaseUserObject)
91     *   FXVolumeUserObject (via FXBaseUserObject)
92     */
93    MutableCoefficientsInterface * interface;
94
95    // Check to see if the object with object_name is a Function
96    if (base.hasFunction(object_name))
```

```
97    {
98      Function & function = base.getFunction(object_name);
99      interface = dynamic_cast<MutableCoefficientsInterface *>(&function);
100
101     // Check to see if the function is a subclass of MutableCoefficientsInterface
102     if (interface)
103       return &MultiAppFXTransfer::getMutableCoefficientsFunction;
104     else
105       mooseError("Function '",
106                  object_name,
107                  "' in '",
108                  app_name,
109                  "' does not inherit from MutableCoefficientsInterface.",
110                  " Please change the function type and try again.");
111   }
112   // Check to see if the object with object_name is a UserObject
113   else if (base.hasUserObject(object_name))
114   {
115     // Get the non-const qualified UserObject, otherwise we would use getUserObject()
116     UserObject * user_object = base.getUserObjects().getActiveObject(object_name).get();
117     interface = dynamic_cast<MutableCoefficientsInterface *>(user_object);
118
119     // Check to see if the userObject is a subclass of MutableCoefficientsInterface
120     if (interface)
121       return &MultiAppFXTransfer::getMutableCoefficientsUserOject;
122     else
123       mooseError("UserObject '",
124                  object_name,
125                  "' in '",
126                  app_name,
127                  "' does not inherit from MutableCoefficientsInterface.",
128                  " Please change the function type and try again.");
129   }
130
131   return NULL;
132 }
133
134 MutableCoefficientsInterface &
135 MultiAppFXTransfer::getMutableCoefficientsFunction(FEProblemBase & base,
136                                                    const std::string & object_name,
137                                                    THREAD_ID thread)
138 {
139   return dynamic_cast<MutableCoefficientsInterface &>(base.getFunction(object_name,
    ↳thread));
140 }
141
142 MutableCoefficientsInterface &
143 MultiAppFXTransfer::getMutableCoefficientsUserOject(FEProblemBase & base,
144                                                     const std::string & object_name,
145                                                     THREAD_ID thread)
146 {
147   // Get the non-const qualified UserObject, otherwise we would use getUserObject()
148   UserObject * user_object = base.getUserObjects().getActiveObject(object_name, thread).
    ↳get();
149
150   return dynamic_cast<MutableCoefficientsInterface &>(*user_object);
151 }
152
153 void
154 MultiAppFXTransfer::execute()
155 {
156   _console << "Beginning MultiAppFXTransfer: " << name() << std::endl;
157
158   switch (_direction)
159   {
160     // LocalApp -> MultiApp
```

```
161     case TO_MULTIAPP:
162     {
163       // Get a reference to the object in the LocalApp
164       const MutableCoefficientsInterface & from_object =
165           (this->*getMultiAppObject)(_multi_app->problemBase(), _this_app_object_name, 0);
166
167       for (unsigned int i = 0; i < _multi_app->numGlobalApps(); ++i)
168       {
169         if (_multi_app->hasLocalApp(i))
170           for (THREAD_ID t = 0; t < libMesh::n_threads(); ++t)
171           {
172             // Get a reference to the object in each MultiApp
173             MutableCoefficientsInterface & to_object =
174                 (this->*getSubAppObject)(_multi_app->appProblemBase(i), _multi_app_object_name, t);
175
176             if (to_object.isCompatibleWith(from_object))
177               to_object.importCoefficients(from_object);
178             else
179               mooseError("'",
180                          _multi_app_object_name,
181                          "' is not compatible with '",
182                          _this_app_object_name,
183                          "'");
184           }
185       }
186       break;
187     }
188
189     // MultiApp -> LocalApp
190     case FROM_MULTIAPP:
191     {
192       /*
193       * For now we will assume that the transfers are 1:1 and the coefficients are synchronized
194       * among all instances, thus we only need to grab the set of coefficients from the first
195       * SubApp.
196       */
197       if (_multi_app->hasLocalApp(0))
198       {
199         // Get a reference to the first thread object in the first MultiApp
200         const MutableCoefficientsInterface & from_object =
201             (this->*getSubAppObject)(_multi_app->appProblemBase(0), _multi_app_object_name, 0);
202
203         for (THREAD_ID t = 0; t < libMesh::n_threads(); ++t)
204         {
205           // Get a reference to the object in each LocalApp instance
206           MutableCoefficientsInterface & to_object =
207               (this->*getMultiAppObject)(_multi_app->problemBase(), _this_app_object_name, t);
208
209           if (to_object.isCompatibleWith(from_object))
210             to_object.importCoefficients(from_object);
211           else
212             mooseError("'",
213                        _multi_app_object_name,
214                        "' is not compatible with '",
215                        _this_app_object_name,
216                        "'");
217         }
218       }
219       break;
220     }
221   }
222
```

```
223   _console << "Finished MultiAppFXTransfer: " << name() << std::endl;
224 }
```

F.7 UserObjects

File F.30 – An abstract class that provides a few methods necessary to adapting MOOSE SideIntegralVariableUserObject for use with FXIntegralBaseUserObject. FXBoundaryBaseUserObject.h

```
1  //* This file is part of the MOOSE framework
2  //* https://www.mooseframework.org
3  //*
4  //* All rights reserved, see COPYRIGHT for full restrictions
5  //* https://github.com/idaholab/moose/blob/master/COPYRIGHT
6  //*
7  //* Licensed under LGPL 2.1, please see LICENSE for details
8  //* https://www.gnu.org/licenses/lgpl-2.1.html
9
10 #ifndef FXBOUNDARYBASEUSEROBJECT_H
11 #define FXBOUNDARYBASEUSEROBJECT_H
12
13 #include "SideIntegralVariableUserObject.h"
14
15 #include "FXIntegralBaseUserObject.h"
16
17 class FXBoundaryBaseUserObject;
18
19 template <>
20 InputParameters validParams<FXBoundaryBaseUserObject>();
21
22 /**
23  * This class provides the base for generating a functional expansion on a boundary by inheriting
24  * from FXIntegralBaseUserObject and providing SideIntegralVariableUserObject as the template
25  * parameter
26  */
27 class FXBoundaryBaseUserObject : public FXIntegralBaseUserObject<
      ↳SideIntegralVariableUserObject>
28 {
29 public:
30   FXBoundaryBaseUserObject(const InputParameters & parameters);
31
32 protected:
33   // Overrides from FXIntegralBaseUserObject
34   virtual Point getCentroid() const final;
35   virtual Real getVolume() const final;
36 };
37
38 #endif // FXBOUNDARYBASEUSEROBJECT_H
```

File F.31 – An abstract class that provides a few methods necessary to adapting MOOSE SideIntegralVariableUserObject for use with FXIntegralBaseUserObject. FXBoundaryBaseUserObject.C

```
1  //* This file is part of the MOOSE framework
2  //* https://www.mooseframework.org
3  //*
4  //* All rights reserved, see COPYRIGHT for full restrictions
5  //* https://github.com/idaholab/moose/blob/master/COPYRIGHT
6  //*
7  //* Licensed under LGPL 2.1, please see LICENSE for details
8  //* https://www.gnu.org/licenses/lgpl-2.1.html
9
10 #include "FXBoundaryBaseUserObject.h"
11
12 template <>
```

```
InputParameters
validParams<FXBoundaryBaseUserObject>()
{
  InputParameters params = validParams<SideIntegralVariableUserObject>();

  params += validParams<FXIntegralBaseUserObjectParameters>();

  return params;
}

FXBoundaryBaseUserObject::FXBoundaryBaseUserObject(const InputParameters & parameters)
  : FXIntegralBaseUserObject(parameters)
{
  mooseInfo("Using FXInterface-type UserObject '",
            name(),
            "'.\nNote: it is your responsibility to ensure that the dimensionality, order, and "
            "series parameters for FunctionSeries '",
            _function_series.name(),
            "' are sane.");
}

Point
FXBoundaryBaseUserObject::getCentroid() const
{
  return _current_side_elem->centroid();
}

Real
FXBoundaryBaseUserObject::getVolume() const
{
  return _current_side_volume;
}
```

File F.32 – A UserObject that generates an FE from the flux distribution at a boundary. FXBoundaryFluxUserObject.h

```
//* This file is part of the MOOSE framework
//* https://www.mooseframework.org
//*
//* All rights reserved, see COPYRIGHT for full restrictions
//* https://github.com/idaholab/moose/blob/master/COPYRIGHT
//*
//* Licensed under LGPL 2.1, please see LICENSE for details
//* https://www.gnu.org/licenses/lgpl-2.1.html

#ifndef FXBOUNDARYFLUXUSEROBJECT_H
#define FXBOUNDARYFLUXUSEROBJECT_H

#include "FXBoundaryBaseUserObject.h"

class FXBoundaryFluxUserObject;

template <>
InputParameters validParams<FXBoundaryFluxUserObject>();

/**
 * This boundary FX evaluator calculates the flux
 */
class FXBoundaryFluxUserObject final : public FXBoundaryBaseUserObject
{
public:
  FXBoundaryFluxUserObject(const InputParameters & parameters);

```

```
protected:
  // Override from SideIntegralVariableUserObject
  virtual Real computeQpIntegral() final;

  /// Name of the diffusivity property in the local material
  const std::string _diffusivity_name;

  /// Value of the diffusivity
  const MaterialProperty<Real> & _diffusivity;
};

#endif // FXBOUNDARYFLUXUSEROBJECT_H
```

File F.33 – A UserObject that generates an FE from the flux distribution at a boundary. FXBoundaryFluxUserObject.C 

