DEVELOPING FUEL MANAGEMENT CAPABILITIES BASED ON COUPLED MONTE CARLO DEPLETION IN SUPPORT OF THE MIT RESEARCH REACTOR (MITR) CONVERSION

by

PAUL K. ROMANO

B.S. Nuclear Engineering, 2007
B.S. Mathematics, 2007
Rensselaer Polytechnic Institute

Submitted to the Department of Nuclear Science and Engineering in Partial Fulfillment of the Requirements for the Degree of

Master of Science in Nuclear Science and Engineering
at the
Massachusetts Institute of Technology

May 2009

Signature of Author: _____
Department of Nuclear Science and Engineering
May 8, 2009

Certified by: _____
Benoit Forget, Ph.D.
Assistant Professor of Nuclear Science and Engineering
Thesis Supervisor

_____
Thomas H. Newton, Jr., Ph.D.
Associate Director, Engineering, Nuclear Reactor Laboratory
Thesis Reader

Accepted by: _____
Jacquelyn C. Yanch, Ph.D.
Professor of Nuclear Science and Engineering
Chair, Department Committee on Graduate Students

DEVELOPING FUEL MANAGEMENT CAPABILITIES BASED ON COUPLED MONTE CARLO DEPLETION IN SUPPORT OF THE MIT RESEARCH REACTOR (MITR) CONVERSION

by

PAUL K. ROMANO

Submitted to the Department of Nuclear Science and Engineering on
May 8, 2009 in Partial Fulfillment of the Requirements for the
Degree of Master of Science in Nuclear Science and Engineering

ABSTRACT

Pursuant to a 1986 NRC ruling, the MIT Reactor (MITR) is planning on converting from the use of highly enriched uranium (HEU) to low enriched uranium (LEU) for fuel. Prior studies have shown that the MITR will be able to operate using monolithic U-Mo LEU fuel while achieving neutron fluxes close to that of an HEU core. However, to date, detailed studies on fuel management and burnup while using LEU fuel have not been performed. In this work, a code package is developed for performing detailed fuel management studies at the MITR that is easy to use and is based on state-of-the-art computational methodologies.

A wrapper was written that enables fuel management operations to be modeled using MCODE, a code developed at MIT that couples MCNP to the point-depletion code ORIGEN. To explicitly model the movement of the control blades in the MITR as the core is being depleted, a criticality search algorithm was implemented to determine the critical position of the control blades at each depletion timestep. Additionally, a graphical user interface (GUI) was developed to automate the creation of model input files. The fuel management wrapper and GUI were developed in Python, with the PyQt4 extension being used for GUI-specific features.

The MCODE fuel management wrapper has been shown to perform reliably based on a number of studies. An LEU equilibrium core was modeled and burned for 640 days with the fuel being moved in the same pattern every 80 days. The control blade movement and nuclide concentrations were shown to be in agreement with what one would intuitively predict. The fuel management capabilities of REBUS-PC and the MCODE fuel management wrapper were compared by modeling the same refueling scheme using an HEU core. The element power peaking factors for the two models showed remarkable agreement.

Together, the fuel management wrapper and graphical user interface will help the staff at the MITR perform in-core fuel management calculations quickly and with a higher level of detail than that previously possible.

Thesis Supervisor: Benoit Forget
Title: Assistant Professor of Nuclear Science and Engineering

Acknowledgments

This research was sponsored by the Reduced Enrichment for Research and Test Reactors (RERTR) Program at Argonne National Laboratory and the U.S. NRC Graduate Education Fellowship.

I would like to express my most sincere gratitude to Tom Newton, Mujid Kazimi, and Ben Forget for their guidance, assistance, and patience throughout the course of my first two years at MIT but most of all their willingness to let me run with my crazy idea of doing fuel management using stochastic methods. The many hours they spent working with me to create a reliable code that will actually be used for future calculations will not be forgotten. I would also like to thank Ed Pilat and Pavel Hejzlar for their suggestions relating to my work and for the timely review of conference papers and presentations that they so willingly obliged to.

Also deserving acknowledgment is Nick Touran who originally inspired the idea for building a graphical user interface through his work on building an interface to automate calculations using REBUS and MC**2. His helpful advice on Python and GUI extensions was instrumental in choosing Python as the primary language for development.

I am indebted to Zhiwen Xu, the author of MCODE, for his prompt assistance whenever I ran into problems with MCODE. I would also like to thank Tim Trumbull who was very helpful in assisting with the development of the criticality search and for insights on the search capabilities implemented in MC21.

Without the love, encouragement, and (of course) financial support of my mom and dad, I would not be where I am today. As I have reached adulthood, they have become my best friends whom I know that I can always rely on when I need support, emotional or otherwise.

Lastly, I must thank my closest friend Rian Bahran. His companionship, collegiality, and motivation were paramount in helping me to develop into the person that I am today. The countless hours that we spent together day and night, be they in an office, on the court, in the gym, or in our apartment, helped me to keep my sanity and maintain a positive outlook on life.

Table of Contents

Acknowledgments **5**
Table of Contents **7**
List of Figures **10**
List of Tables **12**
1 Introduction **13**
1.1 Motivation 13
1.2 Objectives 13
1.3 Description of the MIT Reactor 14
1.3.1 History 14
1.3.2 MITR-II Description 14
1.3.3 Fuel Element Design 15
1.4 Neutronic and Burnup Modeling 17
1.4.1 Previous Models and Codes 17
1.4.2 MCNP Model 18
1.4.3 REBUS-PC Model 19
1.4.4 MCODE Depletion Code 22
1.4.5 Comparison of Execution Time 23
1.5 Using MCODE for Fuel Management 24
2 Interface Development **25**
2.1 Data Model and Abstraction 25
2.2 Graphical User Interface 26
2.2.1 Programming Language Choice 26
2.2.2 Description of Modules 28
2.2.3 Installing 28
2.2.4 Main Window 29
2.2.5 Creating/Editing Paths 30
2.2.6 Creating/Editing Fuel Elements 31
2.2.7 Creating/Editing Materials 32
2.2.8 Saving/Exporting a Run 33

2.3 Summary 34
3 Fuel Management Wrapper Development 35
3.1 Methodology 35
3.2 MCODE Modification 35
3.3 Wrapper Description 36
3.3.1 Language Choice 36
3.3.2 Description of Modules 36
3.3.3 Code Logic 37
3.4 Criticality Search 39
3.5 Summary 43
3.5.1 Input Requirements 43
4 Testing and Verification 45
4.1 MCODE Modification 45
4.2 LEU Equilibrium Core 47
4.2.1 Description 47
4.2.2 Control Blade Movement 48
4.2.3 Nuclide Concentrations 48
4.3 HEU “Equilibrium” Core 51
4.3.1 Description 51
4.3.2 Power Peaking 51
4.4 Summary 52
5 Conclusions and Future Work 54
5.1 Conclusions 54
5.2 Future Work 55
5.2.1 Code and Algorithmic Improvements 55
5.2.2 Graphical User Interface Improvements 55
5.2.3 Post-Processing Utilities 56
5.2.4 Optimizing Run Parameters 56
5.2.5 Validation against Experimental Data 56
5.2.6 Fuel Conversion Studies 57
Appendix A: Fuel Management Wrapper Source Code 58
A.1 mcodeFM.py 58
A.2 subs.py 59

A.3 fileIO.py 65
A.4 data/fuelLocations.py 66
A.5 data/classes.py 67
A.5 utils/keffsearch.py 69
Appendix B: Graphical User Interface Source Code 74
B.1 interface.pyw 74
B.2 gui/mainWindow.py 74
B.3 gui/listElementDialog.py 79
B.4 gui/listMaterialDialog.py 82
B.4 gui/editElementDialog.py 85
B.6 gui/editMaterialDialog.py 89
B.7 gui/editPathDialog.py 90
B.8 gui/editTimeDialog.py 94
B.9 gui/geometry.py 96
Appendix C: Example Files 98
C.1 Example of mcodeFM_input 98
C.2 Example of control_input 98
C.3 Example of mcnp.sh 98
References 99

List of Figures

Figure 1-1 Plan view of the MITR-II core. 15
Figure 1-2 Engineering drawing of MITR-II HEU fuel element cross-section [**11**] 16
Figure 1-3 Engineering drawing of MITR-II HEU fuel plate [**11**] 16
Figure 1-4 Engineering drawing of MITR-II HEU fuel element [**11**] 17
Figure 1-5 MCNP model of MITR-II core configuration #2 19
Figure 1-6 REBUS-PC model of the MITR-II core. 20
Figure 1-7 Element radial peaking factors in HEU core #2, **MCNP** and REBUS-PC results, from [**21**] 21
Figure 1-8 HEU core #2 integral control blade worth curves, adapted from [**21**] 21
Figure 2-1 UML class diagram for the data modeling of a burn-up run. 26
Figure 2-2 NSIS installer for the GUI 28
Figure 2-3 Main window for the graphical user interface 29
Figure 2-4 Dialog window editing cycle times 30
Figure 2-5 Dialog window for specifying fuel locations in a fuel path 31
Figure 2-6 Dialog window for specifying the radial/axial mesh and assigning materials 32
Figure 2-7 Dialog window displaying all fuel elements in the run 32
Figure 2-8 Dialog window for specifying a material composition 33
Figure 2-9 Dialog window displaying all materials in the run 33
Figure 3-1 New MCODE irradiation material specification 36
Figure 3-2 "Skeleton" MCNP model of the MITR with no fuel assembly detail 37
Figure 3-3 Logical flow chart for the fuel management wrapper 39
Figure 3-4 Logical flow chart for criticality search on control device 42
Figure 4-1 Pin-cell test model for MCODE change validation 45
Figure 4-2 Percent difference between k_{eff} and isotope concentrations for MCODE test cases 46
Figure 4-3 ^{235}U concentration for MCODE test cases 46
Figure 4-4 Path for each fuel element in the LEU equilibrium core run 47
Figure 4-5 Control blade height in LEU equilibrium core run 48
Figure 4-6 Relative ^{235}U concentration for LEU equilibrium core 49
Figure 4-7 Plate power peaking factors for LEU equilibrium core 49
Figure 4-8 ^{239}Pu concentration in atom/b-cm for LEU equilibrium core 50

Figure 4-9 Relative radiative capture reaction rate in ^{238}U for LEU equilibrium core 50
Figure 4-10 Element power peaking in HEU "equilibrium" core for REBUS and MCODE 52

List of Tables

Table 1-1 Chemical specification of uranium metal supplied to research reactors 17
Table 2-1 Comparison of programming languages..... 27
Table 3-1 Summary of input files for MCODE fuel management wrapper..... 44
Table 4-1 Material composition of LEU used in equilibrium core run..... 47
Table 4-2 Fuel movements for transition from fresh HEU core to equilibrium core..... 51

1 Introduction

1.1 Motivation

Historically, many research and test reactors across the world have been designed for use with highly enriched uranium (HEU) fuels because the relatively small size of such reactors makes it difficult to achieve useful neutron fluxes without a high enrichment. However, in light of U.S. non-proliferation policy aimed at minimizing the use of HEU in civilian applications, the U.S. Department of Energy initiated the Reduced Enrichment for Research and Test Reactors (RERTR) Program in 1978 with the intent of developing the technical means to convert research and test reactors fueled with (HEU) to low enriched uranium (LEU) [**1**]. The main functions of this program are the development of high-density LEU fuels and targets and assisting existing research reactors and isotope production facilities in the conversion process. Following the establishment of the RERTR program, the U.S. Nuclear Regulatory Commission (NRC) ruled in 1986 that all domestic research reactors would be required to convert to LEU fuel contingent upon the availability of Federal funds and fuel suitable for use in each particular reactor [**2**].

In 2004, the RERTR program was placed under the National Nuclear Security Administration as part of the Global Threat Reduction Initiative in light of renewed focus on removing proliferation-sensitive material in the civilian fuel cycle, and an aggressive goal of converting all domestic research reactors by September 2014 was established [**3**]. Since 1978, 11 currently operating U.S. research reactors have converted to LEU fuel under the purview of the RERTR program, and NNSA plans to convert another seven reactors by 2014. Two of these seven reactors are on schedule to be converted by September 2009 [**4**,**5**]. The remaining five reactors are unable to convert using currently qualified LEU fuels. Of these five reactors, two are university research reactors, one being the MIT reactor (MITR) and the other being the University of Missouri Research Reactor Center (MURR) at the University of Missouri-Columbia.

1.2 Objectives

Over the half-century that the MITR has been in operation, it has been fueled with HEU which enables it to achieve a high neutron flux that can be used as a source for in-core and ex-core experiments. As much of the operating revenue at the MITR is derived from the use of the neutron flux for experiments, maintaining current flux levels in the future is essential to the continued operation of the MITR.

Replacing the HEU fuel with currently qualified LEU fuel would impose serious technical challenges on the design and operation of the MITR. For one, it would be hard to achieve criticality with the current geometry, and even if criticality were to be obtained, the neutron flux to experiments would be significantly reduced. Thus, converting to LEU based on currently qualified fuels is not feasible as an option for conversion. As such, the MITR and four other reactors await the development of a high-density monolithic LEU fuel composed of an alloy of uranium and molybdenum. The high-density monolithic fuel being developed will have a density of 17 g/cm^3 [**6**]. To give some perspective, the highest density LEU fuel that is currently qualified for use by the NRC is 4.8 g/cm^3 [**7**]. The high-density fuel will allow

a large enough fuel loading for the compact MITR core to achieve criticality even with enrichments below 20%.

Prior studies have shown that the MITR will be able to operate using monolithic U-Mo LEU fuel while achieving neutron fluxes close to that of an HEU core [**8**]. However, to date, detailed studies on fuel management and burnup while using LEU fuel have not been performed. The objective of this thesis is to develop an interface for performing detailed fuel management studies at the MITR that is easy to use and is based on state-of-the-art computational methodologies. Such an interface and code package would represent a significant improvement over the decades-old fuel management practices currently being utilized at the MITR.

1.3 Description of the MIT Reactor

1.3.1 History

The MIT Reactor is one of the oldest university research reactors in the U.S., achieving initial criticality on July 21, 1958 [**9**]. The original version of the reactor (MITR-I) was heavy-water moderated and cooled and used an open array of plate-type fuel assemblies. After re-evaluating the research needs of the reactor and further optimization studies, the core was redesigned and the current reactor (MITR-II) was built and achieved criticality on August 14, 1975.

1.3.2 MITR-II Description

The MITR-II differs significantly from the original design. Rather than having an open array of fuel elements, the MITR-II uses a close-packed array of finned, plate-type assemblies cooled and moderated by light-water. The fuel assemblies (colloquially called fuel elements) have a rhombus-shaped cross-section. This allows the fuel elements to be arranged in three radial rings, one composed of three elements (A-ring), one composed of nine elements (B-ring), and one composed of 15 elements (C-ring). The geometric arrangement of the MITR-II core is shown in Figure 1-1. Typically, at least three fuel positions are occupied by either an in-core experimental facility or a solid aluminum “dummy” element that helps reduce power peaking. The remaining positions are filled with standard fuel elements. The reactor is currently licensed to operate at a steady-state power of 5 MW.

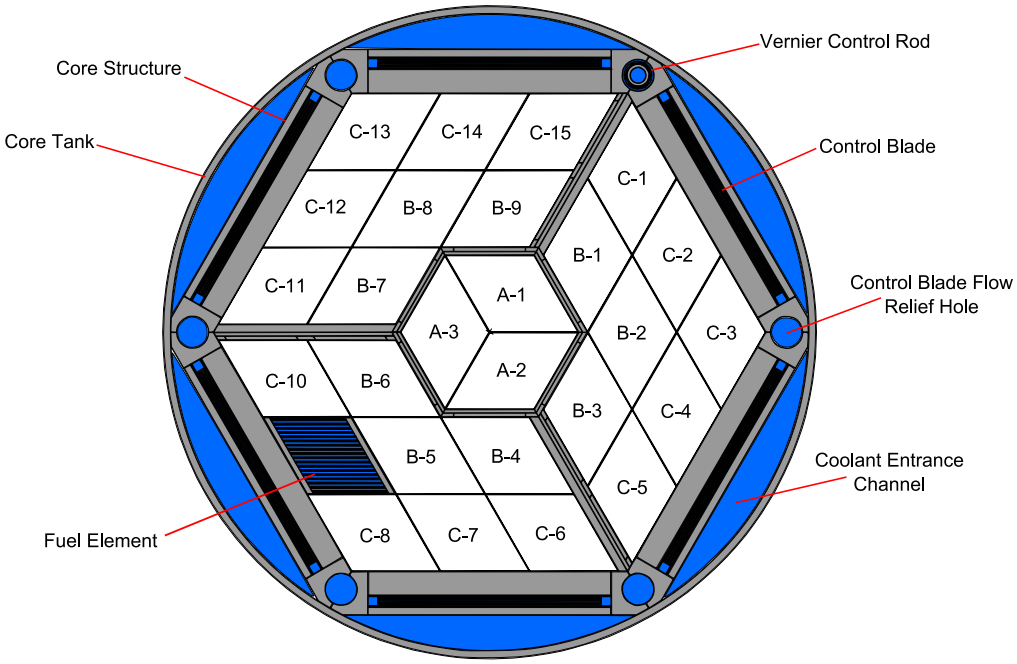

Figure 1-1 Plan view of the MITR-II core

The core is light-water moderated and cooled and is surrounded by a heavy water reflector. Beyond the heavy water reflector is a secondary reflector made of graphite. Boron impregnated stainless steel control blades are located at the periphery of the core on each of the sides of the hexagon. In addition to the control blades, a single cadmium regulating rod is also present at the periphery of the core.

In 1999, relicensing documents were submitted to the U.S. Nuclear Regulatory Commission which would increase the operational power to 6 MW [**10**]. This license application is referred to as MITR-III even though the core configuration and operation conditions are the same as for the MITR-II core. The power up-rate in this request is achieved by taking advantage of excess safety margin in the MITR-II design.

1.3.3 Fuel Element Design

The design of HEU and LEU fuel elements is different is several respects. The HEU fuel elements contain fifteen aluminum-clad fuel plates with 93% enriched uranium in an aluminide cermet matrix. An engineering drawing showing a horizontal cross-section of an HEU fuel element with dimensions is shown in Figure 1-2. The thickness of the fuel meat is 0.030 in (0.762 mm) with 0.015 in (0.381 mm) of aluminum cladding. In addition to the cladding, there are 0.010 in by 0.010 in (0.254 mm) longitudinal aluminum fins on the plates to enhance the heat transfer. An engineering drawing section showing the dimension for a fuel plate is shown in Figure 1-3.

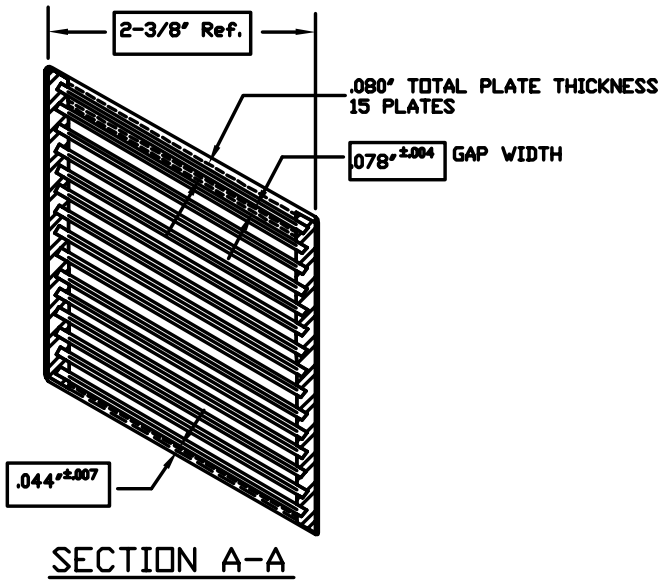

Figure 1-2 Engineering drawing of MITR-II HEU fuel element cross-section [**11**]

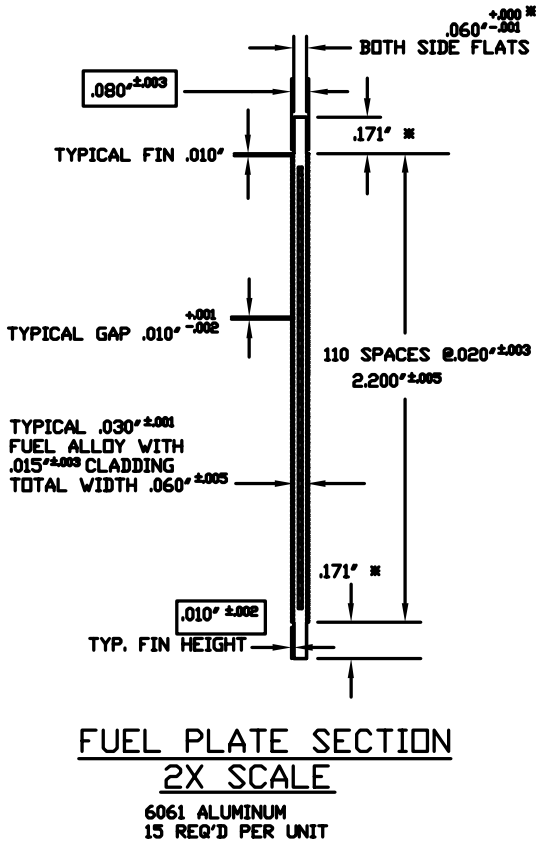

Figure 1-3 Engineering drawing of MITR-II HEU fuel plate [**11**]

The HEU fuel originally used in the MITR-II had a density of 3.4 g/cm^3 with a total loading of 445 g of ^{235}U in each fuel element [**8**]. After 1990, higher density fuel (3.7 g/cm^3) was used with total loading of 506 g of ^{235}U in each fuel element. The fuel meat is tapered at the ends of the fuel plates as shown in Figure 1-4.

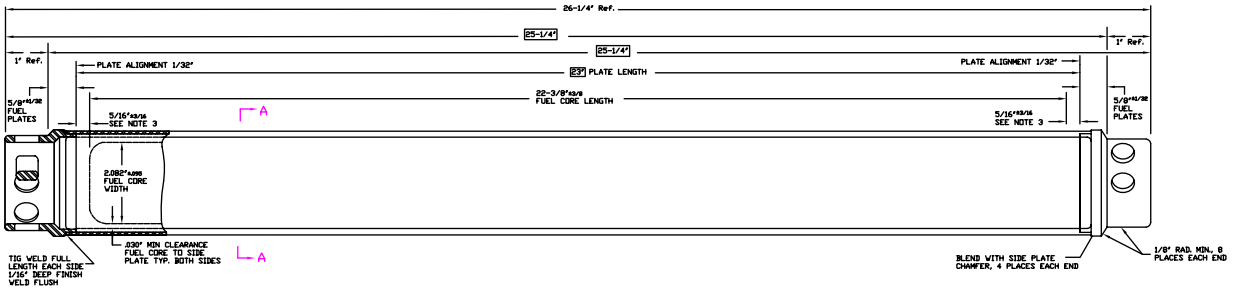

Figure 1-4 Engineering drawing of MITR-II HEU fuel element [**11**]

In contrast to the HEU fuel elements, the LEU fuel elements have eighteen aluminum-clad fuel plates with a monolithic uranium-molybdenum alloy. The choice of using 18 plates rather than 15 is the result of thermal-hydraulic optimization studies [**12**]. The fuel meat itself is 10 wt% molybdenum, and the uranium is enriched to 19.75%. Table 1-1 shows a summary of the chemical specification of the low-enriched uranium supplied to research reactors [**13**]. The U-10Mo alloy has a density of 17.02 g/cm^3 [**14**]. The vastly denser fuel results in a high enough fuel loading to reach criticality in the same configuration as an HEU core despite having a much lower enrichment.

Table 1-1 Chemical specification of uranium metal supplied to research reactors

Element	Concentration
^{232}U	≤ 0.002 μg/gU
^{234}U	≤ 0.260 wt%
^{235}U	19.75 ± 0.20 wt%
^{236}U	≤ 4600 Bq/gU
Activation Product	≤ 100.0 Bq/gU
Fission Products	≤ 600.0 Bq/gU
Total Impurities	≤ 1.2 mg/gU

The thickness of the fuel meat in the LEU fuel elements is 0.020 in (0.508 mm) with 0.010 in (0.381 mm) of aluminum cladding. Again, 0.010 in by 0.010 in (0.254 mm) longitudinal fins are employed to enhance heat transfer. Engineering drawings for an LEU fuel element have not been created yet since the design is still tentative.

1.4 Neutronic and Burnup Modeling

1.4.1 Previous Models and Codes

Over the 35 year history of the MITR-II, many neutronics models have been developed and used for core analysis, fuel management, and design studies. The first physics evaluations of the MITR-II design were made by Addae [**15**] in 1970 using the EXTERMINATOR-II and PDQ-7 diffusion theory codes. These

models were subsequently used by Kadak [**16**] to develop a fuel management strategy using the HARMONY option in PDQ.

During the mid 1970s, the focus of the experimental programs at the MITR-II shifted from use of the beam ports for neutrons to in-core irradiation facilities that would allow for a higher fluence in irradiated materials. Because Kadak's fuel management strategy had been predicated on the assumption that there would be only one in-core facility, the strategy was no longer applicable and the need for a new strategy emerged. In tandem with developing a new fuel management strategy, it was decided to develop new calculational methods as well. Bernard developed a fuel management code for the MITR-II using the CITATION finite-difference diffusion theory code [**17**]. The CITATION model for the MITR-II remains in use today for routine fuel management calculations.

1.4.2 MCNP Model

Most, if not all, modern Monte Carlo transport codes use combinational geometry to model complex arrangements of physical materials. The advantage of using combinational geometry over a discretized mesh as is common in deterministic transport codes is the ability for the user to easily model complex geometries and the lack of approximations (so long as all surfaces being modeled are second-order or less). In addition, Monte Carlo codes are able to utilize continuous-energy cross-sections, thus obviating the need to construct multi-group cross-sections. Given these advantages, Monte Carlo codes are considered the "gold standard" for performing reactor physics calculations. Even so, any user of such a code must be aware that stochastic uncertainties, poor coupling with depletion, and poor source convergence for problems with high dominance ratios may lead to erroneous results. Monte Carlo codes are a powerful tool for solving the transport equation, but only if used properly.

Using Monte Carlo methods in a primary design role in lieu of deterministic methods has become possible in recent years, especially for small cores such as the MITR. Recognizing this, an MCNP model of the MITR-II was constructed by Redmond, Yanch, and Harling [**18**] in the early 1990s. MCNP [**19**] is a generalized geometry, continuous-energy Monte Carlo transport code developed by Los Alamos National Laboratory and can simulate neutrons, photons, and electrons. The MCNP model was established for several configurations including three depleted cores with fuel data generated by Bernard's aforementioned fuel management code based on CITATION. These models were validated by comparing predictions for k_{eff} and the fast neutron flux in two in-core experimental facilities against experimental data. The MCNP model has been widely used and adapted for various neutronics studies at the MITR due to its accuracy and ease of use.

The MCNP model for the MITR-II is highly detailed and contains very few approximations. A cross-section of the MCNP model for MITR-II core configuration #2 with five aluminum dummy elements is shown in Figure 1-5. Each fuel plate is modeled discretely according to fuel specifications, and all reactor structures out to the outer edge of the graphite reflector are modeled. The fins on the fuel plates are not modeled explicitly as they are not neutronically important and are instead treated by extending the clad to preserve the mass of clad and water. The tapering on the ends of the fuel meat are not modeled either, but again, the active fuel length is set to ensure that the proper mass of fuel is present in each plate. For steady-state calculations, cross-sections are evaluated at 300 K and adjusted by using MCNP's built-in free-gas thermal treatment.

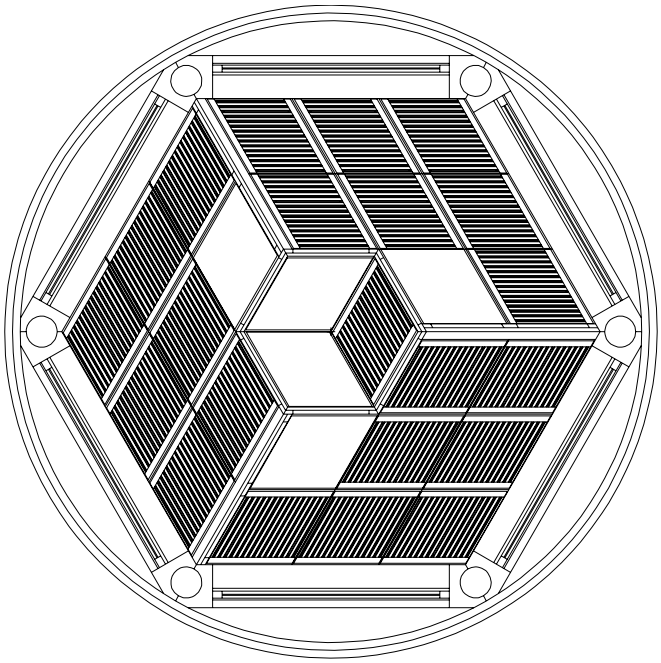

Figure 1-5 MCNP model of MITR-II core configuration #2

Notwithstanding the above considerations, the ability of the MCNP model to correctly predict reactor physics parameters is fundamentally limited not by the geometrical accuracy of the model but by our own limited knowledge of the material compositions, "as-built" dimensions due to manufacturing tolerances, and of course uncertainty in the evaluated cross-section data sets.

In order to convert from the current HEU core to an LEU core, it will be necessary to perform burnup modeling and fuel management calculations. However, MCNP provides only a snapshot of the physics in a system, i.e. material compositions are not affected by the nuclear reactions that take place. Thus, to analyze burnup in an MCNP model, the calculated fluxes and reaction rates must be passed to another code, e.g. ORIGEN2, that is capable of solving the Bateman equations for nuclear irradiation and decay. Several codes have been developed to do exactly that as will be discussed later. However, none of these MCNP-ORIGEN linkage codes have the capability to shuffle fuel. Thus, early in the LEU conversion fuel management studies, it was decided to create a model of the MITR in REBUS-PC, a fuel cycle analysis code based on diffusion theory, since it had the necessary fuel management capabilities.

1.4.3 REBUS-PC Model

REBUS-PC [**20**] is a system of codes based on DIF3D that is designed specifically for the analyses of research reactor fuel cycles. It is capable of solving for equilibrium conditions under a fixed fuel management scheme or explicit cycle-by-cycle operation under a specified fuel management program. The DIF3D neutronics processor within REBUS-PC is fully capable of solving problems using a triangular or hexagonal mesh.

In the REBUS-PC model of the MITR-II [**21**], shown in Figure 1-6, each fuel element is homogenized and consists of an 8 by 16 triangular mesh. As a result, the entire reactor core consists of a radial mesh of

542 by 312 triangles. The core is modeled using a triangular-Z mesh due to the rhomboid shape of the fuel elements in the MITR-II. Any circular boundaries such as the reactor tank are modeled by a jagged boundary based on mesh centroid radii. The WIMS-ANL 1D transport code [**22**] is used to generate 7-group neutron cross-section libraries. To model the reactor in WIMS-ANL, a series of 17 input files were created, each which accurately calculates the spectrum in a different physical region of the core.

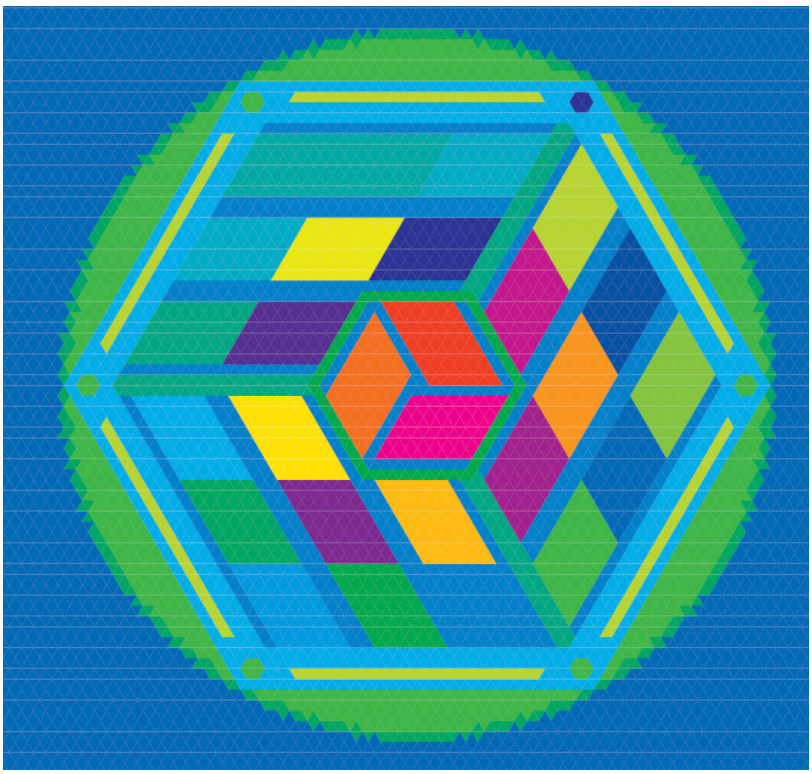

Figure 1-6 REBUS-PC model of the MITR-II core

Preliminary validation of the REBUS-PC model was very promising. For the MITR-II core, a comparison of the power profile and integral control blade worth curves calculated with REBUS-PC and MCNP showed very good agreement. Figure 1-7 shows the element peaking factors for the core #2 configuration calculated with REBUS-PC and MCNP. Figure 1-8 shows the integral control blade worth calculated with REBUS-PC and MCNP. This core configuration used five solid aluminum dummies as shown in Figure 1-7 with no fixed absorbers. It was operated at 2.5 MW for several months in 1976 [**8**].

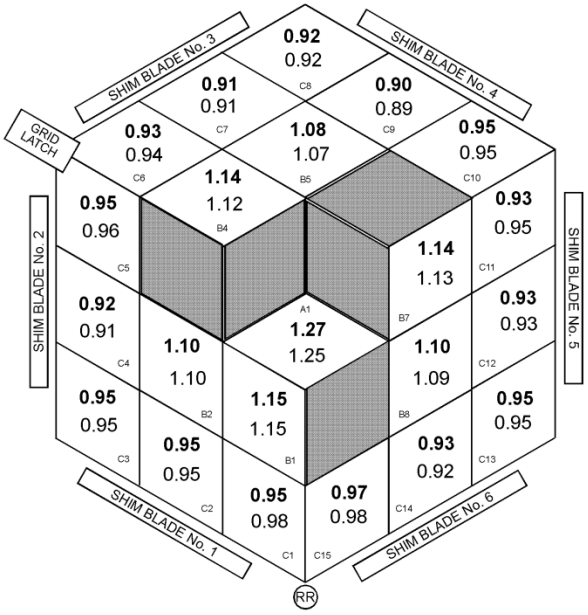

Figure 1-7 Element radial peaking factors in HEU core #2, **MCNP** and REBUS-PC results, from [**21**]

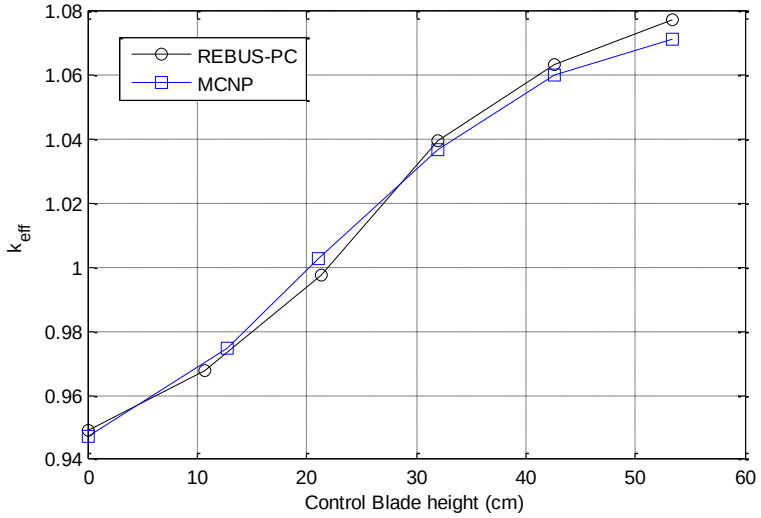

Figure 1-8 HEU core #2 integral control blade worth curves, adapted from [**21**]

Despite the success in modeling the MITR-II HEU core using REBUS-PC, the results for an LEU core have been rather discouraging. To date, the cross-sections generated by WIMS-ANL for an LEU core have not produced accurate results when used in REBUS-PC. After many struggles attempting to accurately model an LEU core in REBUS-PC, it was decided to pursue using MCNP for fuel management calculations with the added burden of developing the needed fuel management capabilities in tandem with the model development efforts on for REBUS-PC. The development of an interface that allows fuel management calculations to be performed using MCODE [**23**], a code developed at MIT that couples MCNP to the point-depletion code ORIGEN, is the focus of this thesis.

1.4.4 MCODE Depletion Code

Numerous codes have been developed to couple the continuous-energy Monte Carlo code MCNP to the point-depletion code ORIGEN2 [**24**] for performing burnup calculations. The first attempt to do this was a program called MOCUP (MCNP-ORIGEN Coupling Utility Program) [**25**] which was developed at Idaho National Engineering Laboratory (now Idaho National Laboratory). Shortly afterwards, Los Alamos National Laboratory also developed their own MCNP-ORIGEN coupling tool called MONTEBURNS [**26**]. While these are the two most well known MCNP-ORIGEN coupling codes, several others have been developed over time.

It is the opinion of many people that both MOCUP and MONTEBURNS are not user-friendly and require considerable effort to learn and use. Thus, to overcome the difficulties of using these codes, an effort to develop a new MCNP-ORIGEN coupling code at MIT started around 2002. The result of this effort was MCODE [**23**], an MCNP-ORIGEN Depletion Program, a tool for performing depletion calculations that focuses on functionality, versatility, and usability.

The current version, MCODE-2.2, is written entirely in ANSI C making it portable between different operating systems. Beyond being easier to use than MOCUP and MONTEBURNS, the depletion algorithm in MCODE is more accurate as well. To illustrate the differences in the algorithms, let us look at the transmutation equations. The rate at which material compositions change under irradiation is given by the following equation, also known as the Bateman equation [**23**]:

$$\frac{dX_{ij}}{dt} = \sum_{k=1}^{N_i} l_{k\to j}\lambda_k X_{ik} + \sum_{k=1}^{N_i} f_{k\to j} X_{ik} \int dE\ \sigma_k(E)\phi_i(E) - \lambda_j X_{ij} - X_{ij} \int dE\ \sigma_j(E)\phi_i(E) \tag{1.1}$$

where

X_{ij} = atom number density of nuclide j in material i;
N_i = number of nuclides in material i;
$l_{k\to j}$ = fraction of radioactive disintegrations by k that lead to formation of j;
λ_k = radioactive decay constant of nuclide k;
$f_{k\to j}$ = fraction of neutron absorptions by k that lead to formation of j;
$\phi_i(E)$ = spatial-average neutron energy spectrum in cell i;
$\sigma_k(E)$ = neutron absorption cross section of nuclide k.