```
//* This file is part of the MOOSE framework
//* https://www.mooseframework.org
//*
//* All rights reserved, see COPYRIGHT for full restrictions
//* https://github.com/idaholab/moose/blob/master/COPYRIGHT
//*
//* Licensed under LGPL 2.1, please see LICENSE for details
//* https://www.gnu.org/licenses/lgpl-2.1.html

#include "FXBoundaryFluxUserObject.h"

template <>
InputParameters
validParams<FXBoundaryFluxUserObject>()
{
  InputParameters params = validParams<FXBoundaryBaseUserObject>();

  params.addClassDescription("Generates an Functional Expansion representation for a boundary flux "
                             "condition using a 'FunctionSeries'-type Function");

  params.addRequiredParam<std::string>("diffusivity",
                                       "The name of the material diffusivity "
                                       "property, or raw value, that will be used "
                                       "in the flux computation.");

  return params;
}

FXBoundaryFluxUserObject::FXBoundaryFluxUserObject(const InputParameters & parameters)
  : FXBoundaryBaseUserObject(parameters),
    _diffusivity_name(parameters.get<std::string>("diffusivity")),
    _diffusivity(getMaterialProperty<Real>(_diffusivity_name))
{
}

Real
FXBoundaryFluxUserObject::computeQpIntegral()
{
  return -_diffusivity[_qp] * _grad_u[_qp] * _normals[_qp];
}
```

File F.34 – A UserObject that generates an FE from the value distribution at a boundary. FXBoundaryValueUserObject.h 

```
//* This file is part of the MOOSE framework
//* https://www.mooseframework.org
```

```
//*
//* All rights reserved, see COPYRIGHT for full restrictions
//* https://github.com/idaholab/moose/blob/master/COPYRIGHT
//*
//* Licensed under LGPL 2.1, please see LICENSE for details
//* https://www.gnu.org/licenses/lgpl-2.1.html

#ifndef FXBOUNDARYVALUEUSEROBJECT_H
#define FXBOUNDARYVALUEUSEROBJECT_H

#include "FXBoundaryBaseUserObject.h"

class FXBoundaryValueUserObject;

template <>
InputParameters validParams<FXBoundaryValueUserObject>();

/**
 * This boundary FX evaluator calculates the values
 */
class FXBoundaryValueUserObject final : public FXBoundaryBaseUserObject
{
public:
  FXBoundaryValueUserObject(const InputParameters & parameters);
};

#endif // FXBOUNDARYVALUEUSEROBJECT_H
```

File F.35 – A UserObject that generates an FE from the value distribution at a boundary. FXBoundaryValueUserObject.C

```
//* This file is part of the MOOSE framework
//* https://www.mooseframework.org
//*
//* All rights reserved, see COPYRIGHT for full restrictions
//* https://github.com/idaholab/moose/blob/master/COPYRIGHT
//*
//* Licensed under LGPL 2.1, please see LICENSE for details
//* https://www.gnu.org/licenses/lgpl-2.1.html

#include "FXBoundaryValueUserObject.h"

template <>
InputParameters
validParams<FXBoundaryValueUserObject>()
{
  InputParameters params = validParams<FXBoundaryBaseUserObject>();

  params.addClassDescription("Generates an Functional Expansion representation for a boundary "
                             "value condition using a 'FunctionSeries'-type Function");

  return params;
}

FXBoundaryValueUserObject::FXBoundaryValueUserObject(const InputParameters & parameters)
  : FXBoundaryBaseUserObject(parameters)
{
}
```

File F.36 – A polymorphic UserObject that provides an abstracted approach to generating an FE from any integral performed by another UserObject. FXIntegralBaseUserObject.h

```
//* This file is part of the MOOSE framework
//* https://www.mooseframework.org
//*
//* All rights reserved, see COPYRIGHT for full restrictions
//* https://github.com/idaholab/moose/blob/master/COPYRIGHT
//*
//* Licensed under LGPL 2.1, please see LICENSE for details
//* https://www.gnu.org/licenses/lgpl-2.1.html

#ifndef FXINTEGRALBASEUSEROBJECT_H
#define FXINTEGRALBASEUSEROBJECT_H

#include "AuxiliarySystem.h"
#include "MooseError.h"
#include "MooseMesh.h"
#include "MooseVariable.h"
#include "UserObject.h"

#include "libmesh/quadrature.h"

#include "FunctionSeries.h"
#include "MutableCoefficientsInterface.h"

/**
 * Class declaration for parameters - we cannot use templated types in validParams<>()
 */
class FXIntegralBaseUserObjectParameters
{
  // Empty class, used only for parameters
};

template <>
InputParameters validParams<FXIntegralBaseUserObjectParameters>();

/**
 * This class interacts with a MooseApp through functional expansions. It is templated to allow the
 * inheritance of two dual classes that operate in a volume (FXVolumeUserObject) or on a boundary
 * (FXBoundaryFluxUserObject and FXBoundaryValueUserObject)
 *
 * It uses an instance of FunctionSeries to generate the orthonormal function series required to
 * generate the functional expansion coefficients.
 */
template <class IntegralBaseVariableUserObject>
class FXIntegralBaseUserObject : public IntegralBaseVariableUserObject,
                                 public MutableCoefficientsInterface
{
public:
  FXIntegralBaseUserObject(const InputParameters & parameters);

  /**
   * Return a reference to the underlying function series
   */
  const FunctionSeries & getFunctionSeries() const;

  // Override from <IntegralBaseVariableUserObject>
  virtual Real getValue() final;

  // Overrides from UserObject
  virtual void finalize() final;
  virtual void initialize() final;
  virtual Real spatialValue(const Point & location) const final;
  virtual void threadJoin(const UserObject & sibling) final;

protected:
  // Policy-based design requires us to specify which inherited members we are using
```

```
66    using IntegralBaseVariableUserObject::_JxW;
67    using IntegralBaseVariableUserObject::_communicator;
68    using IntegralBaseVariableUserObject::_console;
69    using IntegralBaseVariableUserObject::_coord;
70    using IntegralBaseVariableUserObject::_integral_value;
71    using IntegralBaseVariableUserObject::_q_point;
72    using IntegralBaseVariableUserObject::_qp;
73    using IntegralBaseVariableUserObject::_variable;
74    using IntegralBaseVariableUserObject::computeIntegral;
75    using IntegralBaseVariableUserObject::computeQpIntegral;
76    using IntegralBaseVariableUserObject::getFunction;
77    using IntegralBaseVariableUserObject::name;
78
79    // Override from <IntegralBaseVariableUserObject>
80    virtual Real computeIntegral() final;
81
82    /**
83     * Get the centroid of the evaluated unit
84     */
85    virtual Point getCentroid() const = 0;
86
87    /**
88     * Get the volume of the evaluated unit
89     */
90    virtual Real getVolume() const = 0;
91
92    /// History of the expansion coefficients for each solve
93    std::vector<std::vector<Real>> _coefficient_history;
94
95    /// Current coefficient partial sums
96    std::vector<Real> _coefficient_partials;
97
98    /// Reference to the underlying function series
99    FunctionSeries & _function_series;
100
101   /// Keep the expansion coefficients after each solve
102   const bool _keep_history;
103
104   /// Flag to prints the state of the zeroth instance in finalize()
105   const bool _print_state;
106
107   /// Volume of the standardized functional space of integration
108   const Real _standardized_function_volume;
109
110   /// Moose volume of evaluation
111   Real _volume;
112 };
113
114 template <class IntegralBaseVariableUserObject>
115 FXIntegralBaseUserObject<IntegralBaseVariableUserObject>::FXIntegralBaseUserObject(
116     const InputParameters & parameters)
117   : IntegralBaseVariableUserObject(parameters),
118     MutableCoefficientsInterface(this, this, parameters),
119     _function_series(FunctionSeries::checkAndConvertFunction(
120         getFunction("function"), UserObject::getParam<std::string>("_moose_base"), name()
    ↳)),
121     _keep_history(UserObject::getParam<bool>("keep_history")),
122     _print_state(UserObject::getParam<bool>("print_state")),
123     _standardized_function_volume(_function_series.getStandardizedFunctionVolume())
124 {
125   // Size the coefficient arrays
126   _coefficient_partials.resize(_function_series.getNumberOfTerms(), 0.0);
127   _coefficients.resize(_function_series.getNumberOfTerms(), 0.0);
128   _characteristics = _function_series.getOrders();
129
130   if (!_keep_history)
```

```
    _coefficient_history.resize(0);
}

template <class IntegralBaseVariableUserObject>
Real
FXIntegralBaseUserObject<IntegralBaseVariableUserObject>::computeIntegral()
{
  Real sum = 0.0;
  const Point centroid = getCentroid();

  // Check to see if this element/side is within the valid boundaries
  if (!_function_series.isInPhysicalBounds(centroid))
    return 0.0;

  // Loop over the quadrature points
  for (_qp = 0; _qp < _q_point.size(); ++_qp)
  {
    // Get the functional terms for a vectorized approach
    _function_series.setLocation(_q_point[_qp]);
    const std::vector<Real> & term_evaluations = _function_series.getOrthonormal();

    // Evaluate the functional expansion coefficients at each quadrature point
    const Real local_contribution = computeQpIntegral();
    const Real common_evaluation = local_contribution * _JxW[_qp] * _coord[_qp];
    for (std::size_t c = 0; c < _coefficient_partials.size(); ++c)
      _coefficient_partials[c] += term_evaluations[c] * common_evaluation;

    sum += local_contribution;
  }

  _volume += getVolume();

  return sum;
}

template <class IntegralBaseVariableUserObject>
void
FXIntegralBaseUserObject<IntegralBaseVariableUserObject>::finalize()
{
  // Sum the coefficient arrays over all processes
  _communicator.sum(_coefficient_partials);
  _communicator.sum(_volume);