Recognizing that Eq. (1.2) is a system of first-order linear differential equations, we can rewrite it in vector form:

$$\frac{d\boldsymbol{X}_i(t)}{dt} = \boldsymbol{A}_i(t)\boldsymbol{X}_i(t). \tag{1.2}$$

where the matrix $\boldsymbol{A}$ is called the transition matrix. If one assumes that the transition matrix is constant over time, the formal solution to Eq. (1.2) is:

$$\boldsymbol{X}_i(t) = e^{\boldsymbol{A}t}\boldsymbol{X}_i(0). \tag{1.3}$$

However, when a material is irradiated, the reaction rates do change with time since both the cross sections and fluxes are time-dependent. The total burnup time of a problem of interest is usually divided into time-steps over which the reaction rates are not expected to change appreciably so that the solution from Eq. (1.3) is valid. That being said, the manner in which the transition matrix is evaluated will have a considerable effect on how accurate the depletion algorithm is.

There are several methods for treating the transition matrix. The most obvious method is to simply take the reaction rates at the beginning of a time-step (t_{l-1}) and use them to predict the end-of-step (t_l) nuclides concentrations. This can be stated mathematically as:

$$\boldsymbol{X}_i(t_l) = e^{\boldsymbol{A}_i(t_{l-1})\cdot(t_l - t_{l-1})}\boldsymbol{X}_i(t_{l-1}). \tag{1.4}$$

This is the method that is used by MOCUP to predict end-of-step nuclide concentrations. While it is the simplest approach, it is also the most prone to error since it does not attempt to account for the fact that the reaction rates change over the time-step. A slightly better approach is taken by MONTEBURNS, which uses the middle-of-step reaction rates to predict the end-of-step nuclide concentrations, i.e.

$$\boldsymbol{X}_i(t_l) = \left\{\exp\left[\boldsymbol{A}_i\left(\frac{t_{l-1}+t_l}{2}\right)\cdot(t_l - t_{l-1})\right]\right\}\boldsymbol{X}_i(t_{l-1}). \tag{1.5}$$

MCODE-2.2 instead uses a predictor-corrector method as is used in CASMO-4 [**27**]. In this approach, two depletion calculations are performed. The end-of-step nuclide concentrations are calculated by taking the average of the compositions determined by depleting based on beginning-of-step reaction rates as well as predicted end-of-step reaction rates. This approach can be written as follows:

$$\begin{aligned}
\boldsymbol{X}_i^P(t_l) &= e^{\boldsymbol{A}_i(t_{l-1})\cdot(t_l - t_{l-1})}\boldsymbol{X}_i(t_{l-1}) \\
\boldsymbol{X}_i^C(t_l) &= e^{\boldsymbol{A}_i^P(t_l)\cdot(t_l - t_{l-1})}\boldsymbol{X}_i(t_{l-1}) \\
\boldsymbol{X}_i(t_l) &= \frac{\boldsymbol{X}_i^P(t_l) + \boldsymbol{X}_i^C(t_l)}{2}.
\end{aligned} \tag{1.6}$$

The predictor-corrector approach is a proven methodology that has been shown to produce more accurate results than the aforementioned approaches [**28**].

1.4.5 Comparison of Execution Time

As the speed of computers has continued to increase over time, the feasibility of using Monte Carlo methods is becoming more attractive thanks to their accuracy and ability to easily model complex geometries as discussed previously. That being said, the fact remains that simulations using Monte Carlo codes may take considerably longer than their deterministic counterparts, especially when determining local quantities such as power distributions. It is important to consider the time requirements for the various neutronic simulation options.

While Monte Carlo methods benefit from the use of combinatorial geometry, the very nature of simulating individual particles making random walks through the problem makes Monte Carlo codes very computationally expensive. As a result, Monte Carlo calculations are usually limited by how much raw processing power is available to the user. Solving the transport equation via deterministic methods is less

computationally expensive than doing so via Monte Carlo methods. However, deterministic methods may be limited by the memory of the computer depending on the size of the problem of interest. To solve very large problems, one might be forced to use domain decomposition to split the spatial domain of the problem into smaller pieces that can be stored on a single processing node.

By assuming a linearly anisotropic angular distribution of the neutron flux (i.e. diffusion approximation), one can solve neutron transport problems even faster than would be possible with higher-order deterministic transport methods. However, this comes at the expense of the accuracy of the solution obtained. For the MITR, solving a full-core depletion calculation using the diffusion theory code REBUS-PC will generally take on the order of 10 hours. With MCODE and the fuel management wrapper developed in this thesis, an equivalent calculation may take on the order of several days. That being said, MCNP can be run in parallel whereas REBUS-PC is currently only capable of serial calculations. Significant reductions in run times may be possible by running MCNP on many processors in parallel if the problem scales well. Furthermore, algorithmic improvements to MCODE may further reduce run times as discussed in section 5.2.1.

1.5 Using MCODE for Fuel Management

Despite the inherent advantages of using MCNP/MCODE over REBUS-PC for performing neutronics calculations, the fact remains that MCODE-2.2 does not have the ability to perform fuel management operations during irradiation and thus is of limited use for fuel management calculations. We are faced with two options as to how to deal with this shortcoming. The first option is to simply manipulate the input files ourselves at each point in time we desire to shuffle fuel assemblies and insert/remove fuel elements. However, when one considers that there are likely to be hundreds of materials being depleted separately in a single run, the task of performing detailed in-core fuel management calculations by doing the fuel management operations by hand begins to appear quite onerous and prone to human-error.

A better approach for performing fuel management operations is to develop software that automates the input file generation, data manipulation, and post-processing of the output data for analysis. To achieve this, two pieces of software have been developed. The first is a graphical user interface that serves as the front-end for allowing a user to define all the necessary data in a fuel management calculation. The second is a fuel management wrapper which reads the user data from the GUI, creates and runs an MCODE input file for each cycle, and handles the transfer of data between cycles.

2 Interface Development

2.1 Data Model and Abstraction

To meet the goal of an interface that would be applicable to more than one reactor type and to aid future developers in understanding the code, an object-oriented approach was taken whereby the data and its' associated procedures are abstracted. The data model consists of five main classes: *RunData*, *Path*, *Element*, *Material*, and *Location*.

The *RunData* class is the top-level class that holds all the data pertinent to a fuel-management run. This class stores the power level of the reactor used for calculating reaction rates, the power fraction at each cycle (effectively allowing the user to define a power history), and the times at which fuel management operations are to be performed.

An instance of the *Path* class contains information on an individual fuel assembly path, i.e. the physical path that a fuel assembly takes as it is depleted and moved through the reactor, and specifies which fuel element is to be used at the start of a path. Since the fuel elements in the MITR are commonly flipped and rotated to even out the burnup, this class also contains attributes which describe if the fuel element is to be flipped or rotated at any time during burnup.

An instance of the *Element* class defines a single fuel element, the radial and axial mesh that subdivides the element into separate depletion nodes, and the materials to be used for each depletion node. The number of plates in the *Element* can be specified to account for the fact that the HEU and LEU fuel elements have a different number of fuel plates.

The *Location* class describes a fixed physical location that a *Path* may occupy at any given time. For the purposes of rendering the shape of the fuel location in the GUI, the *Location* contains information on the points of a polygon that defines the shape of the fuel location as well as a translation and rotation. Being able to specify a translation and rotation for each fuel location is particularly useful since most reactors use fuel assemblies that are the same shape but are translated and rotated in different arrangements.

Lastly, the *Material* class describes a physical material and its constituents. The density and atom/weight fractions of the material follow MCNP conventions. If the density is specified as a positive real number, it is interpreted as the density in atom/barn-cm. If the density is specified as a negative real number, it is interpreted as the density in g/cm^3. To specify atom fractions for each individual nuclide, one would specify the fraction as a positive number whereas weight fractions would be specified with negative numbers.

We summarize the relations between the classes as follows. A fuel *Element* is assigned to beginning of each fuel *Path.* Each depletion node in an *Element* contains a *Material*. Additionally, a *Location* is specified for each cycle in the fuel *Path*. These associations are summarized in Figure 2-1, a Uniform Modeling Language (UML) class diagram.

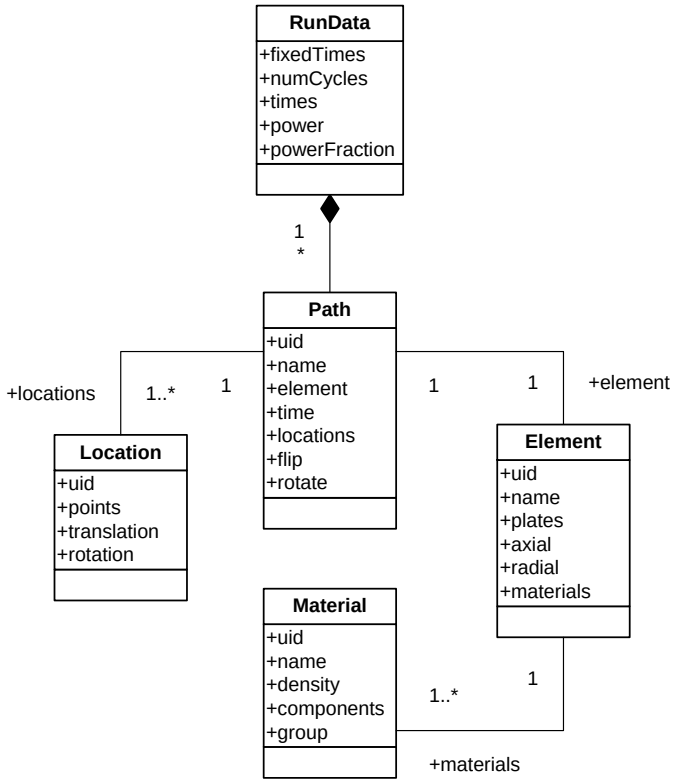

Figure 2-1 UML class diagram for the data modeling of a burn-up run

We note an important distinction between the classes used in the graphical user interface and those used for the fuel management wrapper. When the burnup run is performed using the fuel management wrapper, every depletion node of each fuel element has a unique material such that even fuel paths that have the same fuel element are still defined by distinct materials. Thus, there is no need for a separate *Element* class in the data model for the fuel management wrapper.

2.2 Graphical User Interface

2.2.1 Programming Language Choice

There are many programming languages available that are capable of building graphical user interfaces. These range from low-level, e.g. C, to the very-high-level, e.g. MATLAB. To complicate matters further, many popular GUI toolkits have bindings for several languages. That being said, each language considered should be evaluated based on the project's particular objectives. Like any project, this one had unique features that shaped the ultimate decision on what language and GUI toolkit to use. The factors considered when evaluating each language for its suitability were:

- Easy learning curve
- Code conciseness
- Availability of GUI libraries
- Code readability
- Free and open source (FOSS)

- Cross-platform portability
- Activity of development community

The languages considered for this project were C, C++, Java, Python, Perl, Ruby, and MATLAB. A comparison of these languages based on the objectives outlined above is shown in Table 2-1. The color coding of the cells signifies how well each languages meets the particular objective, green being the best and red being the worst.

Table 2-1 Comparison of programming languages

	Learning Curve	Code Conciseness	FOSS	GUI Libraries Available
C				Only APIs
C++				APIs, Many
Java				APIs, Many
Python				GTK, Qt, Tk, wxWidgets
Perl				GTK, Qt, Tk, wxWidgets
Ruby				GTK, Qt, Tk, wxWidgets
MATLAB				Native

While this particular comparison is rather subjective as it is the author's own evaluation, it is fairly obvious that the learning curve, code conciseness, and code readability will generally go hand-in-hand. C and C++ have the worst learning curve and code conciseness as a result of being low-level languages. Python, Perl, and Ruby are all high-level languages that are considered to be easy to learn and have very good code conciseness. C and C++ were ruled out as options based on the longer development times that are typical of a low-level, compiled, statically-typed language. The main reason to pursue a language such as C or C++ would be the vastly better execution speed over Java or any interpreted languages, but this was not a limiting constraint for this project.

Java is simpler than C or C++ due to features like garbage collection. However, it is still a statically-typed compiled language which leads to lower productivity. For this project where high productivity and rapid development were essential, we opted to rule out Java as an option in favor of a dynamically-typed interpreted language such as Python, Perl, or Ruby. MATLAB was considered for a brief time as it is very simple, has powerful array-processing capabilities, and has native support for building GUIs, but the fact that it is commercial software led us to rule it out too.

The comparison in Table 2-1 indicates that Python, Perl, and Ruby all have a small learning curve, have concise code, are free and open-source, and have bindings for all the major GUI libraries available. While Perl is arguably the most concise and fastest scripting language, this comes at the expense of code readability and simple data structures. This allowed us to narrow the choice down to Python and Ruby. At that point, it became a matter of personal preference as there are many similarities between these two languages. Ultimately, Python was chosen because of its large development community, intuitive handling of lists/arrays and hashes/dictionaries, and the availability of third-party libraries.

There are existing instances of others in the nuclear industry using Python for GUI development, e.g. the work of Touran [**29**]. As indicated earlier, all major GUI toolkits have bindings available for Python. While Touran had used the wxWidgets library for implementing GUI widgets, we have opted to use the

PyQt4 library, which provides Python bindings for Qt4. This particular library was chosen due to its cross-platform portability, rich widget set, and stable application programming interface (API).

2.2.2 Description of Modules

The GUI source code is divided into nine different modules as follows:

- *interface.pyw*: No-console script that displays the GUI
- ***gui****/mainWindow.py*: Main window for the GUI
- ***gui****/listElementDialog.py*: Dialog window listing all elements in the run
- ***gui****/listMaterialDialog.py*: Dialog window listing all materials in the run
- ***gui****/editElementDialog.py*: Dialog window to edit a fuel element
- ***gui****/editMaterialDialog.py*: Dialog window to edit a material
- ***gui****/editPathDialog.py*: Dialog window to edit a fuel path
- ***gui****/editTimeDialog.py*: Dialog window to define the cycle times
- ***gui****/geometry.py*: Defines the problem geometry

Together, these modules combine for a total of about 1600 lines of source code.

2.2.3 Installing

Since Python is an interpreted language, the GUI does not have to be compiled. Any system can run the GUI if Python and PyQt4 are installed. On Linux, Python and PyQt4 can be installed through package managers, whereas on Windows machines, Python and PyQt4 need to be downloaded and installed. It is required to use Python 2.6 or higher due to the use of the new string formatting style standard in Python 3.0 (Python 2.5 and lower do not support the new string formatting style). Given that a Windows user may not wish to install Python and PyQt4 just to run the GUI, a Windows installer was created using the py2exe extension [**30**] and the Nullsoft Scriptable Install System (NSIS) [**31**] that includes all the necessary dynamic-link libraries. Figure 2-2 shows the dialog window when using the NSIS installer.

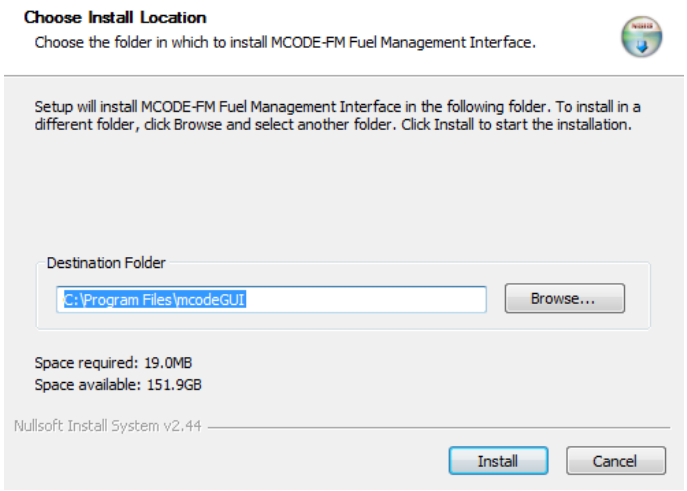

Figure 2-2 NSIS installer for the GUI

2.2.4 Main Window

The main window, shown in Figure 2-3, for the GUI displays a layout of the core for a given cycle where those positions shaded in dark gray are occupied by fuel assemblies and those in light gray by dummy assemblies. Each position occupied by a fuel assembly also has a label that shows the name of the fuel path. By default, the name of the fuel path will be the name of the fuel location that it originated in and the cycle at which it was introduced in parentheses.

The user can select a fuel location on the core layout and add a fuel path, edit an existing path, or remove an existing path. When setting up a run, the first thing that the user should do, although not strictly necessary, is define the cycle times, i.e. the times at which fuel is to be introduced, moved, or removed from the reactor. Above the core layout, the user can click a button to open a dialog window to edit the cycle times, shown in Figure 2-4. Once this is done, the user can select the cycle from the 'Select Time:' drop-down box.

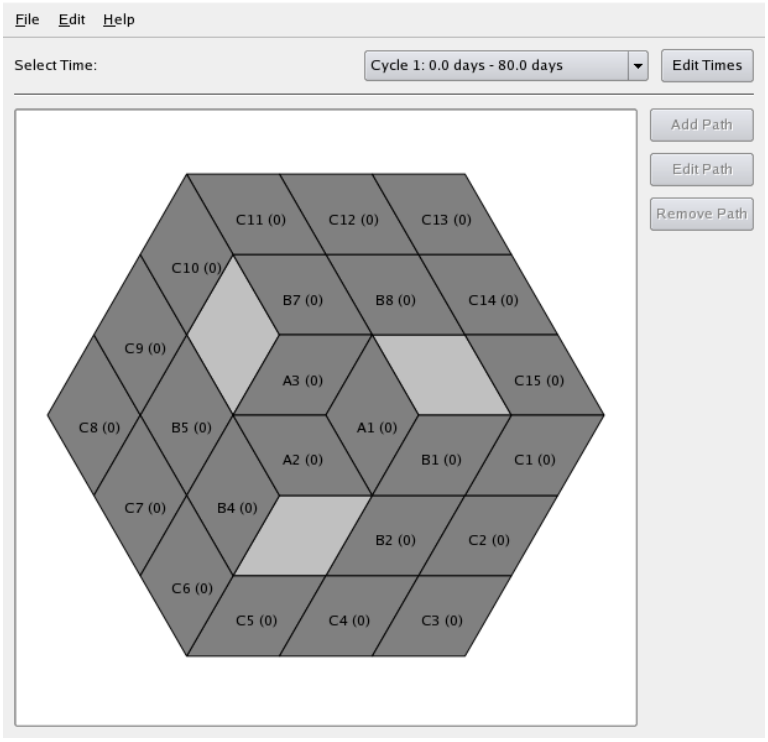

Figure 2-3 Main window for the graphical user interface

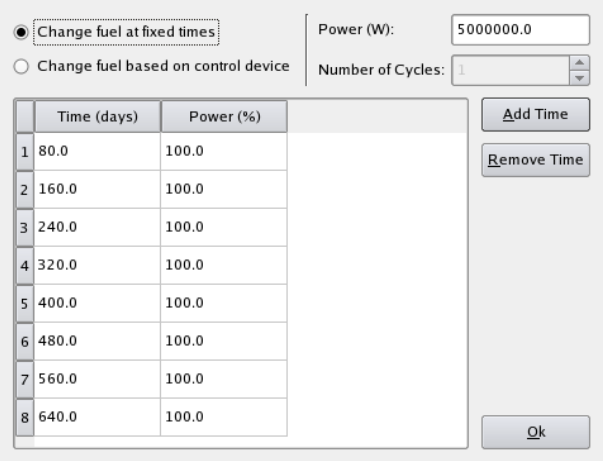

Figure 2-4 Dialog window editing cycle times

2.2.5 Creating/Editing Paths

When a user chooses to add or edit a fuel path from the main window, a separate dialog window appears as shown in Figure 2-5. From this dialog, the user simply clicks the fuel locations in the order that the fuel is to move from one to another. Additionally, the user can flip or rotate a fuel element at a certain cycle via the buttons at the bottom left side of the dialog window. A fuel element that has already been defined can be assigned to the fuel path from a drop-down box. Additionally, there is an option labeled 'New Element…' on the drop-down box that will open up a separate dialog window that allows the user to define a new fuel element. If the user does not assign a fuel element to the path, the drop-down box will be colored red to draw attention. All fuel paths must have fuel elements assigned to them in order for the run to be exported to an input file that can be read by the fuel management wrapper.