  // Normalize the volume of the functional expansion to the FX standard space
  const Real volume_normalization = _standardized_function_volume / _volume;
  for (auto & partial : _coefficient_partials)
    partial *= volume_normalization;

  // We now have the completely evaluated coefficients
  _coefficients = _coefficient_partials;

  // The average value is the same as the zeroth coefficient
  _integral_value = _coefficient_partials[0];

  if (_keep_history)
    _coefficient_history.push_back(_coefficients);

  if (_print_state)
  {
    _function_series.setCoefficients(_coefficients);
    _console << COLOR_YELLOW << _function_series << COLOR_DEFAULT << std::endl;
  }
}

template <class IntegralBaseVariableUserObject>
const FunctionSeries &
```

```
FXIntegralBaseUserObject<IntegralBaseVariableUserObject>::getFunctionSeries() const
{
  return _function_series;
}

template <class IntegralBaseVariableUserObject>
Real
FXIntegralBaseUserObject<IntegralBaseVariableUserObject>::getValue()
{
  return _integral_value;
}

template <class IntegralBaseVariableUserObject>
void
FXIntegralBaseUserObject<IntegralBaseVariableUserObject>::initialize()
{
  IntegralBaseVariableUserObject::initialize();

  // Clear the partial sums
  for (auto & partial : _coefficient_partials)
    partial = 0;

  _volume = 0;
}

template <class IntegralBaseVariableUserObject>
void
FXIntegralBaseUserObject<IntegralBaseVariableUserObject>::threadJoin(const UserObject & s)
{
  const FXIntegralBaseUserObject<IntegralBaseVariableUserObject> & sibling =
      static_cast<const FXIntegralBaseUserObject<IntegralBaseVariableUserObject> &>(s);

  for (std::size_t c = 0; c < _coefficient_partials.size(); ++c)
    _coefficient_partials[c] += sibling._coefficient_partials[c];

  _volume += sibling._volume;
}

template <class IntegralBaseVariableUserObject>
Real
FXIntegralBaseUserObject<IntegralBaseVariableUserObject>::spatialValue(const Point & location) const
{
  _function_series.setLocation(location);

  return _function_series.expand(_coefficients);
}

#endif // FXINTEGRALBASEUSEROBJECT_H
```

File F.37 – A class that provides the input parameters needed by FXIntegralBaseUserObject.

FXIntegralBaseUserObjectParameters.C

```
//* This file is part of the MOOSE framework
//* https://www.mooseframework.org
//*
//* All rights reserved, see COPYRIGHT for full restrictions
//* https://github.com/idaholab/moose/blob/master/COPYRIGHT
//*
//* Licensed under LGPL 2.1, please see LICENSE for details
//* https://www.gnu.org/licenses/lgpl-2.1.html

```

```cpp
#include "FXIntegralBaseUserObject.h"

template <>
InputParameters
validParams<FXIntegralBaseUserObjectParameters>()
{
  InputParameters params = validParams<MutableCoefficientsInterface>();

  params.addClassDescription(
      "This UserObject interacts with a MooseApp through functional expansions.");

  params.addRequiredParam<FunctionName>("function",
                                        "The name of the FunctionSeries \"Function\" object with "
                                        "which to generate this functional expansion.");

  params.addParam<bool>(
      "keep_history", false, "Keep the expansion coefficients from previous solves");

  params.addParam<bool>("print_state", false, "Print the state of the zeroth instance each solve");

  return params;
}
```

File F.38 – A UserObject that generates an FE from the value distribution in a volume. FXVolumeUserObject.h

```cpp
//* This file is part of the MOOSE framework
//* https://www.mooseframework.org
//*
//* All rights reserved, see COPYRIGHT for full restrictions
//* https://github.com/idaholab/moose/blob/master/COPYRIGHT
//*
//* Licensed under LGPL 2.1, please see LICENSE for details
//* https://www.gnu.org/licenses/lgpl-2.1.html

#ifndef FXVOLUMEUSEROBJECT_H
#define FXVOLUMEUSEROBJECT_H

#include "ElementIntegralVariableUserObject.h"
#include "FXIntegralBaseUserObject.h"

class FXVolumeUserObject;

template <>
InputParameters validParams<FXVolumeUserObject>();

/**
 * This volumetric FX calculates the value
 */
class FXVolumeUserObject final : public FXIntegralBaseUserObject<ElementIntegralVariableUserObject>
{
public:
  FXVolumeUserObject(const InputParameters & parameters);

protected:
  // Overrides from FXIntegralBaseUserObject
  virtual Point getCentroid() const;
  virtual Real getVolume() const;
};

#endif // FXVOLUMEUSEROBJECT_H
```

File F.39 – A UserObject that generates an FE from the value distribution in a volume. FXVolumeUserObject.C

```
1  //* This file is part of the MOOSE framework
2  //* https://www.mooseframework.org
3  //*
4  //* All rights reserved, see COPYRIGHT for full restrictions
5  //* https://github.com/idaholab/moose/blob/master/COPYRIGHT
6  //*
7  //* Licensed under LGPL 2.1, please see LICENSE for details
8  //* https://www.gnu.org/licenses/lgpl-2.1.html
9
10 #include "FXVolumeUserObject.h"
11
12 template <>
13 InputParameters
14 validParams<FXVolumeUserObject>()
15 {
16   InputParameters params = validParams<ElementIntegralVariableUserObject>();
17
18   params += validParams<FXIntegralBaseUserObjectParameters>();
19
20   params.addClassDescription("Generates an Functional Expansion representation of a
     variable value "
21                              "over a volume using a 'FunctionSeries'-type Function");
22
23   return params;
24 }
25
26 FXVolumeUserObject::FXVolumeUserObject(const InputParameters & parameters)
27   : FXIntegralBaseUserObject<ElementIntegralVariableUserObject>(parameters)
28 {
29   mooseInfo("Using FXVolumeUserObject '",
30             name(),
31             "'.\nNote: it is your responsibility to ensure that the dimensionality, order
     , and "
32             "series parameters for FunctionSeries '",
33             _function_series.name(),
34             "' are sane.");
35 }
36
37 Point
38 FXVolumeUserObject::getCentroid() const
39 {
40   return _current_elem->centroid();
41 }
42
43 Real
44 FXVolumeUserObject::getVolume() const
45 {
46   return _current_elem_volume;
47 }
```

F.8 Utilities

File F.40 – A utility class that provides unique hashses for locations and times in a finite element mesh.

Hashing.h 

```
//* This file is part of the MOOSE framework
//* https://www.mooseframework.org
//*
//* All rights reserved, see COPYRIGHT for full restrictions
//* https://github.com/idaholab/moose/blob/master/COPYRIGHT
//*
//* Licensed under LGPL 2.1, please see LICENSE for details
//* https://www.gnu.org/licenses/lgpl-2.1.html

#ifndef HASHING_H
#define HASHING_H

#include "MooseTypes.h"

/**
 * This namespace provides efficient algorithms for quickly hashing different types for checking
 * identity with a very low collision probability.
 */
namespace hashing
{
typedef std::size_t HashValue;

/**
 * Final iteration of the variadic template with no additional arguments
 */
inline void
hashCombine(HashValue & /* seed */)
{
}

/**
 * Variadic template to hashing a combination with finite size
 *   see: https://stackoverflow.com/a/38140932
 */
template <class T, class... Rest>
inline void
hashCombine(HashValue & seed, const T & value, Rest... rest)
{
  std::hash<T> hasher;

  seed ^= hasher(value) + 0x9e3779b9 + (seed << 6) + (seed >> 2);
  hashCombine(seed, rest...);
}

/**
 * Hash function for sampling 10 points from a large container and hashing
 *   see: https://stackoverflow.com/a/37007715
 */
template <class Container>
HashValue
hashLargeContainer(Container const & container)
{
  std::size_t size = container.size();
  std::size_t stride = 1 + size / 10;
  HashValue seed = size;

  for (std::size_t i = 0; i < size; i += stride)
  {
    hashCombine(seed, container.data()[i]);
  }
```

```
61
62    return seed;
63  }
64
65  /**
66   * Hashing for Point
67   */
68  inline HashValue
69  hashCombine(const libMesh::Point & point)
70  {
71    // 'Magic seed' seed that provides entropy against the other hashCombine() seed
72    HashValue seed = 3;
73
74    hashCombine(seed, point(0), point(1), point(2));
75
76    return seed;
77  }
78
79  /**
80   * Hashing for Point and time, useful for Functions
81   */
82  inline HashValue
83  hashCombine(Real time, const libMesh::Point & point)
84  {
85    // 'Magic seed' seed that provides entropy against the other hashCombine() seed
86    HashValue seed = 42;
87
88    hashCombine(seed, time, point(0), point(1), point(2));
89
90    return seed;
91  }
92  }
93
94  #endif // HASHING_H
```

F.9 Unit Testing

File F.41 – Provides unit testing of the Cartesian function series. Cartesian.C

```
//* This file is part of the MOOSE framework
//* https://www.mooseframework.org
//*
//* All rights reserved, see COPYRIGHT for full restrictions
//* https://github.com/idaholab/moose/blob/master/COPYRIGHT
//*
//* Licensed under LGPL 2.1, please see LICENSE for details
//* https://www.gnu.org/licenses/lgpl-2.1.html

#include "gtest/gtest.h"

#include "Setup.h"

TEST(FunctionalExpansionsTest, legendreConstructor)
{
  const unsigned int order = 5;
  Legendre legendre({FunctionalBasisInterface::_domain_options = "x"}, {order});
  EXPECT_EQ(legendre.getOrder(0), order);
}