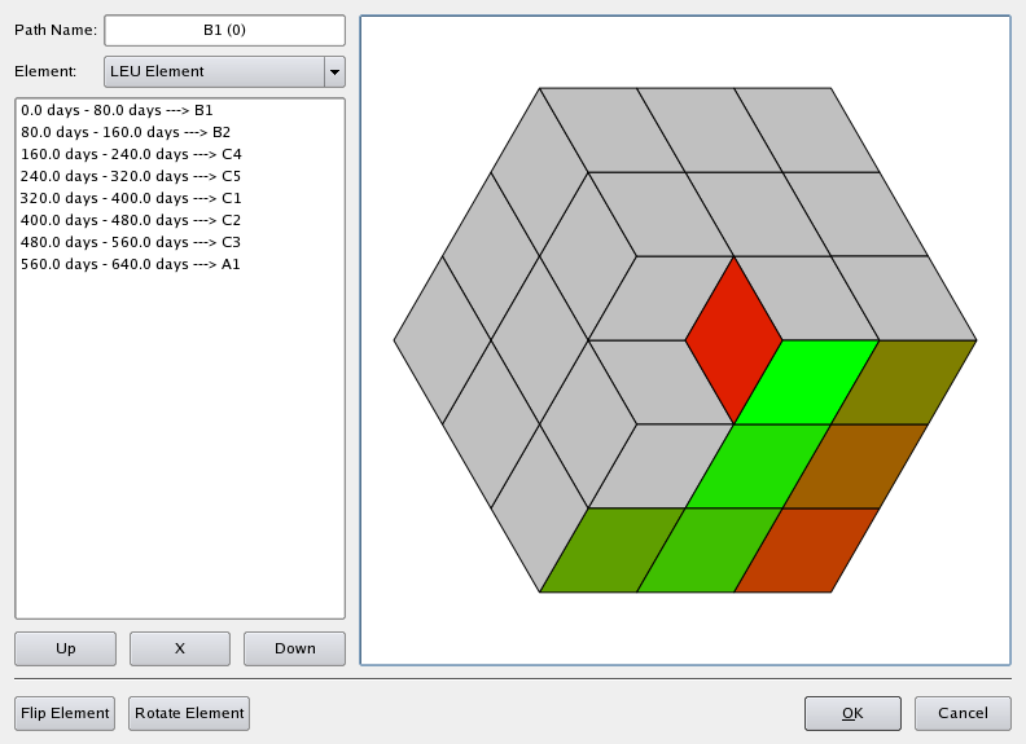

Figure 2-5 Dialog window for specifying fuel locations in a fuel path

2.2.6 Creating/Editing Fuel Elements

If the user chooses the 'New Element…' option on the drop-down box when defining a fuel path, a dialog window appears as shown in Figure 2-6. From here, the user can adjust the number of axial and radial nodes. Since the fuel assemblies in the MITR have fuel plates, specifying a radial mesh allows multiple plates to share a single material that is depleted (grouped plates are displayed with the same color as in Figure 2-6). This is ideal in locations where the power gradient is small. The axial mesh subdivides each radial mesh node into equal segments. The user can select a particular node using the visual interface and the 'Current Node' drop-down box and then apply a material using the 'Current Material' drop-down box. One option on the 'Current Material' drop-down box is titled 'New material…' that allows the user to define a new material to use.

If the user wants to group several plates to share a single depletion node, this is done by dragging a box with the mouse that intersects the plates that are to be grouped and then clicking the 'Group Radial' button. Similarly, selecting several plates and clicking the 'Ungroup Radial' button causes each plate to have its own depletion node.

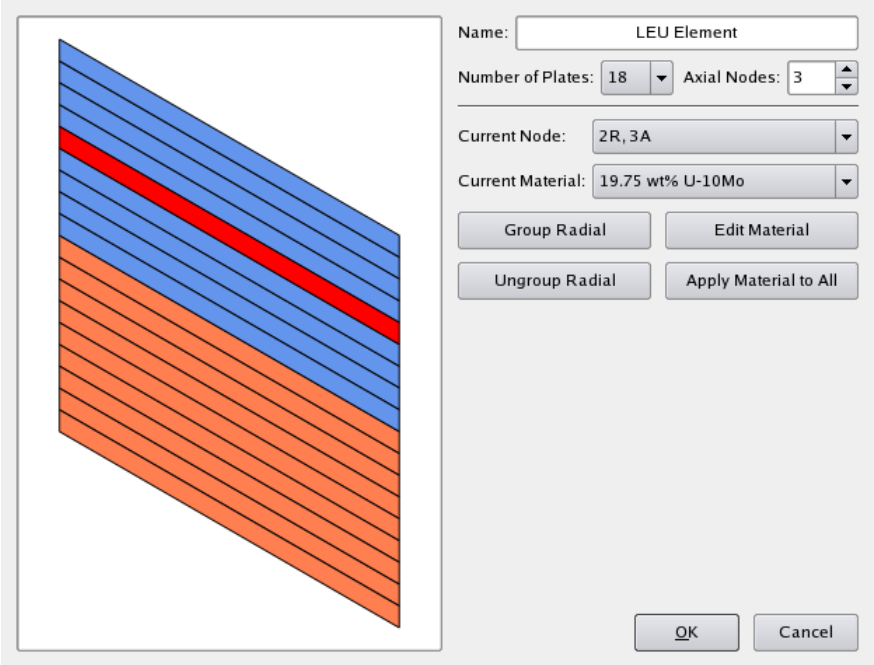

Figure 2-6 Dialog window for specifying the radial/axial mesh and assigning materials

In addition to being able to add a new fuel element directly from the fuel path dialog window, the user can also view a list of all the fuel elements defined by selecting the Edit → Elements option on the menu-bar for the main window of the GUI. This option displays a dialog window as shown in Figure 2-7. From this dialog, the user can add, edit and remove fuel elements. Additionally, the user can load an element from a previous run with depleted materials by clicking the 'Load Element…' button.

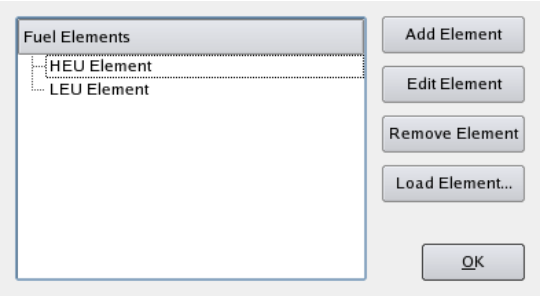

Figure 2-7 Dialog window displaying all fuel elements in the run

2.2.7 Creating/Editing Materials

When a user chooses to add a new material when defining a fuel element, a dialog window appears as shown in Figure 2-8. This dialog window allows the user to define a material in terms of its density, isotopes, and the atom/weight percentage of each isotope. As mentioned before, the density and atom/weight fractions follow the MCNP conventions for specifying material compositions.

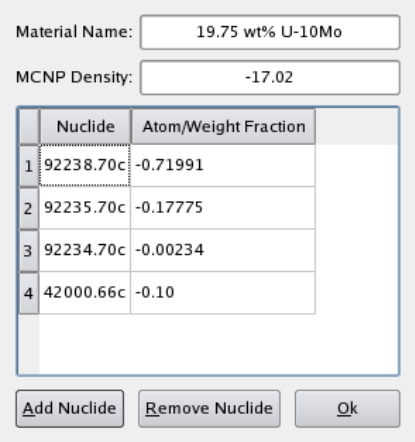

Figure 2-8 Dialog window for specifying a material composition

In addition to being able to add a new material directly when defining a fuel element, the user can also view a list of all the materials in the run by selecting the Edit → Materials option on the menu-bar for the main window of the GUI. This option displays a dialog window as shown in Figure 2-9. From this dialog, the user can add, edit and remove materials. Additionally, the user can load materials from an MCNP input file by clicking the 'Load Material…' button. Any materials that are added are listed under 'Global Materials'. These are materials that are accessible when defining any fuel element. 'Fuel Element Materials', on the other hand, only appear in a single fuel element when the user chooses to add a material from the fuel element dialog window.

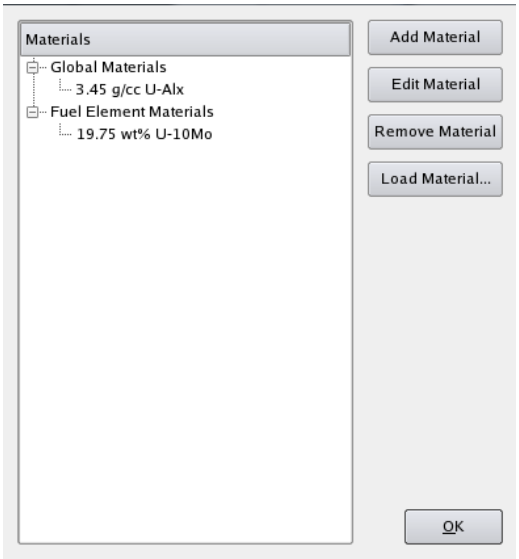

Figure 2-9 Dialog window displaying all materials in the run

2.2.8 Saving/Exporting a Run

At any time, the user can save the run by selecting the File > Save Run… option from the menu-bar on the main window of the GUI. This option saves all the fuel paths, fuel elements, materials, and associated run data in a binary file with a .run extension.

The user can also export the run to an input file that can in turn be read by the MCODE fuel management wrapper. In order to do so, the run must be completely specified, i.e. each fuel path must have a fuel element assigned to it, each node of each fuel element must have a material assigned to it, and all the cycle times must be defined. The exported file is an ASCII file that is fairly "intuitive" to read. This makes it easy for the user to make small changes to the input file, e.g. changing materials or cross-sections, without having to do so via the graphical user interface. Of course, the GUI may be the preferred option for changing run data anyway.

2.3 Summary

A graphical user interface was developed to automate the creation of input files to be read by MCODE-FM, the wrapper code that handles the fuel management operations in MCODE. While it currently is specific to the MITR, the object-oriented approach taken should make it fairly simple to extend the graphical user interface to be used with other reactor types. Python was chosen as the programming language for development thanks to its easy learning curve, code conciseness and readability, large development community, intuitive handling of lists/arrays and hashes/dictionaries, and the availability of third-party libraries. PyQt4 was chosen as the GUI development extension.

The GUI requires no installation on machines where adequate versions of Python and PyQt4 are installed. On Windows machines, a package is available that allows the user to install the GUI without having Python or PyQt4 installed.

Using the GUI to setup a run is simple and intuitive. The user can setup a run by successively defining the fuel paths, which describe where each fuel element moves from cycle to cycle; fuel elements, which describe the number of depletion nodes to be used and the materials to be assigned to each node; and the materials themselves. Once the run is completely defined, the user can save the run to be opened at a later time or export it to an input file that can be read by the fuel management wrapper, to be described in the following chapter.

3 Fuel Management Wrapper Development

3.1 Methodology

There are generally two approaches one can take to perform fuel management operations[1] when using MCODE. The first approach, which was taken in prior work [**32**], is to swap material numbers in MCNP every time a fuel assembly is to be moved. This approach has the advantage that the geometry needn't be changed in the MCNP input file even when a fuel assembly is being moved from one location to another. However, the downside is that a single fuel assembly is difficult to track from cycle-to-cycle. Additionally, for a core such as the MIT Reactor (MITR) whereby dummy assemblies are utilized, it is not possible for the locations of the dummies to also move from cycle-to-cycle. Introducing new fuel into the core after the first cycle adds yet another difficulty.

The second approach to perform fuel management operations is to change the geometry of the MCNP input file for each cycle while keeping the material numbers in each assembly the same. In one sense, this is more complicated than the first approach since the input file must actually be changed, but the bookkeeping required is simpler since each depletion node in each assembly always has the same material number assigned to it. This approach is clearly the more "intuitive" approach and was thus chosen for the fuel management wrapper.

3.2 MCODE Modification

In order for this process to work properly, it was necessary to make a small change in MCODE. As time progresses in an MCODE run, MCNP does not track all the isotopes that ORIGEN does because many isotopes are not important as far as neutron absorption rates are concerned and not all ORIGEN isotopes have corresponding MCNP cross-section libraries. Normally, at the beginning of an MCODE run, the only isotopes present in the problem are completely specified by the MCNP data. For our purposes, at the beginning of a new cycle, we want to retain all the ORIGEN isotopes in each material from the previous cycle.

MCODE was modified to be able to distinguish whether a material is "fresh" or whether ORIGEN data is already present, i.e. the material has been burned, through a new user input option. A single parameter was added to the irradiation material specification card in MCODE that tells it whether or not the material has been previously burned. The new irradiation material specification is:

[1] By fuel management operation, we mean the physical movement of a fuel assembly from one in-core location to another, the introduction of a fresh fuel assembly, or the removal of a depleted fuel assembly.

Format:

mt-num vol ORG-XS-lib TEMP IMP MCNP-XS-opt ntal burned

where mt-num = irradiation material number (defined in the MCNP material block);
vol = occupied volume (in units of cm^3);
ORG-XS-lib = attached ORIGEN one-group cross section library;
TEMP = temperature (an integer in unit of K);
IMP = the neutron absorption fraction threshold;
MCNP-XS-opt = MCNP cross section library option
0 = use user-specified MCNP cross section library,
1 = use MCNP cross section library as indicated by `preproc`;
ntal = MCNP F4 tally number (between 0 and 99);
burned = fresh or depleted material option
0 = fresh material,
1 = depleted material (PCH file already present).

Figure 3-1 New MCODE irradiation material specification

By specifying a material as burned, MCODE will know that a PCH file with all the ORIGEN nuclide number densities is already present in the temporary directory (tmpdir11, by default) and thus will not need to perform the extra depletion step normally required to determine what isotopes to track in MCNP and will not assume zero for all the other ORIGEN nuclides.

3.3 Wrapper Description

3.3.1 Language Choice

While the earlier version of the fuel management wrapper was written in Fortran 90, we have opted again to use Python to maintain consistency with the graphical user interface, as well as for the reasons previously discussed. Python is particularly suited for this task as performing fuel management operations entails creating and manipulating text files and moving data to and from files. In addition, the same data classes that the GUI uses for storing fuel path, fuel element, and material data can be used for the fuel management wrapper as well.

3.3.2 Description of Modules

The fuel management wrapper source code is divided into nine different modules as follows:

- *mcodeFM.py*: Main script that handles execution of MCODE
- *subs.py*: Various subroutines for fuel management wrapper
- *fileIO.py*: Subroutines for loading data from MCNP files
- ***data****/classes.py*: Data classes describe in section 2.1
- ***data****/fuelLocations.py*: Definitions of fuel locations for use in MCNP
- ***utils****/keffsearch.py*: Criticality search module

- ***utils****/power_profile.py*: Utility to determine power distribution
- ***utils****/isotope.py*: Profiles the concentration of specific isotopes

Together, these modules combine for a total of about 1300 lines of source code. Thus, with the graphical user interface, the entire code package is contained in less than 3000 lines of source code.

3.3.3 Code Logic

The logic behind the fuel management wrapper is rather simple. First, the wrapper reads the input file exported from the GUI. This input file has information about what materials are in each fuel element and where each fuel element is to move from cycle to cycle. Based on this, it creates an MCODE input file with the appropriate geometry and a new directory for the first cycle. The MCODE input file is based on the data exported from the GUI as well as a "skeleton" MCNP input file. The skeleton input file is a normal MCNP file with all the geometry of the reactor defined except for the fuel elements themselves. Figure 3-2 shows the geometry of a skeleton input file for the MITR.

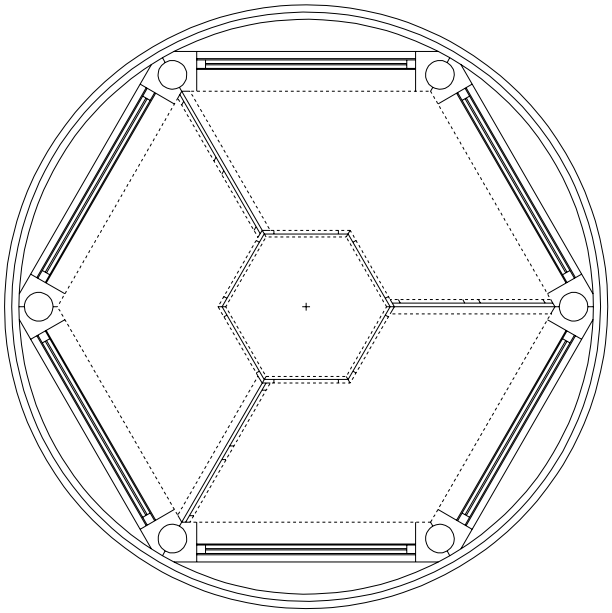

Figure 3-2 "Skeleton" MCNP model of the MITR with no fuel assembly detail

Since the skeleton input file does not have any fuel element geometry in it, it can be used for HEU and LEU cores or transition cores that consist of both HEU and LEU fuel elements.

When the fuel management wrapper creates an MCODE input file, it fills in the areas left blank in the skeleton input file with fuel elements as defined in the GUI. First, it defines surfaces to axially subdivide the fuel plates into several depletion nodes as needed. Then, with these surfaces as well as a series of surfaces already in the skeleton input file that define the radial boundary of the fuel plates, cells are constructed to define the fuel meat, fuel clad, and coolant in each fuel element. It should be noted that since the fuel plate surfaces are defined in the skeleton input file rather than being calculated by the fuel management wrapper, the wrapper will currently only work with fuel elements that have 15 or 18 plates (HEU and LEU, respectively).

After the cells for the fuel elements have been written, a cell for each fuel location is written and is filled with the universe corresponding to whatever fuel element should occupy that location at the given cycle. If a fuel location does not have a fuel element assigned to it, the cell will be filled with the universe for a dummy element by default. If the user desires to have an experimental facility in one of the fuel locations, this default behavior would have to be overridden for that particular fuel location.

With the fuel locations and fuel elements defined, the geometry for the cycle is complete. The fuel management wrapper will then add data cards to specify temperatures for each of the cell cards that it created as well as the material definition for all irradiation materials. Lastly, the fuel management wrapper will write tally specification, irradiation material specification, and depletion option cards that MCODE reads in at the beginning of a cycle. At that point, the MCODE input file is complete and is ready for execution.

In addition to writing the MCODE input file at each cycle, the fuel management wrapper will also write out a file called "data" that contains information on what cells and materials were assigned to each depletion node. The data file is useful for creating post-processing scripts that can read output from the MCODE run and rewrite it to a format that is more amenable for analysis. For instance, two post-processing scripts were created for the MITR to simplify analysis. The first, a power profile utility, reads the output data and creates a comma-separated values (csv) file with the power produced by each plate. The second script is similar to the first; instead of outputting the power in each plate however, it outputs the number density of any given nuclide. By checking, for instance, the number density of ^{235}U, the user can do a quick check to ensure that fuel elements are being moved to the correct fuel locations since the density of ^{235}U should generally decrease with increasing burnup.

Once the MCODE input file and "data" file have been written, the wrapper then calls MCODE and waits for the run to finish. After it has finished, the wrapper updates the material compositions of each depletion node based on the MCODE output. With the new material compositions and fuel path data at hand, a new MCODE input file for the next cycle can be created, placed in a new directory, and run. This process is repeated until all the cycles have finished. A logical flow chart for the fuel management wrapper is shown in Figure 3-3.

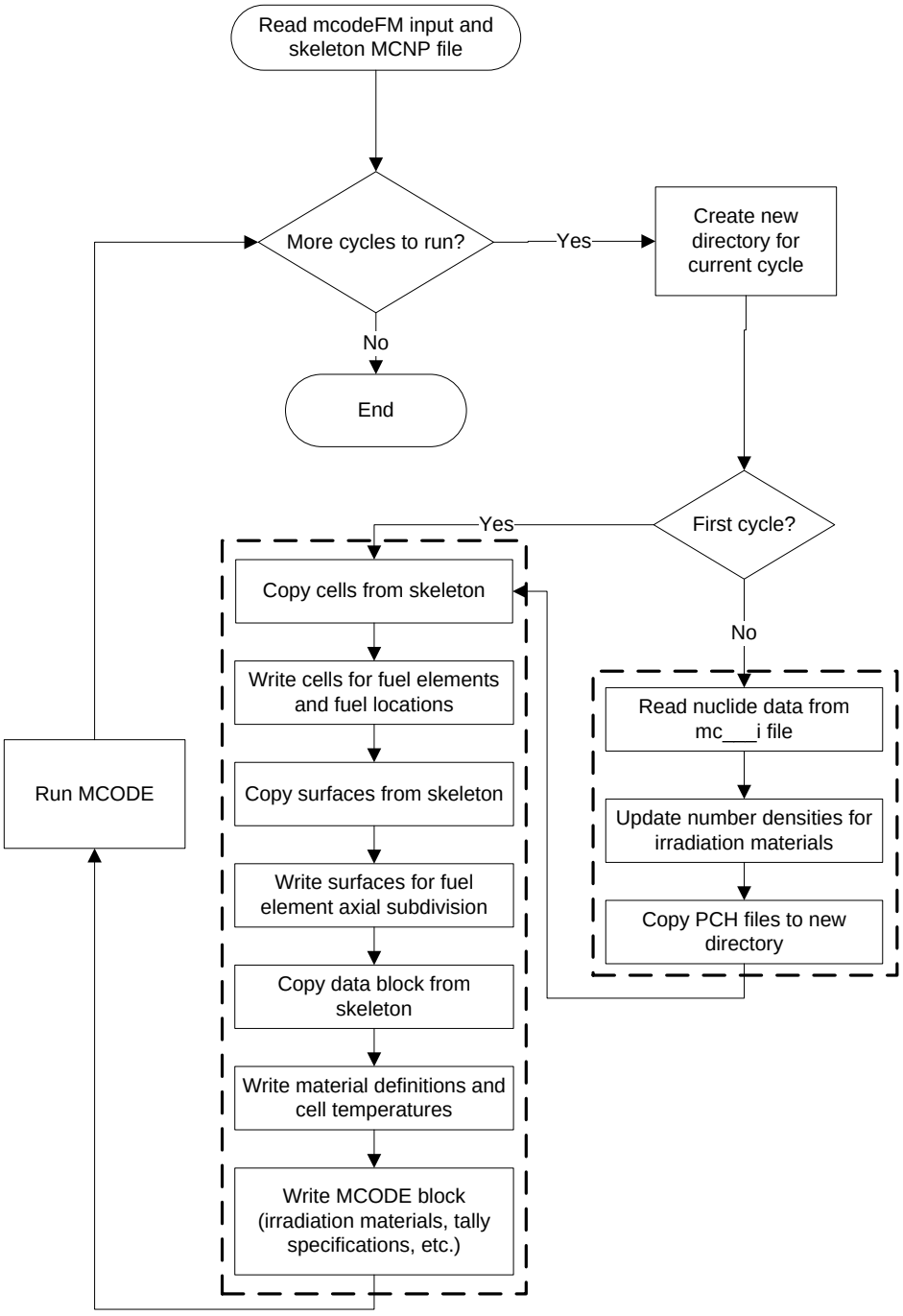

Figure 3-3 Logical flow chart for the fuel management wrapper

3.4 Criticality Search

Due to the MITR having control blades at its periphery, movement of the control blades can have a significant effect on the power distribution in the core. When the control blades are close to being fully inserted, this causes the power to shift more to the center of the core, causing higher peaking in the A- and B-rings. Conversely, when the control blades are fully withdrawn, the power flattens out in the A- and B-

rings somewhat and the reflector peaks at the edge of the C-ring are exacerbated. Thus, to determine the location of the highest peaking in the core as it is being depleted, it is important to model the physical movement of the control blades in addition to the fuel movement from cycle to cycle.

As the core is being depleted, the control blades should always be positioned at a height such that the core is critical. Thus, the critical position of the control blades must be determined at each depletion timestep before the neutron flux calculation is performed. In most reactors, the control device only moves in one dimension. The problem of determining the critical position of a control device is equivalent to determining the intersection of the multiplication factor as a function of the control device position and unity, i.e. solving the equation:

$$k(z) - 1 = 0 \tag{3.1}$$

where $k(z)$ is the multiplication factor as a function of the control device position, z. However, the form of the function $k(z)$ is not known *a priori* and we are forced to attempt a solution by means of a numerical algorithm. It is not possible to apply Newton's method to iteratively solve this equation since Newton's method requires evaluating the first derivative of the function $k(z)$. However, we can use the secant method to iteratively solve for the critical position. The secant method entails evaluating the multiplication factor at two different positions, estimating the differential worth based on the evaluated multiplication factors, and extrapolating to determine a new estimated critical position. The updated differential worth at each iteration is:

$$\left(\frac{\partial k}{\partial z}\right)_i \cong \frac{k_i - k_{i-1}}{z_i - z_{i-1}} \tag{3.2}$$

where k_i is the eigenvalue at the *i*-th iteration and z_i is the control device position at the *i*-th iteration. Then, the predicted critical position of the control device based on the calculated differential worth is:

$$z_{i+1} = z_i + \left[\left(\frac{\partial k}{\partial z}\right)_i\right]^{-1} \cdot (1 - k_i) \tag{3.3}$$

Some care must be taken when using the secant method to solve Eq. (3.3). Firstly, because the neutron flux calculation is being performed using a Monte Carlo simulation, there will be a stochastic uncertainty on the calculated eigenvalues. As a result, it becomes important to consider the desired level of precision on the calculated eigenvalue, i.e. how many cycles should be run. If the control device is very far from its critical position, one only needs a rough estimate of the eigenvalue and thus it would be wasteful to calculate the eigenvalue very precisely. To reduce the time spent converging the control device to its critical position, an adaptive batching algorithm was implemented based on the control device search in MC21 [**33**].

The idea behind the adaptive batching algorithm is that if the distance the control device is to move is less than the uncertainty on the predicted critical position, it makes more sense to continue with the current iteration and find the eigenvalue to a higher precision. To calculate the uncertainty on the predicted position, we first need to know the uncertainty on the differential worth of the control device. By propagating the uncertainty on the calculated eigenvalues, the uncertainty on the differential worth is found to be:

$$\sigma^2_{\frac{\partial k}{\partial z},i} = \frac{\sigma^2_{k,i} + \sigma^2_{k,i-1}}{(z_i - z_{i-1})^4} \tag{3.4}$$

where σ stands for the standard deviation of each quantity. The uncertainty on the predicted critical position is then:

$$\sigma^2_{z,i+1} = (1 - k_i)^2 \cdot \left[\left(\frac{\partial k}{\partial z}\right)_i\right]^{-4} \cdot \sigma^2_{\frac{\partial k}{\partial z},i} + \left[\left(\frac{\partial k}{\partial z}\right)_i\right]^{-2} \cdot \sigma^2_{k,i} \tag{3.5}$$

The criterion for whether to move the control device based on the calculated uncertainty on the predicted critical position is:

$$c \cdot \sigma_{z,i+1} < |z_{i+1} - z_i| \tag{3.6}$$

In the MC21 implementation, the 1σ uncertainty on the predicted critical position was used, i.e. $c = 1$ was used in Eq. (3.6). In our testing, using the 2σ uncertainty resulted in a more stable approach to the critical position and was thus implemented in the criticality search.

If between two successive iterations, the eigenvalue changes very little due to stochastic effects, the differential worth will be very small thus causing the predicted critical position from Eq. (3.3) to be very unreliable. To overcome this problem, the differential control device worth is set to a user-defined value if at any time the uncertainty in the differential worth is more than half the differential worth. A logical flow chart of the criticality search algorithm implemented is shown in Figure 3-4.

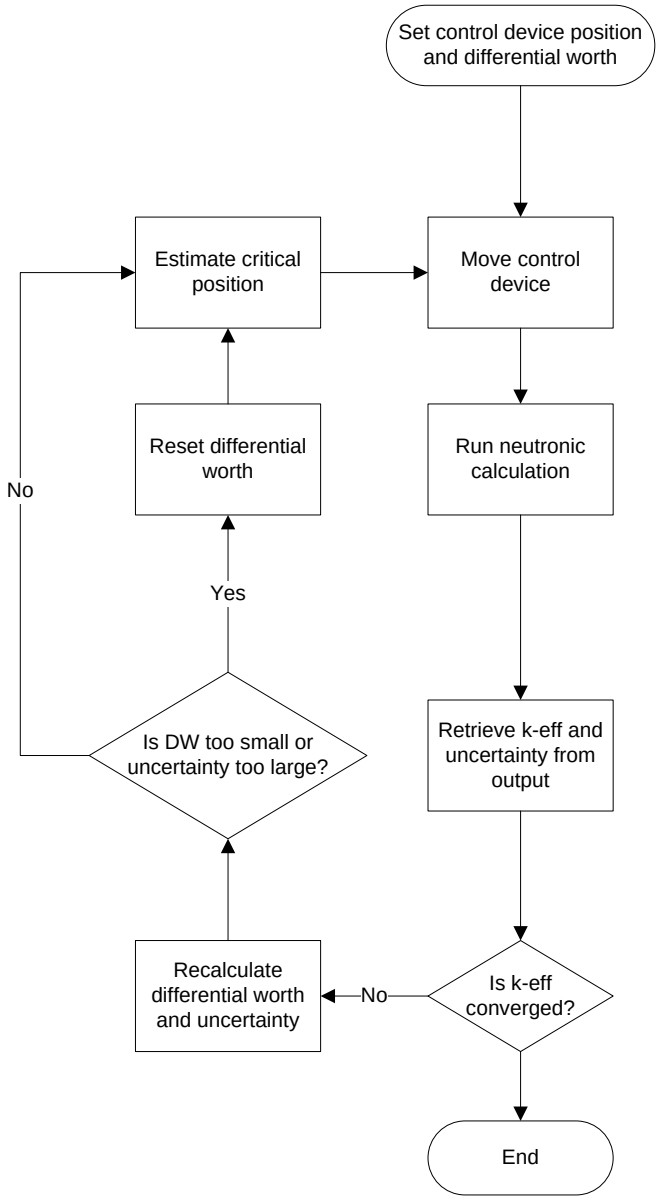

Figure 3-4 Logical flow chart for criticality search on control device

Recall that MCODE-2.2 is based on a predictor-corrector burnup methodology. First, the beginning-of-timestep (BOT) reaction rates from a flux calculation are used to predict the end-of-timestep (EOT) material compositions. Then, the predicted EOT material compositions are used to calculate EOT reaction rates from a flux calculation. A set of corrected EOT material compositions is calculated by re-depleting over the timestep using the EOT reaction rates with BOT material compositions. The final EOT materials compositions are taken to be the average of the predicted and corrected EOT material compositions. This methodology allows for the inclusion of a control device search for criticality whenever the neutron flux solution is computed. Although a change in the position of the control device will perturb the flux solution, the predictor-corrector approach will average out the difference in the reaction rates.

3.5 Summary

In the fuel management wrapper developed, fuel management operations are performed by changing the geometry of the MCNP input file at the beginning of each cycle while keeping the material numbers in each assembly the same. This approach was found in practice to be more versatile than simply swapping material numbers to achieve the affect of fuel elements moving. Since a separate MCODE run is performed for each cycle, MCODE was modified to be able to distinguish whether a material is "fresh" or whether ORIGEN data is already present, i.e. the material has been burned, through a new user input option.

The fuel management wrapper reads an input file exported from the GUI. This input file has information about what materials are in each fuel element and where each fuel element is to move from cycle to cycle. Based on this, it creates an MCODE input file with the appropriate geometry and a new directory for the first cycle. The wrapper then calls MCODE and waits for the run to finish. After it has finished, the wrapper updates the material compositions of each depletion node based on the MCODE output. With the new material compositions and fuel path data at hand, a new MCODE input file for the next cycle can be created, placed in a new directory, and run. This process is repeated until all the cycles have finished.

Due to the MITR having control blades at its periphery, movement of the control blades can have a significant effect on the power distribution in the core. Thus, to determine the location of the highest peaking in the core as it is being depleted, it is important to model the physical movement of the control blades in addition to the fuel movement from cycle to cycle. A numerical algorithm was implemented based on a secant method that determines the critical position of a control device. In order to improve the speed of convergence, an adaptive batching algorithm was implemented based on the control device search in MC21 [**33**].

3.5.1 Input Requirements

Several input files are required to initiate a run using the MCODE fuel management wrapper. The first input file is the data exported using the graphical user interface or created by hand. To create an MCODE file, the fuel management wrapper also requires a skeleton MCNP input file as discussed above. The fuel management wrapper assumes this file is named "skeleton_input" unless the user specifies otherwise.

The control device criticality search can be utilized by writing a shell script that MCODE will call when it runs a flux calculation. The script should first run the criticality search on the MCODE input file to adjust the control blade position to the critical height and subsequently call MCNP to perform the actual flux calculation used to determine the reaction rates that are passed to ORIGEN. The control device search also requires its own input file that specifies what cells or surfaces compose the control device as well as the range of the control device.

Lastly, if the user wants to use a source file to start each flux calculation in MCNP, a "srctp" from MCNP must be supplied as well. Examples of all of these input files are listed in Appendix C. Table 3-1 shows a summary of the required and optional input files for a run using the MCODE fuel management wrapper.

Table 3-1 Summary of input files for MCODE fuel management wrapper

File	Required	Purpose
mcodeFM_input	Yes	This file contains run data created in the graphical user interface. The data include what materials are in each fuel element and where each fuel element is to move from cycle to cycle, as well as the power of the reactor at each cycle, and the length of each cycle.
skeleton_input	Yes	This file is a normal MCNP file with all the geometry of the reactor defined except for the fuel elements themselves. It is used as a template to create an MCODE input file.
control_input	No	This file is optional and tells the criticality search utility what cells/surfaces define the control device and the range of the control device.
mcnp.sh	No	Having MCODE call a script instead of MCNP directly allows a criticality search to be performed prior to the flux calculation in MCNP.
srctp	No	In order to cut down on the number of inactive cycles, a source file can be used to converge the source distribution quickly.

4 Testing and Verification

4.1 MCODE Modification

In order to retain all the fission products from cycle to cycle when running the fuel management wrapper, MCODE was modified to be able to distinguish whether a material is "fresh" or whether ORIGEN data is already present, i.e. the material has been burned. It is instructive to ensure that the changes do not fundamentally alter the MCODE result.

The problem chosen to test the MCODE modification is a simple pin-cell model that is distributed with MCODE. Figure 4-1 shows a simple drawing of the pin-cell model. The fuel is 10.34 g/cm^3 and has an enrichment of 4.2 wt% ^{235}U. The water is at 2200 psi and 600 °F. The gap is modeled as helium and the clad is modeled as natural zirconium. Reflective boundary conditions are applied on the outer sides and vacuum boundary conditions are applied on the top and bottom. The ENDF/B-VII.0 nuclear dataset was used for cross-section evaluation.

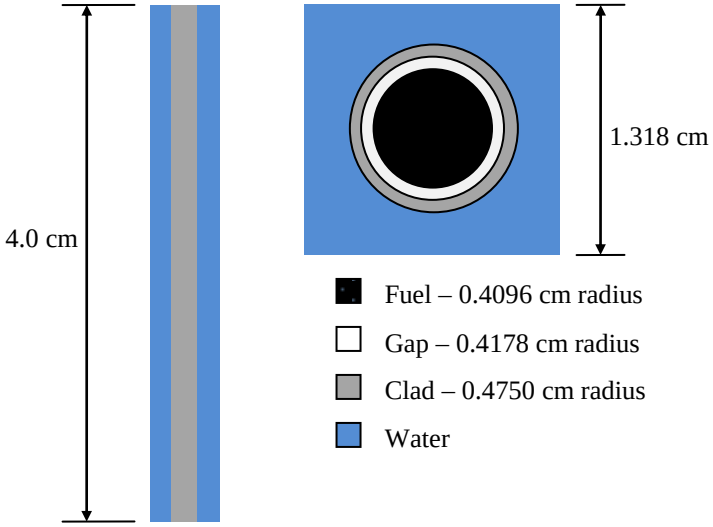

Figure 4-1 Pin-cell test model for MCODE change validation

This pin-cell model was burned in MCODE for 1350 days for two cases. In the first case, all timesteps were run using one MCODE input file. In the second case, an MCODE input file was created for each timestep with material compositions and ORIGEN isotope data files taken from the previous timestep. Each MCNP run used 5000 neutrons per cycle and 1000 cycles (990 active). The percent difference between the multiplication factor and isotope concentrations in the two cases is shown in Figure 4-2.

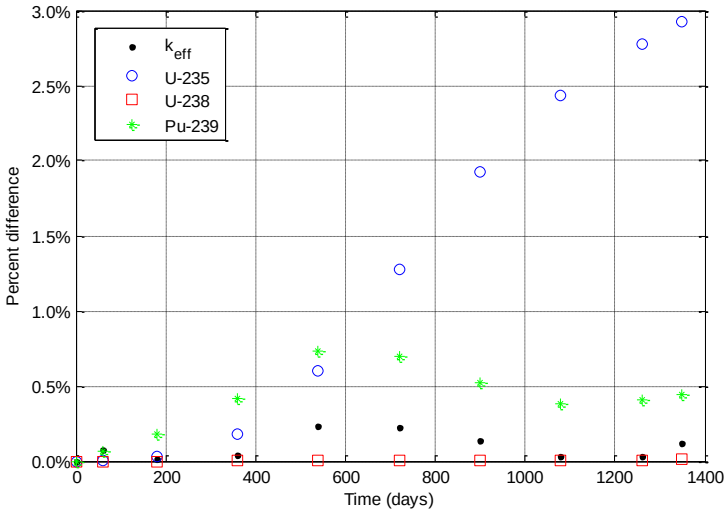

Figure 4-2 Percent difference between k_{eff} and isotope concentrations for MCODE test cases

While the percent difference between the ^{235}U concentrations in the two cases steadily increases with time, one must consider two important facts when interpreting this result. Firstly, since the reaction rates are determined stochastically, running the same case twice may give different reaction rates. Hence, there is an associated uncertainty in the multiplication factor and the isotope concentrations that is not reflected in Figure 4-2. This uncertainty is difficult to characterize since the stochastic uncertainties are not propagated through the depletion calculation. One must also keep in mind that the ^{235}U concentration decreases by two orders of magnitude over the course of the run and thus the absolute error in the concentration is not increasing even though the relative error appears to increase. This can be confirmed by looking at the ^{235}U concentration for the two cases as shown in Figure 4-3.