TEST(FunctionalExpansionsTest, legendreSeriesEvaluation)
{
  const unsigned int order = 15;
  Real location = -0.90922108754014;
  std::array<Real, order> truth = {{0.50000000000000,
                                    -1.36383163131021,
                                    1.85006119760378,
                                    -1.80341832197563,
                                    1.19175581122701,
                                    -0.11669847057321,
                                    -1.20462734483853,
                                    2.48341349094950,
                                    -3.41981864606651,
                                    3.76808851494207,
                                    -3.39261995754146,
                                    2.30300489952095,
                                    -0.66011244776270,
                                    -1.24901920248131,
                                    3.06342136027001}};
  Legendre legendre({FunctionalBasisInterface::_domain_options = "x"}, {order});

  legendre.setLocation(Point(location));
  auto & answer = legendre.getAllOrthonormal();
  for (std::size_t i = 0; i < order; ++i)
    EXPECT_NEAR(answer[i], truth[i], tol);
}

TEST(FunctionalExpansionsTest, CartesianConstructor)
{
  std::vector<MooseEnum> domains;
  std::vector<std::size_t> orders;
  std::vector<MooseEnum> series;

  domains.push_back(FunctionalBasisInterface::_domain_options = "x");
  orders = {19};
  series.push_back(single_series_types_1D = "Legendre");
  Cartesian legendreOne(domains, orders, series, name);
  EXPECT_EQ(legendreOne.getNumberOfTerms(), orders[0] + 1);

  domains.push_back(FunctionalBasisInterface::_domain_options = "y");
```

```
61    orders = {{13, 15}};
62    series.push_back(single_series_types_1D = "Legendre");
63    Cartesian legendreTwo(domains, orders, series, name);
64    EXPECT_EQ(legendreTwo.getNumberOfTerms(), (orders[0] + 1) * (orders[1] + 1));
65
66    domains.push_back(FunctionalBasisInterface::_domain_options = "z");
67    orders = {{14, 21, 22}};
68    series.push_back(single_series_types_1D = "Legendre");
69    Cartesian legendreThree(domains, orders, series, name);
70    EXPECT_EQ(legendreThree.getNumberOfTerms(), (orders[0] + 1) * (orders[1] + 1) * (orders
      [2] + 1));
71  }
72
73  TEST(FunctionalExpansionsTest, Cartesian3D)
74  {
75    const std::vector<MooseEnum> domains = {FunctionalBasisInterface::_domain_options = "x"
      ,
76                                            FunctionalBasisInterface::_domain_options = "y"
      ,
77                                            FunctionalBasisInterface::_domain_options = "z"
      };
78    const std::vector<std::size_t> orders = {14, 21, 22};
79    const std::vector<MooseEnum> series = {single_series_types_1D = "Legendre",
80                                           single_series_types_1D = "Legendre",
81                                           single_series_types_1D = "Legendre"};
82
83    Cartesian legendre3D(domains, orders, series, name);
84
85    const std::vector<Point> locations = {
86        Point(-0.14854612627465, 0.60364074055275, 0.76978431165674),
87        Point(0.93801805187856, 0.74175118177279, 0.74211345600994),
88        Point(0.35423736896098, -0.83921049062126, -0.02231845586669)};
89    const std::vector<Real> standard_truth = {1.32257143058688, 3.68047786932034,
      0.17515811557416};
90    const std::vector<Real> orthogonal_truth = {
91        -2.33043696271172, 74.48747654183713, -14.48091828923379};
92
93    for (std::size_t i = 0; i < locations.size(); ++i)
94    {
95      legendre3D.setLocation(locations[i]);
96      EXPECT_NEAR(legendre3D.getStandardSeriesSum(), standard_truth[i], tol);
97      EXPECT_NEAR(legendre3D.getOrthonormalSeriesSum(), orthogonal_truth[i], tol);
98    }
99  }
100
101 TEST(FunctionalExpansionsTest, functionalBasisInterfaceCartesian)
102 {
103   const std::vector<MooseEnum> domains = {FunctionalBasisInterface::_domain_options = "x"
      ,
104                                           FunctionalBasisInterface::_domain_options = "y"
      ,
105                                           FunctionalBasisInterface::_domain_options = "z"
      };
106   const std::vector<std::size_t> orders = {4, 5, 3};
107   const std::vector<MooseEnum> series = {single_series_types_1D = "Legendre",
108                                          single_series_types_1D = "Legendre",
109                                          single_series_types_1D = "Legendre"};
110
111   Cartesian legendre3D(domains, orders, series, name);
112
113   const Point location(-0.38541903411291, 0.61369802505416, -0.04539307255549);
114   const Real truth = 0.26458908225718;
115   FunctionalBasisInterface & interface = (FunctionalBasisInterface &)legendre3D;
116
117   interface.setLocation(location);
118   EXPECT_NEAR(interface.getStandardSeriesSum(), truth, tol);
```

```
119 }
```

File F.42 – Provides unit testing of the CylindricalDuo function series. CylindricalDuo.C

```
1  //* This file is part of the MOOSE framework
2  //* https://www.mooseframework.org
3  //*
4  //* All rights reserved, see COPYRIGHT for full restrictions
5  //* https://github.com/idaholab/moose/blob/master/COPYRIGHT
6  //*
7  //* Licensed under LGPL 2.1, please see LICENSE for details
8  //* https://www.gnu.org/licenses/lgpl-2.1.html
9
10 #include "gtest/gtest.h"
11
12 #include "Setup.h"
13
14 TEST(FunctionalExpansionsTest, zernikeConstructor)
15 {
16   const unsigned int order = 5;
17   Zernike zernike({FunctionalBasisInterface::_domain_options = "x",
18                    FunctionalBasisInterface::_domain_options = "y"},
19                   {order});
20   EXPECT_EQ(zernike.getOrder(0), order);
21 }
22
23 TEST(FunctionalExpansionsTest, zernikeSeriesEvaluationXY)
24 {
25   const unsigned int order = 4;
26   const Point location(-0.90922108754014, 0.262698547343, 0.156796889218);
27   std::vector<Real> truth;
28
29   Zernike zernike({FunctionalBasisInterface::_domain_options = "x",
30                    FunctionalBasisInterface::_domain_options = "y"},
31                   {order});
32   zernike.setLocation(location);
33   truth = {{0.318309886183791,
34             -0.300628811510287,
35             -1.117940202314423,
36             0.791881706198328,
37             0.623918544603900,
38             1.365900806059890,
39             -1.357069098439213,
40             -0.288633095470502,
41             -1.073332058640328,
42             -1.349767446988918,
43             1.887803726543957,
44             0.404825692079851,
45             0.223343150486739,
46             0.698275682841869,
47             1.080889714660983}};
48   auto & answer = zernike.getAllOrthonormal();
49   for (std::size_t i = 0; i < zernike.getNumberOfTerms(); ++i)
50     EXPECT_NEAR(answer[i], truth[i], tol);
51 }
52
53 TEST(FunctionalExpansionsTest, zernikeSeriesEvaluationXZ)
54 {
55   const unsigned int order = 4;
56   const Point location(-0.90922108754014, 0.262698547343, 0.156796889218);
57   std::vector<Real> truth;
58
59   Zernike zernike({FunctionalBasisInterface::_domain_options = "x",
60                    FunctionalBasisInterface::_domain_options = "z"},
```

```
61                  {order});
62   zernike.setLocation(location);
63   truth = {{0.318309886183791,
64             -0.180774038043348,
65             -1.143454732567233,
66             0.487041727962393,
67             0.623918544603900,
68             1.501849527692451,
69             -0.867504286868072,
70             -0.173560777222340,
71             -1.097828505968007,
72             -1.706149176114187,
73             1.276644449771070,
74             0.248985426801565,
75             0.223343150486739,
76             0.767775375651402,
77             1.761335979897610}};
78   auto & answer = zernike.getAllOrthonormal();
79   for (std::size_t i = 0; i < zernike.getNumberOfTerms(); ++i)
80     EXPECT_NEAR(answer[i], truth[i], tol);
81 }
82
83 TEST(FunctionalExpansionsTest, zernikeSeriesEvaluationYZ)
84 {
85   const unsigned int order = 4;
86   const Point location(-0.90922108754014, 0.262698547343, 0.156796889218);
87   std::vector<Real> truth;
88
89   Zernike zernike({FunctionalBasisInterface::_domain_options = "y",
90                    FunctionalBasisInterface::_domain_options = "z"},
91                   {order});
92   zernike.setLocation(location);
93   truth = {{0.318309886183791,
94             0.052230505695590,
95             0.330374978445085,
96             0.040657672622662,
97             -0.823129261009826,
98             0.125372638358686,
99             0.020923587239414,
100            -0.187295294511344,
101            -1.184703806003468,
102            0.041151106306071,
103            0.008896570968308,
104            -0.184582980412102,
105            0.978025517529447,
106            -0.569182979683797,
107            0.012274247968574}};
108  auto & answer = zernike.getAllOrthonormal();
109  for (std::size_t i = 0; i < zernike.getNumberOfTerms(); ++i)
110    EXPECT_NEAR(answer[i], truth[i], tol);
111 }
112
113 TEST(FunctionalExpansionsTest, cylindricalDuoConstructorAxialX)
114 {
115   const std::vector<MooseEnum> domains = {FunctionalBasisInterface::_domain_options = "x",
116                                           FunctionalBasisInterface::_domain_options = "y",
117                                           FunctionalBasisInterface::_domain_options = "z"};
118   const std::vector<std::size_t> orders = {5, 18};
119   const std::vector<MooseEnum> series = {single_series_types_1D = "Legendre",
120                                          single_series_types_2D = "Zernike"};
121
122   CylindricalDuo cylindrical(domains, orders, series, name);
123   EXPECT_EQ(cylindrical.getNumberOfTerms(),
```

```
124             (orders[0] + 1) * ((orders[1] + 1) * (orders[1] + 2)) / 2);
125 }
126
127 TEST(FunctionalExpansionsTest, cylindricalDuoConstructorAxialY)
128 {
129   const std::vector<MooseEnum> domains = {FunctionalBasisInterface::_domain_options = "y",
130                                           FunctionalBasisInterface::_domain_options = "x",
131                                           FunctionalBasisInterface::_domain_options = "z"};
132   const std::vector<std::size_t> orders = {23, 8};
133   const std::vector<MooseEnum> series = {single_series_types_1D = "Legendre",
134                                          single_series_types_2D = "Zernike"};
135
136   CylindricalDuo cylindrical(domains, orders, series, name);
137   EXPECT_EQ(cylindrical.getNumberOfTerms(),
138             (orders[0] + 1) * ((orders[1] + 1) * (orders[1] + 2)) / 2);
139 }
140
141 TEST(FunctionalExpansionsTest, cylindricalDuoConstructorAxialZ)
142 {
143   const std::vector<MooseEnum> domains = {FunctionalBasisInterface::_domain_options = "z",
144                                           FunctionalBasisInterface::_domain_options = "x",
145                                           FunctionalBasisInterface::_domain_options = "y"};
146   const std::vector<std::size_t> orders = {21, 23};
147   const std::vector<MooseEnum> series = {single_series_types_1D = "Legendre",
148                                          single_series_types_2D = "Zernike"};
149
150   CylindricalDuo cylindrical(domains, orders, series, name);
151   EXPECT_EQ(cylindrical.getNumberOfTerms(),
152             (orders[0] + 1) * ((orders[1] + 1) * (orders[1] + 2)) / 2);
153 }
154
155 TEST(FunctionalExpansionsTest, cylindricalDuoEvaluator)
156 {
157   const std::vector<MooseEnum> domains = {FunctionalBasisInterface::_domain_options = "x",
158                                           FunctionalBasisInterface::_domain_options = "y",
159                                           FunctionalBasisInterface::_domain_options = "z"};
160   const std::vector<std::size_t> orders = {15, 17};
161   const std::vector<MooseEnum> series = {single_series_types_1D = "Legendre",
162                                          single_series_types_2D = "Zernike"};
163
164   CylindricalDuo cylindrical(domains, orders, series, name);
165
166   const std::vector<Point> locations = {
167       Point(-0.14854612627465, 0.60364074055275, 0.76978431165674),
168       Point(0.93801805187856, 0.74175118177279, 0.45207131996044),
169       Point(0.35423736896098, -0.83921049062126, -0.02231845586669)};
170   const std::vector<Real> standard_truth = {
171       0.42889689399543629, 4.3724388003439207, 0.82275646257084989};
172   const std::vector<Real> orthogonal_truth = {
173       -10.386457517518826, -161.7959192066881, -3.9949571266605481};
174
175   for (std::size_t i = 0; i < locations.size(); ++i)
176   {
177     cylindrical.setLocation(locations[i]);
178     EXPECT_NEAR(cylindrical.getStandardSeriesSum(), standard_truth[i], tol);
179     EXPECT_NEAR(cylindrical.getOrthonormalSeriesSum(), orthogonal_truth[i], tol);
180   }
```

```
}

TEST(FunctionalExpansionsTest, functionalBasisInterfaceCylindrical)
{
  const std::vector<MooseEnum> domains = {FunctionalBasisInterface::_domain_options = "x",
    FunctionalBasisInterface::_domain_options = "y",
    FunctionalBasisInterface::_domain_options = "z"};
  const std::vector<std::size_t> orders = {4, 5};
  const std::vector<MooseEnum> series = {single_series_types_1D = "Legendre",
                                         single_series_types_2D = "Zernike"};

  CylindricalDuo cylindrical(domains, orders, series, name);

  const Point location(-0.38541903411291, 0.61369802505416, -0.04539307255549);
  const Real truth = 0.10414963426362565;
  FunctionalBasisInterface & interface = (FunctionalBasisInterface &)cylindrical;

  interface.setLocation(location);
  EXPECT_NEAR(interface.getStandardSeriesSum(), truth, tol);
}
```

File F.43 – Provides unit testing of the location hashing utilities. Hashing.C

```
//* This file is part of the MOOSE framework
//* https://www.mooseframework.org
//*
//* All rights reserved, see COPYRIGHT for full restrictions
//* https://github.com/idaholab/moose/blob/master/COPYRIGHT
//*
//* Licensed under LGPL 2.1, please see LICENSE for details
//* https://www.gnu.org/licenses/lgpl-2.1.html

#include "gtest/gtest.h"

#include "Hashing.h"

TEST(FunctionalExpansionsTest, hashPoint)
{
  const Point location(0.9780619323, 0.2136556650, 0.3509044429);
  const hashing::HashValue truth = 12573501117625172216u;

  EXPECT_EQ(hashing::hashCombine(location), truth);
}

TEST(FunctionalExpansionsTest, hashPointAndTime)
{
  const Point location(0.8392988091, 0.8482835642, 0.2509438471);
  const Real time = 0.5049258208;
  const hashing::HashValue truth = 5131894250494321433u;

  EXPECT_EQ(hashing::hashCombine(time, location), truth);
}

TEST(FunctionalExpansionsTest, hashIntVector)
{
  const std::vector<int> vector = {{351,  456, 49,   140,  491,  -482, 86,   314,  -442,
    154,
                                    430,  231, 47,   -261, -64,  -126, -187, 12,   61,
    -171,
                                    -145, 76,  -470, -100, 367,  1,    485,  458,  88,
    112,
```

```
36                                      -212, 357, -403, 467,  127,  138,  388,  -244, -479, -239,
37                                      -354, 475, -453, -36,  -365, -248, -95,  93,   -286, 436}};
38   const hashing::HashValue truth = 15665072587259105797u;
39
40   EXPECT_EQ(hashing::hashLargeContainer(vector), truth);
41 }
42
43 TEST(FunctionalExpansionsTest, hashRealVector)
44 {
45   const std::vector<Real> vector = {
46       {0.7436426667, 0.6962231856, 0.7257906803, 0.8084661009, 0.2137680124, 0.1470705959,
47        0.3657163957, 0.8818058481, 0.5513600342, 0.1984376524, 0.4276321100, 0.4177158632,
48        0.5599816732, 0.5976773302, 0.0534853375, 0.7162077056, 0.9344288478, 0.4947189992,
49        0.6393245755, 0.6877651005, 0.9567775877, 0.8757637166, 0.3850183877, 0.6800440879,
50        0.5090953855, 0.7779340857, 0.2310272569, 0.7807447395, 0.3012011716, 0.2879436719,
51        0.9785884888, 0.1201041026, 0.6422951422, 0.8657404100, 0.2686119524, 0.4199450789,
52        0.2437974613, 0.3544349330, 0.5725840559, 0.2903856081, 0.0055479019, 0.6819050123,
53        0.5512080507, 0.7301519914, 0.0077125671, 0.5284511770, 0.6894292950, 0.5014027958,
54        0.9773264137, 0.8477810277}};
55   const hashing::HashValue truth = 17839983326257054046u;
56
57   EXPECT_EQ(hashing::hashLargeContainer(vector), truth);
58 }
59
60 TEST(FunctionalExpansionsTest, hashVararg)
61 {
62   const hashing::HashValue truth = 11318853036716890990u;
63   hashing::HashValue value = 42;
64
65   hashing::hashCombine(value,
66                        0.9873791320,
67                        0.1953364838,
68                        0.8116485930,
69                        0.1863965161,
70                        0.5928596550,
71                        0.2295234343,
72                        0.6904344651,
73                        0.0045536257,
74                        0.1940171658,
75                        0.4950894997,
76                        0.8079496584,
77                        0.8060619760,
78                        0.7486861178,
79                        0.5493002792,
80                        0.1596405782,
81                        0.4023849890,
82                        0.2782852666,
83                        0.6461232825,
84                        0.1064983494,
85                        0.8130189389,
86                        0.5726072736,
87                        0.3327263263,
88                        0.1472734104,
89                        0.0234982033,
90                        0.6812964288,
91                        0.3276164827,
```

```
                          0.6911670346,
                          0.8299179444,
                          0.6484517577,
                          0.7986116002,
                          0.8813936466,
                          0.0049727250,
                          0.2010708901,
                          0.4933756641,
                          0.8354504016,
                          0.9452099799,
                          0.4643204087,
                          0.7382737011,
                          0.8045729776,
                          0.7870302766,
                          0.3384656050,
                          0.3401508297,
                          0.1595941894,
                          0.0673033342,
                          0.7483309385,
                          0.8940644866,
                          0.7948297883,
                          0.1890952442,
                          0.6151646001,
                          0.1672976625);

  EXPECT_EQ(value, truth);
}
```

File F.44 – Provides some common definitions used in the module's unit tests. Setup.h 

```
//* This file is part of the MOOSE framework
//* https://www.mooseframework.org
//*
//* All rights reserved, see COPYRIGHT for full restrictions
//* https://github.com/idaholab/moose/blob/master/COPYRIGHT
//*
//* Licensed under LGPL 2.1, please see LICENSE for details
//* https://www.gnu.org/licenses/lgpl-2.1.html

#include "Cartesian.h"
#include "CylindricalDuo.h"
#include "Legendre.h"
#include "Zernike.h"

#ifndef SETUP_H
#define SETUP_H

// Set the global tolerances
const double tol = 1e-13;

// Set the name
const std::string name = "UnitTesting";

// Recreate the MooseEnum types used in validParams<FunctionSeries>()
extern MooseEnum single_series_types_1D;
extern MooseEnum single_series_types_2D;