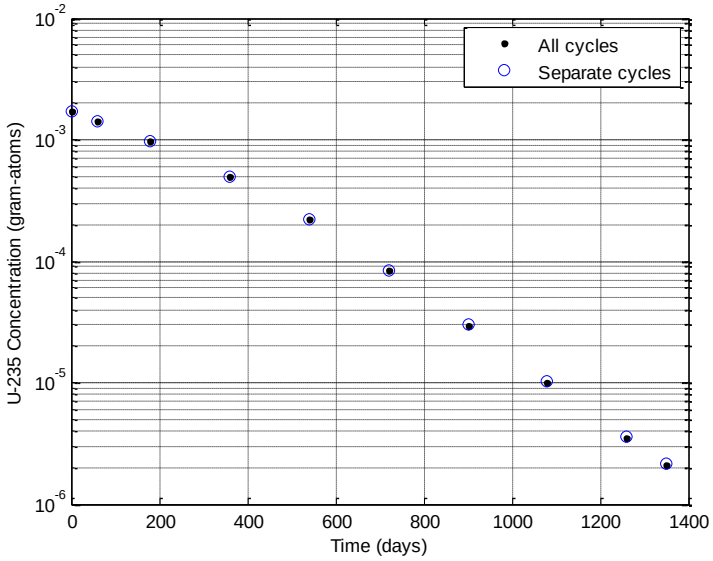

Figure 4-3 ^{235}U concentration for MCODE test cases

4.2 LEU Equilibrium Core

4.2.1 Description

With the MCODE changes validated, the next step is to ensure that the fuel management wrapper also performs as expected. To test the functionality of the wrapper, a 24-element core with all fresh LEU fuel was burned for 640 days with the fuel being moved in the same pattern every 80 days. The material composition of the LEU fuel follows the Y-12 specification for LEU used in research reactor fuel. Table 4-1 lists the weight fractions of the nuclides in the LEU fuel. The ENDF/B-VII.0 evaluated nuclear dataset was used for the cross-sections of the uranium isotopes and the ENDF/B-VI dataset was used for the natural molybdenum.

Table 4-1 Material composition of LEU used in equilibrium core run

Nuclide	Weight Fraction
^{234}U	0.00234
^{235}U	0.17775
^{238}U	0.71991
^{Nat}Mo	0.10

All fuel elements start with fresh fuel at the first cycle and follow the paths shown in Figure 4-4 at each successive cycle. Three fuel elements in the inner ring (A-1, A-2, and A-3) were removed at each cycle and three new fuel elements were placed in the middle ring (B-1, B-4, and B-7). The dummy elements were placed in rotationally symmetric positions (B-3, B-6, and B-9) so that after eight cycles, the material compositions in the core would be close to their equilibrium values for this particular fuel pattern. A criticality search on the control blade height was performed at every timestep using the utility described in section 3.4.

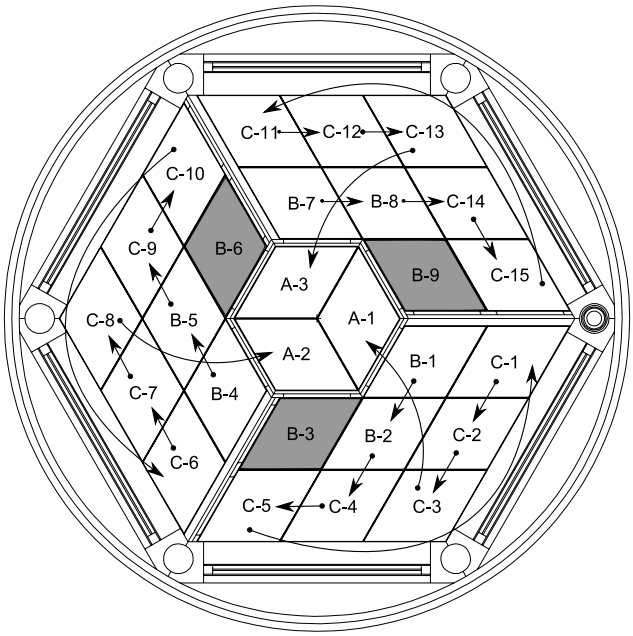

Figure 4-4 Path for each fuel element in the LEU equilibrium core run

4.2.2 Control Blade Movement

This particular test case was also run with an earlier version of the fuel management wrapper written in Fortran that existed before the development of the GUI. Rather than changing the geometry of the MCNP model at each cycle, the methodology used for performing fuel management operations in the Fortran fuel management wrapper was to swap material numbers whenever fuel needed to be moved rather than changing the geometry. Thus, we can compare the control blade height using both the old and new fuel management wrappers. A comparison of the control blade height from this run using the Fortran-based wrapper and the new Python-based wrapper is shown in Figure 4-5.

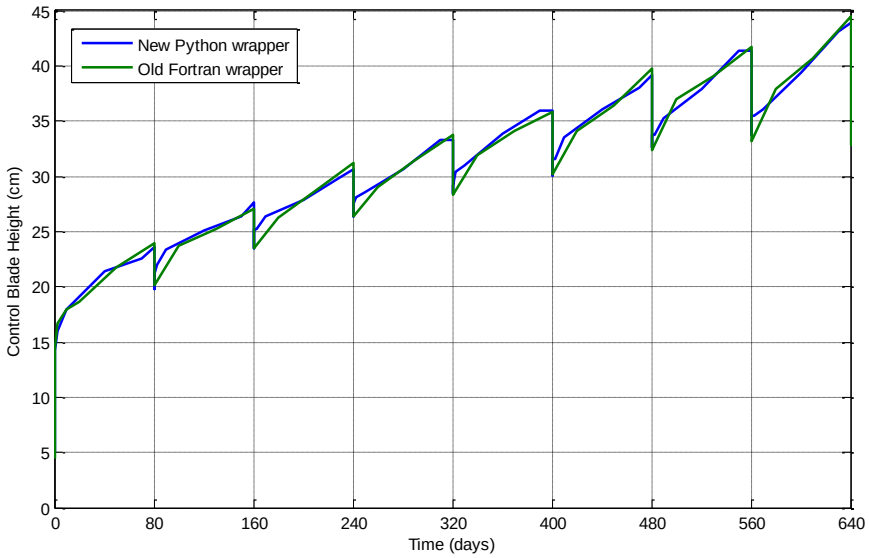

Figure 4-5 Control blade height in LEU equilibrium core run

There is a slight disagreement between the two wrappers towards the beginning of each cycle due to the use of longer timesteps in the run using the Fortran-based wrapper. However, the two wrappers do show excellent overall agreement. The behavior of the control blades is just as one would expect. At the beginning of each cycle when fresh fuel is introduced, the blades must be lowered to compensate for the positive reactivity from the new fuel but are quickly withdrawn over the course of the following three days to compensate for the buildup of equilibrium Xenon. The overall trend shows the transition from the fresh core at time zero to the near-equilibrium core at the end of the run.

4.2.3 Nuclide Concentrations

Another metric that we can use to verify expected physical behavior in this run is the concentration of particular nuclides of interest in each fuel element. We would expect that those fuel elements that have resided in the core for a longer period of time will have a lower concentration of ^{235}U. This is exactly what is shown in Figure 4-6 where the relative ^{235}U concentration decreases with residence time. One other effect that is noticeable in Figure 4-6 is that the outer plates of each fuel element have a higher burnup than the inner plates. This can be explained by the fact that the outer plates receive the highest

power as shown in Figure 4-7. The numbers displayed on top of the fuel elements indicate how many cycles each element has previously resided in the core.

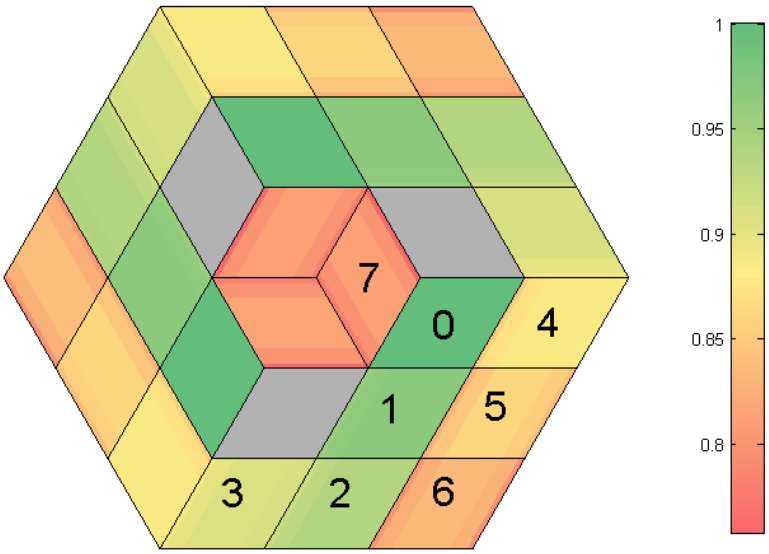

Figure 4-6 Relative ^{235}U concentration for LEU equilibrium core

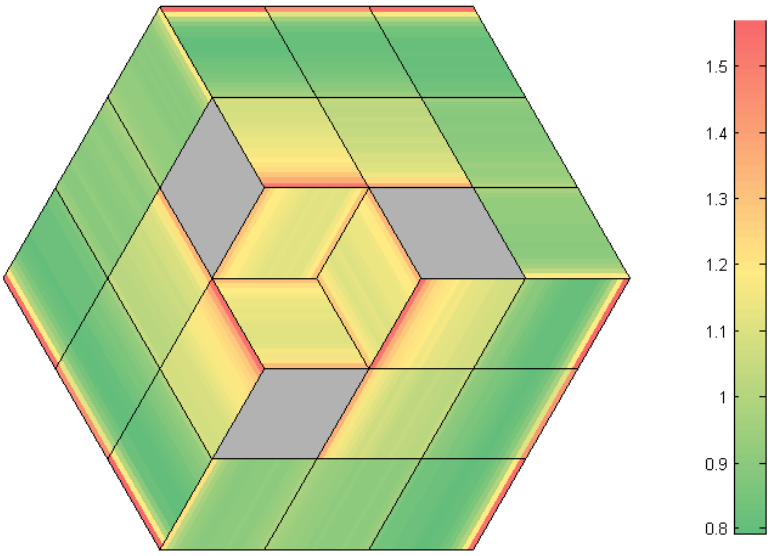

Figure 4-7 Plate power peaking factors for LEU equilibrium core

Additionally, we can check the buildup of ^{239}Pu in the fuel elements. We would expect that those elements that have experienced a higher fluence will have a higher buildup of ^{239}Pu. The observed behavior again matches our expectation. Figure 4-8 shows the density of ^{239}Pu in atom/b-cm. It is clear from this figure that as a fuel element moves through the core, the ^{239}Pu concentration builds up proportionally to its fluence. We note that at very high burnup, the concentration of ^{239}Pu may actually begin to decrease if its destruction rate due to fission exceeds its production rate due to (n,γ) capture in ^{238}U. However, for this case, the burnup is not sufficiently high to observe this effect.

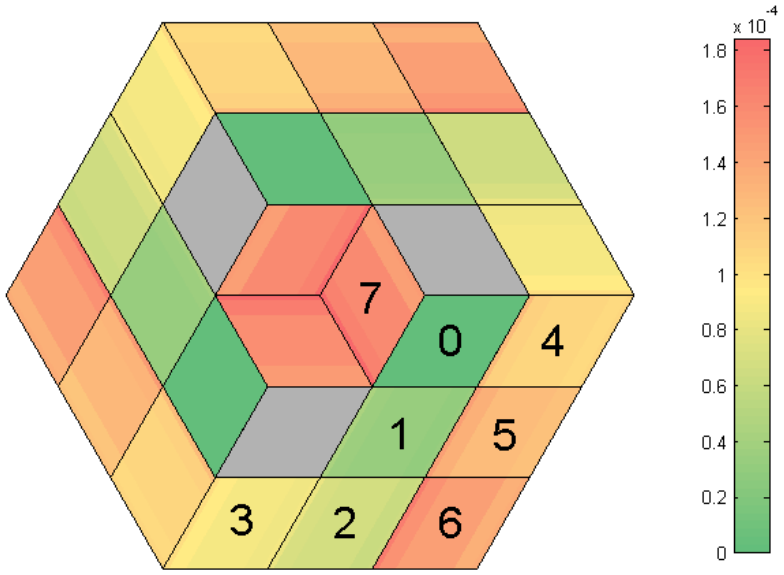

Figure 4-8 ^{239}Pu concentration in atom/b-cm for LEU equilibrium core

Looking at the C-ring fuel elements in Figure 4-8, one can see that the plates closer to the center of the core have a higher concentration of ^{239}Pu. The spectrum gradually softens going from the center of the core to the edge due to the fact that fast neutrons born closer to the edge of the core are more likely to leak or be moderated in the D_2O reflector and return to the core as thermal neutrons. Thus, the (n,γ) absorption rate in ^{238}U will decrease towards the edge of the core as shown in Figure 4-9 since a softer spectrum will result in less resonance absorption in ^{238}U.

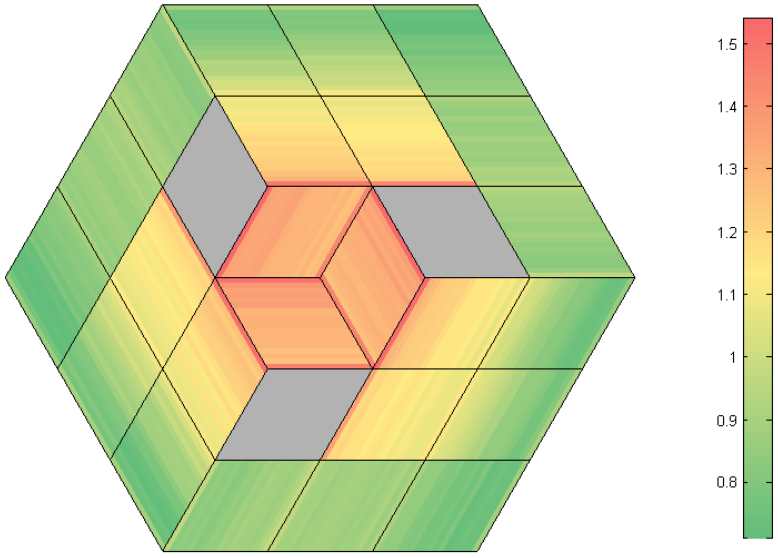

Figure 4-9 Relative radiative capture reaction rate in ^{238}U for LEU equilibrium core

The equilibrium core run took about two days to run with MCNP running in parallel on six 2.2 GHz AMD Opteron processors.

4.3 HEU "Equilibrium" Core

4.3.1 Description

While a pure LEU core has not been successfully modeled in REBUS to date, benchmarking done on the HEU model has shown good agreement with MCNP and experimental results. It is thus possible to compare the fuel management capabilities of REBUS and the MCODE fuel management wrapper by testing an HEU model.

Although there is no set refueling scheme for the MITR-II core (HEU), new fuel is usually placed in the B-ring and run for several cycles prior to being moved to the C-ring in order to reduce power peaking at the edge of the core. In modeling a transition from new fuel to a representative "equilibrium" core, the early operating history of the reactor was studied beginning with operation of the new HEU core #2 which contained five solid Al dummies with 22 HEU fuel elements. The representative refueling scheme from this core to an equilibrium core is shown in Table 4-2.

Table 4-2 Fuel movements for transition from fresh HEU core to equilibrium core

Refueling	Movement	Burnup (MWD)
1	• Initial five dummy core (core configuration #2)	1000
2	• Replace two B-ring dummies with new fuel	667
3	• Replace all fuel in B-ring with new fuel. • Flip A-ring elements • Flip all C-ring elements	416
4	• Replace all fuel in B-ring with new fuel • Replace A-ring element with element removed in refueling 3 (flipped) • Flip all C-ring elements	625

4.3.2 Power Peaking

These fuel movement and burnup parameters were input into both the REBUS model and the MCODE fuel management wrapper using the MCNP model and fuel element peaking results were compared with one another. It is not possible to compare individual plate power peaking since the fuel elements in the REBUS model are homogenized. In order to minimize differences in leakage, the end-of-life peaking factors from MCODE and REBUS were obtained by setting the control blades to the same height, in this case 21.3 cm (8.4 in).

The REBUS and MCODE models show remarkable agreement in power peaking results as shown in Figure 4-10 [**34**]. It should be noted that ^{135}Xe was allowed to decay in the REBUS model, while the MCODE model results are at xenon equilibrium with the blades lowered to the reference position. However, the differences in Xenon conditions have little effect on the power distribution since the Xenon

concentration at equilibrium is roughly proportional to the flux. In both cases, the xenon reactivity worth is about -2.3% Δk/k.

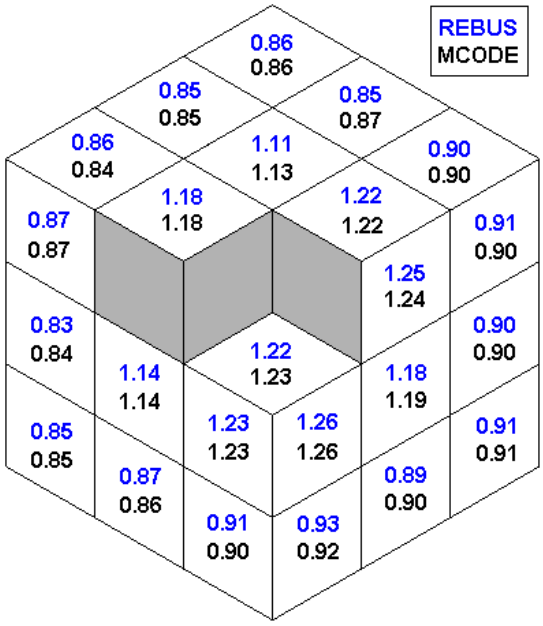

Figure 4-10 Element power peaking in HEU "equilibrium" core for REBUS and MCODE

Although the proposed equilibrium core models do not exactly mirror any given actual core configuration, these models appear to give an adequate representation of current operating conditions from which further evaluations can be made.

4.4 Summary

The MCODE fuel management wrapper has been shown to perform reliably based on a number of studies. The first step towards validating the new capabilities was to ensure that the small change made to MCODE to allow for fuel management operations did not fundamentally alter results given by MCODE. This was done by testing a pin-cell model that was distributed with MCODE. This test showed that even at high burnup, the results were not appreciably altered due to the change in MCODE.

With the MCODE change verified, the next step was to ensure that the fuel management wrapper was correctly performing fuel management operations. To test the functionality of the wrapper, a 24-element core with all fresh LEU fuel was burned for 640 days with the fuel being moved in the same pattern every 80 days. While the core starts as all fresh fuel, after 640 days, the material compositions should be close to their equilibrium values. The result of this run shows that the movement of the control blade with time is in agreement with what one would intuitively predict, moving out as the core is being depleted and moving in whenever new fuel is added. In addition, the concentration of ^{235}U and ^{239}Pu in each element, shown in Figure 4-6 and Figure 4-8, respectively, also are in agreement with our predictions. Fuel elements that have resided in the core for longer times have a lower concentration of ^{235}U and a higher concentration of ^{239}Pu.

While a pure LEU core has not been successfully modeled in REBUS to date, benchmarking done on an HEU model has shown good agreement with MCNP. It was thus possible to compare the fuel management capabilities of REBUS and the MCODE fuel management wrapper by testing an HEU model. This was done by running the same "equilibrium" refueling scheme for both REBUS and the MCODE fuel management wrapper. The representative refueling scheme included replacing depleted fuel with new fuel, shuffling fuel elements, and flipping fuel elements. The end-of-life element power peaking factors for the two models showed remarkable agreement as shown in Figure 4-10 despite being evaluated at different Xenon conditions.

5 Conclusions and Future Work

5.1 Conclusions

Until recently, the procedure for performing fuel management calculations for the MITR was based on a diffusion theory code developed in the 1970s [**17**]. With a need to perform detailed depletion-modeling of the MITR in support of an anticipated conversion from HEU to LEU fuel, a new fuel management wrapper for MCODE was developed herein to enable studies using full-core coupled Monte Carlo depletion. In addition to the fuel management wrapper, a graphical user interface was developed to automate the creation of input files for the fuel management wrapper. Together, these tools will help to simplify the onerous task of performing a detailed fuel management run in MCODE and analyzing the results from a run, all in a manner that is less prone to human error. This will enable the studies in support of the MITR conversion project to be carried out effectively and will greatly aid the MITR staff in performing routine in-core fuel management calculations after the conversion is complete.