#endif // SETUP_H
```

References

[1] T. Goorley, M. James, T. Booth, F. Brown, J. Bull, J. Cox, and J. Durkee. Initial MCNP6 Release Overview. *Nuclear Technology*, 2012. 180(3):298–315. doi:10.13182/nt11-135

[2] J. Michopoulos, C. Farhat, and J. Fish. Modeling and Simulation of Multiphysics Systems. *Journal of Computing and Information Science in Engineering*, 2005. 5(3):198. doi:10.1115/1.2031269

[3] J. Ragusa and V. Mahadevan. Consistent and accurate schemes for coupled neutronics thermal-hydraulics reactor analysis. *Nuclear Engineering and Design*, 2009. 239(3):566–579. doi:10.1016/j.nucengdes.2008.11.006

[4] J. Hales, M. Tonks, F. Gleicher, B. Spencer, S. Novascone, R. Williamson, G. Pastore, and D. Perez. Advanced multiphysics coupling for LWR fuel performance analysis. *Annals of Nuclear Energy*, 2015. 84:98–110. doi:10.1016/j.anucene.2014.11.003

[5] J. Flusser, T. Suk, and B. Zitová. *2D & 3D image analysis by moments*. John Wiley & Sons, Inc., 2016

[6] E. Weisstein. Norm. `http://mathworld.wolfram.com/Norm.html`, 2017

[7] E. Kreyszig. *Advanced Engineering Mathematics*, chapter 11, pp. 500–501. John Wiley & Sons, Inc., 2011

[8] V. Mousseau, H. Zhang, and H. Zhao. Coupled High Fidelity Thermal Hydraulics and Neutronics for Reactor Safety Simulations. In *PHYSOR 2008 - International Conference on the Physics of Reactors "Nuclear Power: A Sustainable Resource"*. 2008 pp. 19–21

[9] A. Bhatia and E. Wolf. The Zernike Circle Polynomials Occurring in Diffraction Theory. *Proceedings of the Physical Society, Section B*, 1952. 65(11):909–910

[10] W. Stukeley. Memoirs of Sir Isaac Newton's Life, 1752. Transcript available online at: `https://goo.gl/PMGLMD`

[11] J. Deneubourg, S. Aron, S. Goss, and J. Pasteels. The Self-Organizing Exploratory Pattern of the Argentine Ant. *Journal of Insect Behavior*, 1990. 3(2):159–168. doi:10.1007/BF01417909

[12] D. Nigg, A. LaPorta, J. Nielsen, J. Parry, M. DeHart, S. Bays, and W. Skerjanc. A Complex-Geometry Validation Experiment for Advanced Neutron Transport Codes. *Transactions of the American Nuclear Society*, 2013. 109:1490–1493

[13] J. Oden, E. Lima, R. Almeida, Y. Feng, M. Rylander, D. Fuentes, D. Faghihi, M. Rahman, M. DeWitt, M. Gadde, and J. Zhou. Toward Predictive Multiscale Modeling of Vascular Tumor Growth. *Archives of Computational Methods in Engineering*, 2015. doi:10.1007/s11831-015-9156-x

[14] J. de Freitas, M. Niranjan, A. Gee, and A. Doucet. Sequential Monte Carlo Methods to Train Neural Network Models. *Neural Computation*, 2000. 12(4):955–993. doi:10.1162/089976600300015664

[15] M. Andersson, J. Yuan, and B. Sundén. Review on modeling development for multiscale chemical reactions coupled transport phenomena in solid oxide fuel cells. *Applied Energy*, 2010. 87(5):1461–1476. doi:10.1016/j.apenergy.2009.11.013

[16] S. Garain, H. Ghosh, and S. Chakrabarti. Effects of Compton Cooling on Outflow in a Two Component Accretion Flow around a Black Hole: Results of a Coupled Monte Carlo Total Variation Diminishing Simulation. *The Astrophysical Journal*, 2012. 758(2):114. doi:10.1088/0004-637X/758/2/114

[17] J. Gero. Design Prototypes: A Knowledge Representation Schema for Design. *AI Magazine*, 1990. 11(4):26. doi:10.1609/aimag.v11i4.854

[18] TOP500 News Team. Global Supercomputing Capacity Creeps Up as Petascale Systems Blanket Top 100. `https://goo.gl/FcZjva`, 2016

[19] Exascale Computing Project. `https://www.exascaleproject.org/`, 2017

[20] M. Asgari, T. Bahadir, D. Kropaczek, E. Gibson, and J. Williams. Analysis of the pin power peaking of the Hatch unit 1 cycle 21 failed fuel assemblies. In *TopFuel 2010 - LWR Fuel Performance*. Orlando, Florida, 2010 pp. 452–460

[21] M. Daeubler, A. Ivanov, B. Sjenitzer, V. Sanchez, R. Stieglitz, and R. Macian-Juan. High-fidelity coupled Monte Carlo neutron transport and thermal-hydraulic simulations using Serpent 2/SUBCHANFLOW. *Annals of Nuclear Energy*, 2015. 83(September 2015):352–375. doi:10.1016/j.anucene.2015.03.040

[22] D. Aumiller, F. Buschman, E. Tomlinson, and D. Gill. Development of an Integrated Code System Using R5EXEC and RELAP5-3D. *Nuclear Technology*, 2016. 193(1):183–199. doi:10.13182/NT15-5

[23] D. Gill, D. Aumiller, and D. Griesheimer. Monte Carlo and Thermal-Hydraulic Coupling Via PVMEXEC. In *PHYSOR 2014 – The Role of Reactor Physics Toward a Sustainable Future*. 2014

[24] V. Mahadevan, E. Merzari, T. Tautges, R. Jain, A. Obabko, M. Smith, and P. Fischer. High-resolution coupled physics solvers for analysing fine-scale nuclear reactor design problems. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, 2014. 372(2021). doi:10.1098/rsta.2013.0381

[25] S. Portegies Zwart, S. McMillan, S. Harfst, D. Groen, M. Fujii, B. Nualláin, E. Glebbeek, D. Heggie, J. Lombardi, P. Hut, V. Angelou, S. Banerjee, H. Belkus, T. Fragos, J. Fregeau, E. Gaburov, R. Izzard, M. Jurić, S. Justham, A. Sottoriva, P. Teuben, J. van Bever, O. Yaron, and M. Zemp. A multiphysics and multiscale software environment for modeling astrophysical systems. *New Astronomy*, 2009. 14(4):369–378. doi:10.1016/j.newast.2008.10.006

[26] T. Ikonen, V. Tulkki, E. Syrjälahti, V. Valtavirta, and J. Leppänen. FINIX – Fuel Behavior Model and Interface for Multiphysics Applications. In *TopFuel 2013 - LWR Fuel Performance*. Charlotte, NC, 2013 pp. 726–733

[27] D. Gaston, C. Permann, J. Peterson, A. Slaughter, D. Andrš, Y. Wang, M. Short, D. Perez, M. Tonks, J. Ortensi, L. Zou, and R. Martineau. Physics-based multiscale coupling for full core nuclear reactor simulation. *Annals of Nuclear Energy*, 2015. 84:45–54. doi:10.1016/j.anucene.2014.09.060

[28] C. Taut, C. Correa, O. Deutschmann, J. Warnatz, S. Einecke, C. Schulz, and J. Wolfrum. Three-dimensional modeling with Monte Carlo-probability density function methods and laser diagnostics of the combustion in a two-stroke engine. *Proceedings of the Combustion Institute*, 2000. 28(1):1153–1159. doi:10.1016/S0082-0784(00)80325-2

[29] K. Willis, S. Hagness, and I. Knezevic. Terahertz conductivity of doped silicon calculated using the ensemble Monte Carlo/finite-difference time-domain simulation technique. *Applied Physics Letters*, 2010. 96(6):062 106. doi:10.1063/1.3308491

[30] H. Wijesinghe, R. Hornung, A. Garcia, and N. Hadjiconstantinou. Three-dimensional Hybrid Continuum-Atomistic Simulations For Multiscale Hydrodynamics. *Journal of Fluids Engineering*, 2004. 126(5):768–777. doi:10.1115/1.1792275

[31] D. Molnar, R. Mukherjee, A. Choudhury, A. Mora, P. Binkele, M. Selzer, B. Nestler, and S. Schmauder. Multiscale simulations on the coarsening of Cu-rich precipitates in α-Fe using kinetic Monte Carlo, molecular dynamics and phase-field simulations. *Acta Materialia*, 2012. 60(20):6961–6971. doi:10.1016/j.actamat.2012.08.051

[32] A. Bogaerts, R. Gijbels, and W. Goedheer. Hybrid Modeling of a Capacitively Coupled Radio Frequency Glow Discharge in Argon: Combined Monte Carlo and Fluid Model. *Japanese Journal of Applied Physics, Part 1: Regular Papers and Short Notes and Review Papers*, 1999. 38(7B):4404–4415. doi:10.1143/JJAP.38.4404

[33] M. Combes, C. Grigné, L. Husson, C. Conrad, S. Le Yaouanq, M. Parenthoën, C. Tisseau, and J. Tisseau. Multiagent simulation of evolutive plate tectonics applied to the thermal evolution of the Earth. *Geochemistry Geophysics Geosystems*, 2012. 13(5):Q05 006. doi:10.1029/2011GC004014

[34] J. Lee, S. Haupt, and G. Young. Down-Selecting Numerical Weather Prediction Multi-Physics Ensembles with Hierarchical Cluster Analysis. *Journal of Climatology & Weather Forecasting*, 2016. 04(01). doi:10.4172/2332-2594.1000156

[35] Department of Energy and Climate Change (UK). My2050. `https://goo.gl/Spif6h`, 2011

[36] S. Bowman, L. Leal, O. Hermann, and C. Parks. ORIGIN-ARP, A Fast and Easy-to-Use Source Term Generation Tool. In *Ninth International Conference on Radiation Shielding*. 1999