In analyzing the time-dependent neutronic behavior of the MITR, there are at least three effects which are important to model for the sake of ensuring that the all applicable safety limits are met. These are:

- Changing material compositions due to irradiation and spatial rearrangement of the fuel;
- Movement of the control blades to make up for lost reactivity due to burnup; and
- Transient behavior.

The fuel management wrapper developed allows the first effect here, i.e. depletion of materials and fuel shuffling, to be modeled explicitly for a structured fuel management pattern or irregular fuel management operations. For reasons described in section 3.4, the movement of the control blade can have a vast effect on the power distribution in the core and thus a general criticality search algorithm was developed to model this effect. By having MCODE call the criticality search at each flux calculation, this ensures that the correct reaction rates are used to predict end-of-timestep material compositions.

Prior experience showed that even with a wrapper code to automate fuel management operations in MCODE, setting up the input file was still time-consuming [**32**]. The graphical user interface developed here makes setting up an input file for the fuel management wrapper simple and intuitive. The graphical user interface has been tested and shown to work on both Linux and Windows platforms and is expected to work on all other platforms that support Python and PyQt.

With the fuel management wrapper and graphical user interface developed, several tests were performed to ensure that the specified irradiation materials were being depleted correctly, that the desired fuel management operations were being modeled correctly, and that the control blade movement made sense physically. Modeling of an LEU equilibrium core showed that the fuel management wrapper performed reliably and, moreover, that it was possible to do detailed depletion studies using a Monte Carlo code without the run time being too limiting. A typical refueling scheme for an HEU core was modeled using

both the fuel management wrapper for MCODE as well as the REBUS-PC diffusion theory code. The results showed that power peaking results were consistent between the two codes.

5.2 Future Work

5.2.1 Code and Algorithmic Improvements

While many different codes exists that couple MCNP to ORIGEN for the purpose of performing depletion calculations, MCODE is, in the author's own biased opinion, one of the most versatile and easy-to-use. Nevertheless, there are a number of improvements that could be made to MCODE to improve the accuracy of the results as well as the execution speed. One planned improvement to MCODE is updating ORIGEN from version 2.2 to ORIGEN-S, the latest version included in the SCALE package [**35**]. ORIGEN-S has free-form input processing and is able to utilize fluxes and cross-sections in any multi-group energy structure.

One downside of coupled Monte Carlo depletion systems as currently implemented is that as the number of nuclides builds up through the depletion, the computational expense becomes more prohibitive due to the increased number of reaction rate tallies necessary. Recent work has shown that tallying a very fine-group flux and then using this flux to collapse the cross-sections into one-group cross sections rather than tallying the reaction rate in each nuclide can vastly improve the execution time [**36**]. While the prior work has used multi-group cross-sections to collapse the fine-group flux, studies will be performed at MIT to determine whether this method is feasible to implement using continuous-energy cross-sections.

There are also opportunities to improve the coupling between MCNP and a depletion module. Rather than using a predictor-corrector method to predict the change in material compositions over a timestep, some have suggested that one should assume that the reaction rates vary linearly over the timestep. Such a change would require modification to the depletion solution algorithm itself, as has been done with the MC21 Monte Carlo code [**37**].

One area which may offer promise for improved execution time is development of a better criticality search algorithm. The secant method described in section 3.4 entails performing several flux calculations in MCNP at different control blade heights before the actual flux calculation to determine the reaction rates is even performed. This can in many cases increase the run time by a factor of between two and five. Any improvement which would make the criticality search algorithm faster would be a worthwhile endeavor.

5.2.2 Graphical User Interface Improvements

Although the graphical user interface and the fuel management wrapper as currently written have a number of features and subroutines that are specific to the MITR, the long-term vision of this project is to have an interface that requires little code modification in order to be used for a variety of different reactor designs. This will require having a standardized reactor template format that the GUI would be able to load in. The most substantive change would be to make the dialog window for specifying fuel assembly parameters, shown in Figure 2-6, customized to the particular assembly design.

One simple improvement that may be useful for certain purposes would be to implement the ability to run a calculation wherein each cycle ends when the control device hits a specified stop point. Additional planned improvements to the GUI and fuel management wrapper include the ability to allow fission products in the fuel to decay between cycles (since refueling will take several days) and adding visual aids for indicating when fuel elements are flipped or rotated in the GUI.

5.2.3 Post-Processing Utilities

The graphical user interface and fuel management wrapper automate the front-end of a fuel management calculation for the MITR. It is also desirable to have automation on the back-end of a calculation when one needs to analyze the data from a run previously performed. Several utilities have already been developed for this purpose.

One such utility is a tool which can plot power distributions. The power distribution utility first tallies the fission reaction rate in each fuel plate and creates a comma-separated value (csv) file listing all the reaction rates where each column corresponds to a single fuel element. Then, a MATLAB script reads in the csv file and creates a plot with each fuel plate colored according to its fission reaction rate and displayed in its respective position. Several examples of plots produced using this utility are shown in Figure 4-7 and Figure 4-9.

A similar utility developed plots the total concentration of any specified nuclide throughout the core. Examples of these plots are shown in Figure 4-3 and Figure 4-8. However, there is still room for improving the ability to quickly perform analysis on the data from a run, e.g. automatically plotting control blade heights through a run, producing power distribution plots without the use of MATLAB, etc.

5.2.4 Optimizing Run Parameters

While the user is free to choose how coarse or how fine they desire the depletion mesh over a fuel element to be, there is a tradeoff between execution speed and modeling accuracy when choosing the mesh. To date, no studies have been performed to determine how coarse or how fine the depletion mesh needs to be to obtain reasonable results. It is suggested that sensitivity studies be performed to examine the effect of the fineness of the mesh on power peaking and other parameters of interest.

The manner in which the fuel management wrapper chooses timesteps for depletion is also arbitrary. Smaller timesteps are used at the beginning of a cycle when Xenon concentrations are rapidly changing to approach equilibrium, and longer timesteps are used once equilibrium Xenon has been achieved. However, because the size of the timesteps is arbitrary, it may be possible to utilize longer timesteps without adverse consequences on the accuracy of the results. Again, sensitivity studies would shed light on the appropriate length of timesteps.

5.2.5 Validation against Experimental Data

While the testing and comparisons outlined in chapter 4 are a good first step towards validating solutions obtained from the MCODE fuel management wrapper, further studies should be performed to verify to accuracy solutions for a wide variety of problems. While comparisons between REBUS-PC and MCODE have been made for HEU cores, the same has not been done yet for LEU cores due to aforementioned

difficulties in obtaining suitable cross-sections for LEU cores. Once REBUS-PC is able to model LEU cores correctly, comparisons should be made between MCODE and REBUS-PC solutions.

To date, there have been very few comparisons against experimental data for any of the neutronic models. The conversion of the fuel from HEU to LEU will provide opportunities to validate code predictions to experimentally measured data. Plans are underway to make flux and reactivity measurements when LEU fuel is introduced. These measurements can then be compared to values computed using the MCODE fuel management wrapper or REBUS-PC. Additionally, if the MCODE fuel management wrapper is to be used for routine in-core fuel management calculations, it should be validated against data from previously burned cores.

5.2.6 Fuel Conversion Studies

Once sensitivity studies have been performed to determine the optimal choice of depletion mesh and depletion timesteps and validation studies are complete, the fuel management wrapper and graphical user interface will be used in support of the MITR fuel conversion studies. One issue of particular concern is that the high density U-Mo LEU fuel that will be used to convert the MITR will have never been resident in a reactor other than for basic irradiation and materials testing. As a result, there is a desire to slowly introduce LEU fuel elements into the core over the course of several cycles rather than replacing all of the HEU fuel at once. All the mixed core configurations will need to be analyzed to ensure that all applicable safety limits are met during the transition from all HEU to all LEU. It may be necessary to perform optimization studies to ensure that these limits and others are met.

Appendix A: Fuel Management Wrapper Source Code

A.1 mcodeFM.py

```
#!/usr/bin/env python

from __future__ import division

import sys
import os
import subprocess
import subs
from math import *


__version__ = "0.8.4"

#-------------------------------------------------------------------------------
#---------------------------------- MAIN() -------------------------------------
#-------------------------------------------------------------------------------

def main():

    # Parse command line options
    from optparse import OptionParser
    usage = "usage: %prog [options] file"
    version = "MCODE-FM v{0}".format(__version__)
    p = OptionParser(usage=usage, version=version)
    p.add_option("-i", "--input-only", action="store_true", dest="inputOnly",
                 help="only create input files at each cycle",
                 default=False)
    (options, args) = p.parse_args()
    if not args or len(args) != 1:
        p.print_help()
        return
    filename = args[0]

    # get current working directory
    cwd = os.getcwd()

    # read in input from .run file
    (surfs, cells, mats) = subs.findSurfsCellsMats('skeleton')
    (data, paths) = subs.loadRun(filename, mats)
    (fixedTimes, nCycles, time, power) = data

    # Create universe for each path
    # -Note- This could be improved by automatically reading what universes are
    # present in the skeleton input file.
    universes = set([0,2,35,36,45,46,54,55,56,62])
    for path in paths:
        path.universe = subs.grabCard(universes)

    # loop over cycles:
    cycle = 0
    while cycle < len(time):
        directory = "{0}/cycle_{1}/".format(cwd,cycle+1)
        if not os.path.isdir(directory):
            os.mkdir(directory)
        os.chdir(directory)
        if cycle > 0:
            if not options.inputOnly:
                subs.updateMaterials(cwd, cycle, paths)
            days = time[cycle] - time[cycle-1]
        else:
```

```
            days = time[cycle]
        subs.makeInput(cwd, cycle, days, power[cycle], paths, surfs, cells)
        if not options.inputOnly:
            subprocess.call(["preproc input"], shell=True)
            subprocess.call(["mcode input"], shell=True)
            subprocess.call(["mcodeout input -fm"], shell=True)
        os.chdir(os.pardir)
        cycle += 1

if __name__ == "__main__":
    main()
```

A.2 subs.py

```
#!/usr/bin/env python

from __future__ import division

import os
import sys
import copy
import subprocess
from math import *
from data.classes import *
import data.fuelLocations as fuel


#-------------------------------------------------------------------------------
#------------------------------- SUBROUTINES -----------------------------------
#-------------------------------------------------------------------------------

def loadRun(file, mats):
    fh = None
    paths = []
    elements = Container()
    try:
        fh = open(file,'r')
        # Read run data
        fixedTimes = eval(fh.readline().split()[1])
        nCycles = eval(fh.readline().split()[1])
        times = [eval(i) for i in fh.readline().split()[1:]]
        power = [eval(i) for i in fh.readline().split()[1:]]
        # Start reading remainder of file
        while True:
            line = fh.readline()
            if not line: break
            words = line.split()
            if not words: continue
            # Read path data
            if words[0] == "path":
                locations = [word if word.isdigit() else fuel.translate[word]
                             for word in words[1:]]
                uid = fh.readline().split()[1]
                time = eval(fh.readline().split()[1])
                path = Path("", locations, time, uid)
                for word in fh.readline().split()[1:]:
                    path.flip.add(eval(word))
                for word in fh.readline().split()[1:]:
                    path.rotate.add(eval(word))
                paths.append(path)
            # Read element data
            if words[0] == 'element':
                uid = words[1]
                plates = eval(fh.readline().split()[1])
                axial = eval(fh.readline().split()[1])
                radial = [eval(i) for i in fh.readline().split()[1:]]
                element = Element("", plates, axial, radial, uid)
```

```
                while True:
                    position = fh.tell()
                    words = fh.readline().split()
                    if len(words) == 0:
                        break
                    if words[0] == 'material':
                        radial = eval(words[1])
                        axial = eval(words[2])
                        density = words[3]
                        nuclides = []
                        while True:
                            position = fh.tell()
                            words = fh.readline().split()
                            if len(words) == 0: break
                            if words[0] == 'material': break
                            for i in range(len(words)//2):
                                nuclides.append((words[2*i],words[2*i+1]))
                        fh.seek(position)
                        element.materials[(radial,axial)] = {
                            'density': density,
                            'nuclides': nuclides,
                            'cells': set()}
                elements.addItem(element)
                fh.seek(position)
    finally:
        if fh is not None:
            fh.close()
    print '{0:15}{1}'.format('Fixed Time: ',fixedTimes)
    print '{0:15}{1}'.format('Cycles:',nCycles)
    print '{0:15}{1}'.format('Times:',times)
    print '{0:15}{1}'.format('Power:',power)
    print '\n'
    for path in paths:
        element = elements.getItem(path.uid)
        path.plates = element.plates
        path.axial = element.axial
        path.radial = element.radial
        path.materials = copy.deepcopy(element.materials)
        for node in path.materials:
            if not path.materials[node].has_key('mat'):
                path.materials[node]['mat'] = grabCard(mats)
        del path.element
        print '{0:15}{1}'.format('Locations:',path.locations)
        print '{0:15}{1}'.format('Time:',path.time)
        print '{0:15}{1}'.format('Flip:',path.flip)
        print '{0:15}{1}'.format('Rotate:',path.rotate)
        print '{0:15}{1}'.format('# Plates:',path.plates)
        print '{0:15}{1}'.format('# Axial:',path.axial)
        print '{0:15}{1}'.format('Radial:',path.radial)
        print '{0:15}{1}'.format('Materials:',path.materials.keys())
        print '\n'
    data = (fixedTimes, nCycles, times, power)
    return data, paths


def findSurfsCellsMats(filename):
    paragraph = 1
    surfaces = set()
    cells = set()
    materials = set()
    for line in open(filename,'r'):
        line = line.rstrip()
        if line == '':
            paragraph += 1
            continue
        words = line.split()
        if words[0].isdigit() and paragraph == 1:
            cells.add(eval(words[0]))
        if words[0].isdigit() and paragraph == 2:
            surfaces.add(eval(words[0]))
        if words[0][0] == 'm' and words[0][1].isdigit() and paragraph == 3:
```

```
            materials.add(eval(words[0][1:]))
    for s in surfaces, cells, materials:
        s.add(0)
    for i in range(101,128):
        cells.add(i)
    return surfaces, cells, materials

def grabCard(set):
    a = min(set)
    b = max(set)
    while a <= b:
        if a not in set:
            break
        set.remove(a)
        a += 1
    set.add(a)
    return a

def makeInput(cwd, cycle, days, power, paths, surfs, cells):

    # Create set to keep track of added cells
    originalCells = cells.copy()
    originalSurfs = surfs.copy()

    # Check how many different axial types
    axialSet = set()
    for path in paths:
        axialSet.add(path.axial)

    # Create axial surface information
    axialSurfaces = {}
    z = 56.8325/2
    for n in axialSet:
        axialSurfaces[n] = [(grabCard(surfs), z - i*56.8325/n)
                            for i in range(n+1)]

    # Open files for reading/writing
    input = open("{0}/cycle_{1}/input".format(cwd,cycle+1),"w")
    skel = open("{0}/skeleton".format(cwd),"r")

    # Copy cells from skeleton
    line = skel.readline().rstrip()
    while line != '':
        input.write(line + "\n")
        line = skel.readline().rstrip()

    # Determine which paths are present in current cycle
    currentPaths = set()
    lookupUniverse = {}
    for path in paths:
        for i in range(len(path.locations)):
            if cycle == path.time + i:
                lookupUniverse[path.locations[i]] = path.universe
                currentPaths.add(path)
    for i in range(101,128):
        if str(i) not in lookupUniverse:
            lookupUniverse[str(i)] = 35

    # Write cells for fuel elements
    writeTitle(input,"FUEL ELEMENTS")
    importances = []
    steel = "6  6.0034-2   "
    water = "1 -0.9967837  "
    for path in currentPaths:
        i = 1
        addCell(input, path, cells, importances, 'water', 110, 120, i, i+1)
        i += 1
        for j in range(path.plates):
            addCell(input, path, cells, importances, 'steel', 110, 120, i, i+1)
            i += 1
            addCell(input, path, cells, importances, 'steel', 110, 112, i, i+1)
```

```
            addCell(input, path, cells, importances, 'fuel', 112, 118, i, i+1,
                    j, axialSurfaces)
            addCell(input, path, cells, importances, 'steel', 118, 120, i, i+1)
            i += 1
            addCell(input, path, cells, importances, 'steel', 110, 120, i, i+1)
            i += 1
            addCell(input, path, cells, importances, 'water', 110, 120, i, i+1)
            i += 1
        addCell(input, path, cells, importances, 'steel', 100, 110, 1, i)
        addCell(input, path, cells, importances, 'steel', 120, 130, 1, i)
        cell = grabCard(cells)
        input.write("{0!s:5} {1:41}     80 u={2}\n".format(
            cell, "30 9.07359-2", path.universe))
        importances.append("{0!s:8}{1:14}1".format(cell,"2.5300e-8"))
        cell = grabCard(cells)
        input.write("{0!s:5} {1:41} -85    u={2}\n".format(
            cell, "40 8.21151-2", path.universe))
        importances.append("{0!s:8}{1:14}1".format(cell,"2.5300e-8"))
        addCell(input, path, cells, importances, 'water', 100, 130, '', 1)
        addCell(input, path, cells, importances, 'water', 100, 130, i, '')
        addCell(input, path, cells, importances, 'water', '', 100, '', '')
        addCell(input, path, cells, importances, 'water', 130, '', '', '')
        input.write("c\n")

    # Write cells for fuel locations
    writeTitle(input, "FUEL LOCATIONS")
    for item in fuel.cards:
        input.write(item[0] + "\n")
        input.write("      {0}fill={1} {2}\n".format(
            item[1], lookupUniverse[item[0][0:3]], item[2]))
        importances.append("{0!s:8}{1:14}1".format(item[0][0:3],"2.5300e-8"))
    input.write("\n")

    # Copy surfaces from skeleton
    line = skel.readline().rstrip()
    while line != '':
        input.write(line + "\n")
        line = skel.readline().rstrip()

    # Write surfaces from axialSurfaces
    writeTitle(input, "FUEL SEGMENTATION")
    for item in axialSurfaces.values():
        for card in item:
            input.write("{0!s:8}pz    {1}\n".format(card[0],card[1]))
    input.write("\n")

    # Copy data from skeleton and include importances
    line = skel.readline().rstrip()
    while line != '':
        input.write(line + "\n")
        if line[0] == "#":
            for card in importances:
                input.write(card + "\n")
        line = skel.readline().rstrip()

    # If t = 0, write material for each element
    # else, copy material from previous cycle
    writeTitle(input, "MATERIALS")
    for path in currentPaths:
        for node in path.materials.values():
            input.write("m{0!s:5}".format(node['mat']))
            for i, nuclide in enumerate(node['nuclides']):
                if not i%3 and i:
                    input.write("\n      ")
                input.write("{0} {1}  ".format(nuclide[0].upper(), nuclide[1]))
            input.write("\n")
    input.write("\n")

    # Write tally materials
    writeTitle(input, "MCODE TALLY MATERIALS")
    for path in currentPaths:
```

```
    for key in path.materials:
        if path.plates == 18:
            thickness = 0.0508
        if path.plates == 15:
            thickness = 0.0762
        radial = path.radial
        volume = thickness * 5.223 * 56.8325 / path.axial
        if key[0] == 1:
            volume *= path.radial[0]
        else:
            volume *= path.radial[key[0]-1] - path.radial[key[0]-2]
        input.write("{0} {1} pwrue.lib 300 0.999 0 {2} {3}\n".format(
            path.materials[key]['mat'], volume, path.universe,
            1 if cycle > path.time else ""))

# Copy MCNP/ORIGEN options from skeleton
writeTitle(input, "MCNP/ORIGEN OPTIONS")
line = skel.readline().rstrip()
while line != '':
    input.write(line + "\n")
    line = skel.readline().rstrip()

# Write tally cards
writeTitle(input, "MCODE TALLIES")
for path in currentPaths:
    for node in path.materials.values():
        input.write("tal  {0} ({1})\n".format(
            node['mat'], " ".join([str(i) for i in node['cells']])))

# Write depletion options
writeTitle(input, "DEPLETION OPTIONS")
input.write("pow  {0:10.4e}\n".format(power))
input.write("nor  2 0\n")
input.write("cor  3\n")
input.write("mci  -1\n")
if days > 10.0:
    input.write("dep  D  1.0  1\n")
    input.write("        3.0  1\n")
    input.write("        10.0  1\n")
    time = 40.0
    while time < days:
        input.write("        {0}  1\n".format(time))
        time += 30.0
elif days > 3.0 and days <= 10.0:
    input.write("dep  D  1.0  1\n")
    input.write("        3.0  1\n")
elif days > 1.0 and days <= 3.0:
    input.write("dep  D  1.0  1\n")
input.write("        {0}  1\n".format(days))

# Close files
input.close()
skel.close()

# Write out material/cell data
file = "{0}/cycle_{1}/data".format(cwd, cycle+1)
fh = open(file, "w")
for path in currentPaths:
    fh.write("element {0} {1} {2}\n".format(
        path.locations[cycle-path.time], path.plates, path.axial))
    fh.write("{0}\n".format(path.radial))
    materials = {}
    for node in path.materials.keys():
        materials[node] = {'mat': path.materials[node]['mat'],
                           'cells': path.materials[node]['cells']}
    fh.write("{0}\n".format(materials))
fh.close()

# Reset cells and surfaces
toAdd = originalCells.difference(cells)
toRemove = cells.difference(originalCells)
```

```
    for cell in toRemove:
        cells.remove(cell)
    for cell in toAdd:
        cells.add(cell)
    toAdd = originalSurfs.difference(surfs)
    toRemove = surfs.difference(originalSurfs)
    for surface in toRemove:
        surfs.remove(surface)
    for surface in toAdd:
        surfs.add(surface)
    for path in currentPaths:
        for node in path.materials.values():
            node['cells'] = set()


def addCell(input, path, cells, imp, type, p1, p2, p3, p4,
            plate=None, surfs=None):
    # -Note- This currently only supports 15 or 18 plate elements
    if path.plates == 18:
        base = 51000
    elif path.plates == 15:
        base = 71000
    else:
        pass

    if type == "fuel":
        for node, num in enumerate(path.radial):
            if plate < num:
                radial = node+1
                break
        for axial in range(path.axial):
            cell = grabCard(cells)
            string = "{0!s:5} {1} {2} {3} {4} {5} {6} -{7} {8} u=-{9}\n".format(
                cell, path.materials[radial,axial+1]['mat'],
                path.materials[radial,axial+1]['density'],
                -(base+p1), base+p2, -(base+p3), base+p4,
                surfs[path.axial][axial][0], surfs[path.axial][axial+1][0],
                path.universe)
            path.materials[radial,axial+1]['cells'].add(cell)
            imp.append("{0!s:8}{1:14}1".format(cell,"3.2154e-8"))
            input.write(string)
        return

    cell = grabCard(cells)
    if type == "water":
        density = "1 -0.9967837   "
        imp.append("{0!s:8}{1:14}1".format(cell,"2.8277e-8"))
    elif type == "steel":
        density = "6  6.0034-2    "
        imp.append("{0!s:8}{1:14}1".format(cell,"3.2154e-8"))
    p1 = p1 if not p1 else str(-(base + p1))
    p2 = p2 if not p2 else str(base + p2)
    p3 = p3 if not p3 else str(-(base + p3))
    p4 = p4 if not p4 else str(base + p4)
    string = "{0!s:5} {1} {2:6} {3:5} {4:6} {5:5} -80 85 u=".format(
        cell, density, p1, p2, p3, p4)
    if p1 and p2 and p3 and p4:
        string += str(-path.universe)
    else:
        string += str(path.universe)
    input.write(string + "\n")

def writeTitle(input, title):
    input.write("c\n")
    input.write("c " + title + "\n")
    input.write("c\n")

def updateMaterials(cwd, cycle, paths):
    # Determine which paths need updating
    currentPaths = set()
    for path in paths:
```

```
        for i in range(1,len(path.locations)):
            if cycle == path.time + i:
                currentPaths.add(path)

    # Check which timestep to read from
    step = 1
    while True:
        file = "{0}/cycle_{1}/tmpdir11/mc{2:03d}".format(cwd,cycle,step)
        if not os.path.exists(file):
            file = oldfile
            step = step - 1
            break
        oldfile = file
        step += 1

    # Read nuclide data from mc###i
    paragraph = 1
    nuclides = {}
    fh = open(file + "i","r")   # Add try/finally with fh.close
    while True:
        line = fh.readline()
        if not line: break
        line = line.rstrip()
        if line == "":
            paragraph += 1
            continue
        words = line.split()
        if paragraph==3 and words[0][0]=="m" and words[0][1].isdigit():
            mat = eval(words[0][1:])
            nuclides[mat] = []
            for i in range((len(words)-1)//2):
                nuclides[mat].append((words[2*i+1],words[2*i+2]))
            while True:
                position = fh.tell()
                line = fh.readline()
                if len(line) < 5 or line[0:5].rstrip() != "":
                    fh.seek(position)
                    break
                words = line.split()
                for i in range(len(words)//2):
                    nuclides[mat].append((words[2*i],words[2*i+1]))

    # Update densities and copy PCH files
    from_dir = "{0}/cycle_{1}/tmpdir11/".format(cwd,cycle)
    to_dir = "{0}/cycle_{1}/tmpdir11/".format(cwd,cycle+1)
    if not os.path.isdir(to_dir):
        os.mkdir(to_dir)
    for path in currentPaths:
        for node in path.materials.values():
            node['nuclides'] = nuclides[node['mat']]
            node['density'] = sum([eval(i[1]) for i in node['nuclides']])
            from_file = from_dir + "m{0}_{1:03d}.PCH".format(node['mat'],step)
            to_file = to_dir + "m{0}_000.PCH".format(node['mat'])
            subprocess.call(["cp {0} {1}".format(from_file,to_file)], shell=True)
```