[37] J. Leppänen, M. Pusa, T. Viitanen, V. Valtavirta, and T. Kaltiaisenaho. The Serpent Monte Carlo code: Status, development and applications in 2013. *Annals of Nuclear Energy*, 2015. 82:142–150. doi:10.1016/j.anucene.2014.08.024

[38] R. Busch and S. Bowman. *KENO V.a Primer: A Primer for Criticality Calculations with SCALE/KENO V.a Using CSPAN for Input*. Oak Ridge National Laboratory, 2003

[39] J. Leppänen. Development of a dynamic simulation mode in serpent 2 Monte Carlo code. In *International Conference on Mathematics and Computational Methods Applied to Nuclear Science and Engineering (M&C 2013)*. 2013

[40] T. Booth, F. Brown, J. Bull, L. Cox, J. Elson, J. Durkee, M. Fensin, R. Forster, J. Goorley, J. Hendricks, H. Hughes, M. James, R. Johns, B. Kiedrowski, G. McKinney, G. McMath, R. Martz, S. Mashnik, D. Pelowitz, R. Prael, J. Sweezy, and A. Zukaitis. *MCNP6 User's Manual*. Los Alamos National Laboratory, 2014

[41] AMTECK, Inc. ORTEC Applications Software. `https://goo.gl/ZTZGES`, 2016

[42] B. Wendt. *Development of an Automated Testing and Calibration Software Package for the NIFFTE TPC Project*. Masters thesis, Idaho State University, 2015

[43] D. Perez, R. Williamson, S. Novascone, R. Gardner, K. Gamble, A. Rice, G. Pastore, J. Hales, and B. Spencer. Assessment of BISON: A Nuclear Fuel Performance Analysis Code. Technical report, Idaho National Laboratory, 2013

[44] J. Leppänen. Performance of Woodcock delta-tracking in lattice physics applications using the Serpent Monte Carlo reactor physics burnup calculation code. *Annals of Nuclear Energy*, 2010. 37(5):715–722. doi:10.1016/j.anucene.2010.01.011

[45] Nuclear Fuel Cycle and Materials Section. Nuclear Fuel Cycle Simulation System (VISTA). techreport IAEA-TECDOC-1535, Internation Atomic Energy Agency, 2007

[46] SAPHIRE - Systems Analysis Programs for Hands-on Integrated Reliability Evaluations. `https://saphire.inl.gov/`, 2018

[47] Argonne National Laboratory. RESRAD Home Page. `https://goo.gl/GSxAON`, 2016

[48] M. Reginatto and P. Goldhagen. MAXED, A Computer Code For Maximum Entropy Deconvolution of Multisphere Neutron Spectrometer Data. *Health Physics*, 1999. 77(5)

[49] R. Salko and M. Avramova. *COBRA-TF Subchannel Thermal-Hydraulics Code (CTF) Theory Manual*. Pennsylvania State University, 2015

[50] H. Zhang, H. Zhao, L. Zou, D. Andrš, R. Berry, and R. Martineau. *RELAP-7 User's Guide*. Idaho National Laboratory, 2014

[51] N. Fil. Needs and Benefits for Industry in Applying Best-Estimate Analysis Methods. In *THICKET*. 2008

[52] R. Shapiro, I. Younker, and M. Fratoni. Neutronic performance of accident tolerant fuels. *Transactions of the American Nuclear Society*, 2013. 109:1351–1353

[53] H. Joo, J. Cho, K. Kim, M. Chang, B. Han, and C. Kim. Consistent Comparison of Monte Carlo and Whole-Core Transport Solutions for Cores with Thermal Feedback. In *PHYSOR 2004 - The Physics of Fuel Cycles and Advanced Nuclear Systems: Global Developments*. 2004

[54] Y. Xu, T. Downar, A. Ward, T. Kozlowski, and K. Ivanov. Multi-Physics Coupled Code Reactor Analysis with the U.S. NRC Code System TRACE/PARCS. In *PHYSOR 2006 - ANS Topical Meeting on Reactor Physics*. Vancouver, BC, Canada, 2006

[55] V. Seker, J. Thomas, and T. Downar. Reactor Simulation With Coupled Monte Carlo Calculation and Computational Fluid Dynamics. In *Joint International Topical Meeting on Mathematics & Computation and Supercomputing in Nuclear Applications (M&C + SNA 2007)*. Monterey, California, 2007

[56] V. Sanchez and A. Al-Hamry. Development of a Coupling Scheme Between MCNP and COBRA-TF for the Prediction of the Pin Power of a PWR Fuel Assembly. In *International Conference on Mathematics, Computational Methods & Reactor Physics (M&C 2009)*. Saratoga Springs, New York, 2009

[57] J. Cardoni and Rizwan-uddin. Nuclear Reactor Multi-Physics Simulations With Coupled MCNP5 and STAR-CCM+. In *International Conference on Mathematics and Computational Methods Applied to Nuclear Science and Engineering (M&C 2011)*. 2011

[58] D. Kotlyar, Y. Shaposhnik, E. Fridman, and E. Shwageraus. Coupled neutronic thermo-hydraulic analysis of full PWR core with Monte-Carlo based BGCore system. *Nuclear Engineering and Design*, 2011. 241(9):3777–3786. doi:10.1016/j.nucengdes.2011.07.028

[59] J. Leppänen, T. Viitanen, and V. Valtavirta. Multi-physics Coupling Scheme in the Serpent 2 Monte Carlo Code. In *Transactions of the American Nuclear Society*, volume 107. 2012 pp. 1165–1168

[60] M. Ellis, B. Forget, K. Smith, and D. Gaston. Preliminary Coupling of the Monte Carlo Code OpenMC and the Multiphysics Object-Oriented Simulation Environment (MOOSE) for Analyzing Doppler Feedback in Monte Carlo Simulations. In *ANS MC2015 - Joint International Conference on Mathematics and Computation (M&C), Supercomputing in Nuclear Applications (SNA) and the Monte Carlo (MC) Method*. 2015

[61] F. Gleicher, B. Spencer, S. Novascone, R. Williamson, R. Martineau, M. Rose, T. Downar, and B. Collins. Coupling the core analysis program DeCART to the fuel performance application BISON. In *International Conference on Mathematics and Computational Methods Applied to Nuclear Science and Engineering (M&C 2013)*. Sun Valley, Idaho, 2013 pp. 1234–1246

[62] D. Kotlyar and E. Shwageraus. Numerically stable Monte Carlo-burnup-thermal hydraulic coupling schemes. *Annals of Nuclear Energy*, 2014. 63:371–381. doi:10.1016/j.anucene.2013.08.016

[63] J. Ortensi, M. Dehart, F. Gleicher, Y. Wang, A. Alberti, and T. Palmer. Full Core TREAT Kinetics Demonstration Using Rattlesnake/BISON Coupling Within MAMMOTH. Technical report, Idaho National Laboratory, 2015

[64] H. Kleykamp. The chemical state of LWR high-power rods under irradiation. *Journal of Nuclear Materials*, 1979. 84(1-2):109–117. doi:10.1016/0022-3115(79)90154-5

[65] D. Griesheimer, D. Gill, J. Lane, and D. Aumiller. An integrated thermal hydraulic feedback method for Monte Carlo reactor calculations. In *PHYSOR 2008 - International Conference on the Physics of Reactors "Nuclear Power: A Sustainable Resource"*. 2008

[66] R. Williamson, J. Hales, S. Novascone, M. Tonks, D. Gaston, C. Permann, D. Andrš, and R. Martineau. Multidimensional multiphysics simulation of nuclear fuel behavior. *Journal of Nuclear Materials*, 2012. 423(1-3):149–163. doi:10.1016/j.jnucmat.2012.01.012

[67] F. Franceschini, A. Godfrey, S. Stimpson, T. Evans, B. Collins, J. Gehin, J. Turner, A. Graham, and T. Downar. Ap1000® PWR Startup Core Modeling and Simulation With VERA-CS. In *ANFM 2015—Advances in Nuclear Fuel Management V*. Hilton Head Island, South Carolina, 2015

[68] J. Hu and Rizwan-uddin. Coupled Neutronics and Thermal-Hydraulics Simulations Using MCNP and FLUENT. In *Transactions of the American Nuclear Society*. 2008 pp. 606–608

[69] K. Jareteg, P. Vinai, and C. Demazière. Fine-mesh deterministic modeling of PWR fuel assemblies: Proof-of-principle of coupled neutronic/thermal-hydraulic calculations. *Annals of Nuclear Energy*, 2014. 68:247–256. doi:10.1016/j.anucene.2013.12.019

[70] K. Ivanov and M. Avramova. Challenges in coupled thermal-hydraulics and neutronics simulations for LWR safety analysis. *Annals of Nuclear Energy*, 2007. 34(6):501–513. doi:10.1016/j.anucene.2007.02.016

[71] S. Hamilton and Kevin Clarno. Mathematical Framework for Coupling the AMP and Denovo Codes. *Transactions of the American Nuclear Society*, 2011. 105:515–517

[72] F. Gleicher, J. Ortensi, B. Spencer, Y. Wang, S. Novascone, J. Hales, D. Gaston, R. Williamson, and R. Martineau. The Coupling of the Neutron Transport Application Rattlesnake to the Nuclear Fuels Performance Application BISON under the MOOSE Framework. In *PHYSOR 2014 – The Role of Reactor Physics Toward a Sustainable Future*. 2014

[73] libMesh::Elem Class Reference. `http://goo.gl/ipSKgZ`, 2017. LibMesh v1.2.1

[74] D. Ziabletsev, M. Avramova, and K. Ivanov. Development of Pressurized Water Reactor Integrated Safety Analysis Methodology Using Multilevel Coupling Algorithm. *Nuclear Science and Engineering*, 2004. 148:414–425

[75] W. Chadsey, C. Wilson, and V. Pine. X-Ray Photoemission Calculations. *IEEE Transactions on Nuclear Science*, 1975. 22(6):2345–2350. doi:10.1109/TNS.1975.4328131

[76] D. Griesheimer. *Functional Expansion Tallies for Monte Carlo Simulations*. Ph.D. thesis, University of Michigan, 2005

[77] E. Weisstein. Orthogonal Polynomials. `https://goo.gl/k6TsV3`, 2016

[78] D. Légrády and J. Hoogenboom. Monte Carlo Midway Forward-Adjoint Coupling with Legendre Polynomials for Borehole Logging Applications. In *PHYSOR 2004 - The Physics of Fuel Cycles and Advanced Nuclear Systems: Global Developments*. 2004

[79] A. Pavlou and W. Ji. On-the-fly sampling of temperature-dependent thermal neutron scattering data for Monte Carlo simulations. *Annals of Nuclear Energy*, 2014. 71:411–426. doi:10.1016/j.anucene.2014.04.028

[80] G. Yesilyurt, W. Martin, and F. Brown. On-the-Fly Doppler Broadening for Monte Carlo Codes. *Nuclear Science and Engineering*, 2012. 171(3):239–257. doi:10.13182/NSE11-67

[81] J. Favorite and H. Lichtenstein. Exponential Monte Carlo convergence of a three-dimensional discrete ordinates solution. Technical report, Los Alamos National Laboratory, 1999

[82] D. Légrády and J. Hoogenboom. Visualization of space-dependency of responses of Monte Carlo calculations using Legendre polynomials. In *PHYSOR 2004 - The Physics of Fuel Cycles and Advanced Nuclear Systems: Global Developments*. 2004

[83] M. Ellis. *Methods for Including Multiphysics Feedback in Monte Carlo Reactor Physics Calculations*. dissertation, Massachusetts Institute of Technology, 2017

[84] N. Horelik, B. Herman, B. Forget, and K. Smith. Benchmark for evaluation and validation of reactor simulations (BEAVRS). In *International Conference on Mathematics and Computational Methods Applied to Nuclear Science and Engineering (M&C 2013)*. 2013

[85] Idaho National Laboratory. Light Water Reactor Sustainability Program. `https://goo.gl/qdFNvk`, 2016

[86] Argonne National Laboratory. NEAMS: The Nuclear Energy Advanced Modeling and Simulation Program. Technical Report ANL/NE-13/5, Department of Energy, 2013

[87] J. Hoogenboom, A. Ivanov, V. Sanchez, and C. Diop. A Flexible Coupling Scheme for Monte Carlo and Thermal-Hydraulics Codes. In *International Conference on Mathematics and Computational Methods Applied to Nuclear Science and Engineering (M&C 2011)*. 2011

[88] M. Tonks, P. Millett, P. Nerikar, S. Du, A. Andersson, C. Stanek, D. Gaston, D. Andrš, and R. Williamson. Multiscale development of a fission gas thermal conductivity model: Coupling atomic, meso and continuum level simulations. *Journal of Nuclear Materials*, 2013. 440(1-3):193–200. doi:10.1016/j.jnucmat.2013.05.008

[89] N. Brown, H. Ludewig, A. Aronson, G. Raitses, and M. Todosow. Neutronic evaluation of a PWR with fully ceramic microencapsulated fuel. Part I: Lattice benchmarking, cycle length, and reactivity coefficients. *Annals of Nuclear Energy*, 2013. 62:538–547. doi:10.1016/j.anucene.2013.05.025

[90] J. Carmack. Update on U.S. Accident Tolerant Fuel Program. Presentation, 2016

[91] Idaho National Laboratory. Advanced Test Reactor. `https://goo.gl/Do1dU2`, 2016

[92] Idaho National Laboratory. Transient Reactor Test Facility. `https://goo.gl/TKKBbe`, 2016

[93] M. DeHart, F. Gleicher, J. Ortensi, A. Alberti, and T. Palmer. Multi-Physics Simulation of TREAT Kinetics using MAMMOTH. *Transactions of the American Nuclear Society*, 2015. 113:1187–1190

[94] M. DeHart. Rattlesnake, MAMMOTH and Research in Support of TREAT Kinetics Calculations, 2016. From the DOE NEUP-IRP Meeting University of Michigan May 24, 2016

[95] S. Rice. Mathematical Analysis of Random Noise. In N. Wax, editor, *Selected Papers on Noise and Stochastic Processes*, pp. 133–294. Dover Publications, 1954

[96] A. Noel and H. Wio. A new series-expansion approach in Monte Carlo: Application to neutron shielding. *Annals of Nuclear Energy*, 1984. 11(5):225–227. doi:10.1016/0306-4549(84)90053-7

[97] M. Ismail and R. Zhang. A review of multivariate orthogonal polynomials. *Journal of the Egyptian Mathematical Society*, 2017. 25(2):91–110. doi:10.1016/j.joems.2016.11.001

[98] Wikipedia contributors. Frits Zernike. `https://goo.gl/eZXwZU`, 2017

[99] V. Lakshminarayanan and A. Fleck. Zernike polynomials: a guide. *Journal of Modern Optics*, 2011. 58(18):1678–1678. doi:10.1080/09500340.2011.633763

[100] C. Chong, P. Raveendran, and R. Mukundan. A comparative analysis of algorithms for fast computation of Zernike moments. *Pattern Recognition*, 2003. 36(3):731–742. doi:10.1016/S0031-3203(02)00091-2

[101] L. Broukhis, S. Cooper, and L. Noll. The International Obfuscated C Code Contest. `https://goo.gl/th8GMT`, 2016

[102] V. Hugo. *Les Miserables*. International Collector's Library, 1964

[103] Wikipedia contributors. Kinect. `https://goo.gl/VmVYj7`, 2017

[104] B. Wendt, L. Kerby, A. Tumulak, and J. Leppänen. Advancement of Functional Expansion Capabilities: Implementation and Optimization in Serpent 2. *Nuclear Engineering and Design*, 2018 (in review)

[105] Chinese Academy of Sciences. China to Jump Supercomputer Barrier. `https://goo.gl/aAieYo`, 2017

[106] M. Ellis, D. Gaston, B. Forget, and K. Smith. Preliminary Coupling of the Monte Carlo Code OpenMC and the Multiphysics Object-Oriented Simulation Environment for Analyzing Doppler Feedback in Monte Carlo Simulations. *Nuclear Science and Engineering*, 2017. 185(1):184–193. doi:10.13182/NSE16-26

[107] B. Wendt. Functional Expansion (FE) Algorithm Optimization - Benchmarking Data. 2018. doi:10.17632/3rw29g8j9g.2

[108] L. Kerby, M. Dehart, and A. Tumulak. Integration of OpenMC methods into MAMMOTH and Serpent. Technical Report INL/EXT-16-39874, Idaho National Laboratory, 2016

[109] B. Wendt, L. Kerby, A. Tumulak, J. Leppänen, and M. DeHart. Advancement of Functional Expansion Tallies Capabilities in Serpent 2. *Transactions of the American Nuclear Society*, 2017. 116:552–555

[110] United States Nuclear Regulatory Commission. Westinghouse AP1000 Design Control Document Rev. 19. `https://goo.gl/UcjxRR`, 2011

[111] J. Leppänen. *Development of a New Monte Carlo reactor physics code*. Ph.D. thesis, Helsinki University of Technology, 2007

[112] J. Leppänen. On the use of delta-tracking and the collision flux estimator in the Serpent 2 Monte Carlo particle transport code. *Annals of Nuclear Energy*, 2017. 105:161–167. doi:10.1016/j.anucene.2017.03.006

[113] L. Kerby, A. Tumulak, J. Leppänen, and V. Valtavirta. Preliminary Serpent–MOOSE Coupling and Implementation of Functional Expansion Tallies in Serpent. In *International Conference on Mathematics & Computational Methods Applied to Nuclear Science and Engineering (M&C 2017)*. 2017

[114] B. Wendt and L. Kerby. MultiApp Transfers in the MOOSE Framework based on Functional Expansions. *Transactions of the American Nuclear Society*, 2017

[115] B. Wendt and L. Kerby. Monte Carlo Simulation Tally Convergence: Time Comparisons between Functional Expansions and Meshes. In *ANS Annual Meeting*. 2018 (in review)

[116] International Criticality Safety Benchmark Evaluation Project (ICSBEP). `https://www.oecd-nea.org/science/wpncs/icsbep/`, 2017

[117] B. Wendt, A. Novak, L. Kerby, and P. Romano. Integration of Functional Expansion Methodologies as a MOOSE Module. In *PHYSOR 2018: Reactor Physics paving the way towards more efficient systems*. 2018

[118] D. Gaston, C. Newman, G. Hansen, and D. Lebrun-Grandié. MOOSE: A parallel computational framework for coupled systems of nonlinear equations. *Nuclear Engineering and Design*, 2009. 239:1768–1778. doi:10.1016/j.nucengdes.2009.05.021

[119] A. Novak, P. Romano, B. Wendt, R. Rahaman, E. Merzari, L. Kerby, C. Permann, R. Martineau, and R. Slaybaugh. PRELIMINARY COUPLING OF OpenMC AND Nek5000 WITHIN THE MOOSE FRAMEWORK. In *PHYSOR 2018: Reactor Physics paving the way towards more efficient systems*. 2018

[120] J. Mathews and R. Howell. The Schwarz-Christoffel Transformation. 2012