A.3 fileIO.py

```
#!/usr/bin/env python

from __future__ import division

from math import *


#-------------------------------------------------------------------------------
#------------------------------ SUBROUTINES ------------------------------------
#-------------------------------------------------------------------------------
```

```
def loadMCNPdata(filename):

    # Set up data structures
    density = {}
    materials = {}
    paragraph = 1
    continueReading = False

    # Begin reading MCNP file
    for line in open(filename,"r"):

        # Check for new paragraph and split line into words
        line = line.rstrip()
        if line == "":
            paragraph += 1
            continue
        words = line.split()

        # Obtain material densities from cell cards
        try:
            if paragraph == 1 and words[0][0].lower() != "c":
                density[words[1]] = words[2]
        except IndexError: pass

        # If continuation line, add nuclides
        if continueReading and line[:5].isspace():
            for i in range(0,len(words),2):
                if words[i][0] == "$":
                    break
                materials[num]["nuclides"].append((words[i],words[i+1]))
            continue
        continueReading = False

        # Read nuclides on first line of material
        try:
            if (paragraph == 3 and words[0][0] == "m" and
                words[0][1] != "t"):
                if len(words) % 2 != 1: raise InvalidMaterialError
                num = words[0][1:]
                if num in density:
                    materials[num] = {"density": density[num], "nuclides": []}
                else:
                    materials[num] = {"density": "", "nuclides": []}
                for i in range(1,len(words),2):
                    if words[i][0] == "$":
                        break
                    materials[num]["nuclides"].append((words[i],words[i+1]))
                continueReading = True
        except IndexError: pass
    return materials
```

A.4 data/fuelLocations.py

```
#!/usr/bin/env python

cards = [("101   0  -740   750  -650   660   -65   100","*",
          "(3.54592 0 0 30 120 90 60 30 90 90 90 0)"),
         ("102   0  -550   560  -750   760   -65   100","*",
          "(-1.7729619 -3.07086 0 30 -60 90 120 30 90 90 90 0)"),
         ("103   0  -540   550  -640   650   -65   100","",
          "(-1.7729619 3.07086 0 0 1 0 -1 0 0 0 0 1)"),
         ("104   0  -555   565  -665   670   -65   100","",
          "(9.22409 -3.69316 0 0 1 0 -1 0 0 0 0 1)"),
         ("105   0  -565   570  -665   670   -65   100","",
          "(5.67817 -9.83488 0 0 1 0 -1 0 0 0 0 1)"),
         ("106   0  -565   570  -655   665   -65   100","",
```

```
            "(-1.413678 -9.83488 0 0 1 0 -1 0 0 0 0 1)"),
           ("107    0 -635   645  -765   770   -65   100","*",
            "(-7.810416 -6.14172 0 -30 240 90 60 -30 90 90 90 0)"),
           ("108    0 -630   635  -765   770   -65   100","*",
            "(-11.35634 0 0 -30 240 90 60 -30 90 90 90 0)"),
           ("109   0  -630   635  -755   765   -65   100","*",
            "(-7.810416 6.14172 0 -30 240 90 60 -30 90 90 90 0)"),
           ("110   0  -735   745  -530   535   -65   100","*",
            "(-1.413678 9.83488 0 210 120 90 300 210 90 90 90 0)"),
           ("111   0  -730   735  -530   535   -65   100","*",
            "(5.67817 9.83488 0 210 120 90 300 210 90 90 90 0)"),
           ("112   0  -730   735  -535   545   -65   100","*",
            "(9.22409 3.69316 0 210 120 90 300 210 90 90 90 0)"),
           ("113   0  -555   565  -670   675   -65   100","",
            "(16.315941 -3.69316 0 0 1 0 -1 0 0 0 0 1)"),
           ("114   0  -565   570  -670   675   -65   100","",
            "(12.770018 -9.83488 0 0 1 0 -1 0 0 0 0 1)"),
           ("115   0  -570   575  -670   675   -65   100","",
            "(9.22409 -15.9766 0 0 1 0 -1 0 0 0 0 1)"),
           ("116   0  -570   575  -665   670   -65   100","",
            "(2.132246 -15.9766 0 0 1 0 -1 0 0 0 0 1)"),
           ("117   0  -570   575  -655   665   -65   100","",
            "(-4.959601 -15.9766 0 0 1 0 -1 0 0 0 0 1)"),
           ("118   0  -770   775  -635   645   -65   100","*",
            "(-11.35634 -12.28344 0 -30 240 90 60 -30 90 90 90 0)"),
           ("119   0  -770   775  -630   635   -65   100","*",
            "(-14.902264 -6.14172 0 -30 240 90 60 -30 90 90 90 0)"),
           ("120   0  -770   775  -625   630   -65   100","*",
            "(-18.448188 0 0 -30 240 90 60 -30 90 90 90 0)"),
           ("121   0 -765   770  -625   630   -65   100","*",
            "(-14.902264 6.14172 0 -30 240 90 60 -30 90 90 90 0)"),
           ("122   0  -755   765  -625   630   -65   100","*",
            "(-11.35634 12.28344 0 -30 240 90 60 -30 90 90 90 0)"),
           ("123    0 -735   745  -525   530   -65   100","*",
            "(-4.959601 15.9766 0 210 120 90 300 210 90 90 90 0)"),
           ("124   0  -730   735  -525   530   -65   100","*",
            "(2.132246 15.9766 0 210 120 90 300 210 90 90 90 0)"),
           ("125   0  -725   730  -525   530   -65   100","*",
            "(9.22409 15.9766 0 210 120 90 300 210 90 90 90 0)"),
           ("126   0  -725   730  -530   535   -65   100","*",
            "(12.770018 9.83488 0 210 120 90 300 210 90 90 90 0)"),
           ("127   0  -725   730  -535   545   -65   100","*",
            "(16.315941 3.69316 0 210 120 90 300 210 90 90 90 0)")]

translate = {"A1": "101", "A2": "102", "A3": "103", "B1": "104", "B2": "105",
             "B3": "106", "B4": "107", "B5": "108", "B6": "109", "B7": "110",
             "B8": "111", "B9": "112", "C1": "113", "C2": "114", "C3": "115",
             "C4": "116", "C5": "117", "C6": "118", "C7": "119", "C8": "120",
             "C9": "121", "C10": "122", "C11": "123", "C12": "124",
             "C13": "125", "C14": "126", "C15": "127"}
```

A.5 data/classes.py

```
#!/usr/bin/env python

import uuid

class Container(object):
    def __init__(self):
        self.items = {}

    def getItem(self, uid):
        return self.items.get(uid)

    def addItem(self, item):
        self.items[item.uid] = item
```

```
    def removeItem(self, item):
        del self.items[item.uid]
        del item

    def __iter__(self):
        for item in self.items.values():
            yield item


class RunData(object):
    def __init__(self):
        self.fixedTimes = True
        self.nCycles = 1
        self.time = []
        self.power = 5e6
        self.powerFraction = []


class Path(object):
    def __init__(self, name, locations=None, time=None, uid=None):
        if uid:
            self.uid = uid
        else:
            self.uid = str(uuid.uuid4())
        self.name = name
        self.locations = locations if locations else []
        self.flip = set()
        self.rotate = set()
        self.time = time
        self.element = None

    def __len__(self):
        return len(self.locations)


class Element(object):
    def __init__(self, name, plates=18, axial=1, radial=None, uid=None):
        self.name = name
        if uid:
            self.uid = uid
        else:
            self.uid = str(uuid.uuid4())
        self.plates = plates
        self.axial = axial
        self.radial = radial if radial else []
        self.materials = {}

    def nodes(self):
        groups = [range(self.radial[0])]
        for i in range(len(self.radial)-1):
            groups.append(range(self.radial[i],self.radial[i+1]))
        return groups


class Material(object):
    def __init__(self, name, density="1.0", nuclides=None, uid=None):
        self.name = name
        if uid:
            self.uid = uid
        else:
            self.uid = str(uuid.uuid4())
        self.density = density
        self.nuclides = nuclides if nuclides else []

    def __len__(self):
        return len(self.nuclides)
```

A.5 utils/keffsearch.py

```
#!/usr/bin/env python

import sys, os, re
import shutil
import textwrap
import subprocess
from math import *

#-------------------------------------------------------------------------------
#---------------------------------- CLASSES ------------------------------------
#-------------------------------------------------------------------------------

class search:
    def __init__(self, value, paragraph, find, change):
        self.value = str(value)
        self.paragraph = paragraph
        self.find = find
        self.change = int(change)

#-------------------------------------------------------------------------------
#---------------------------------- MAIN() -------------------------------------
#-------------------------------------------------------------------------------

def main(mcnpFile, controlFile, options=None):
    if options:
        tolerance = options.tolerance
        useOriginal = options.useOriginal
    DW = (0.0027, 0.0)

    # Correct for "inp="
    if mcnpFile[0:4] == "inp=":
        mcnpFile = mcnpFile[4:]

    # Check if mcnp and control device input files exist
    if not os.path.isfile(mcnpFile):
        raiseError('%s does not exist.' % (mcnpFile))
        return False
    if not os.path.isfile(controlFile):
        raiseError('%s does not exist.' % (controlFile))
        return False

    # Read control device input file
    searchList = []
    try:
        for line in open(controlFile,'r'):
            words = line.strip().split()

            # Material density (single cell)
            if words[0] == 'cell':
                searchList.append(search(words[1],1,1,3))

            # Surface value
            elif words[0] == 'surface':
                searchList.append(search(words[1],2,1,words[2]))

            # Material density (all cells)
            elif words[0] == 'material' and len(words) == 2:
                searchList.append(search(words[1],1,2,3))
                allCells = True

            # Material composition
            elif words[0] == 'material' and len(words) == 3:
                searchList.append(search(words[1]))

            # General search
            elif words[0] == 'general':
                searchList.append(search(words[1],words[2],words[3],words[4]))
```

```
        elif words[0] == 'range':
            controlMin = words[1]
            controlMax = words[2]
except:
    value = sys.exc_info()[1]
    raiseError('Invalid control device specification.')
    print('\n' + line)
    return False

# Check for invalid input
paragraphs = set()
for item in searchList:
    paragraphs.add(item.paragraph)
if len(paragraphs) > 1:
    raiseError('Multiple paragraphs specified in control device input \
    file. This may be due to specifying a cell and a surface \
    simultaneously.')
    return False
positions = set()
for item in searchList:
    position = getPosition(mcnpFile,item)
    positions.add(position)
    if not position:
        raiseError('Control device position not found.')
        print('\nSearch for %s in position %i of paragraph %i'
              % (item.value, item.find, item.paragraph))
        print('---> Return position %i' % (item.change))
        return False
if len(positions) > 1:
    raiseError('Control device position must be defined by a single number')
    print('\nPositions found:')
    for pos in positions: print('  %s' % (str(pos)))
    return False

# Move control device only option
if options.moveDevice:
    fh = open('position','r')
    words = fh.readline().split()
    fh.close()
    moveControlDevice(mcnpFile, searchList, words[0])
    return

# Make a copy of the input file if necessary
if not useOriginal:
    filecopy = 'srch'
    shutil.copy(mcnpFile, filecopy)
    mcnpFile = filecopy

# Change number of cycles
defaultCycles = 80
oldCycles = changeCycles(mcnpFile, defaultCycles)

# keff search
writeOutput('*'*19 + ' Beginning keff Search ' + '*'*19, lines=2)
z_new = eval(position)
converged = False
cycles = defaultCycles
i = 1
while not converged:
    moveControlDevice(mcnpFile, searchList, z_new)
    if cycles == defaultCycles:
        writeOutput('i=%i   position=%6.3f   ' % (i,z_new), lines=0)
    runMCNP(mcnpFile, cycles)
    k_new = getKeff('srchm')
    # converged if keff within tolerance
    if (k_new[0] + k_new[1] < 1 + tolerance and
        k_new[0] - k_new[1] > 1 - tolerance):
        converged = True
        writeOutput('cycles=%3i   ' % (cycles) +
                    'keff = %.4f +/- %.4f' % k_new, lines=2)
    else:
```

```
            if i > 1:
                DW = ((k_new[0] - k_old[0])/(z_new - z_old),
                      sqrt((k_new[1]**2 + k_old[1]**2)/(z_new - z_old)**4))
                # Is DW too small or unc too large
                if DW[0] < 0.0010 or 2*DW[1] > DW[0]:
                    DW = (0.0020, 0.0)
            # Estimate critical position
            z_crit = (z_new + (1 - k_new[0])/DW[0],
                      sqrt((k_new[1]/DW[0])**2 + ((k_new[0]-1)*DW[1]/DW[0]**2)**2))
            # Adaptive batching algorithm
            if 2*z_crit[1] < abs(z_crit[0] - z_new):
                writeOutput('cycles=%3i   ' % (cycles) +
                            'keff = %.4f +/- %.4f' % k_new)
                k_old = k_new
                z_old = z_new
                z_new = z_crit[0]
                i += 1
                cycles = defaultCycles
            else:
                cycles = cycles + 30
    fh = open('position','w')
    fh.write("%s\n" % (z_new))
    fh.close()
    subprocess.call(['rm srchr srctq'], shell=True, stderr=open('/dev/null','w'))

#-------------------------------------------------------------------------------
#------------------------------- SUBROUTINES -----------------------------------
#-------------------------------------------------------------------------------

def replaceWord(sentence, index, newWord):
    p = re.compile(r'(\s*)((\S+\s+){' + str(index-1) + r',' \
                   + str(index-1) + r'})(\S+)(.*)')
    return p.sub(r'\g<1>\g<2>' + newWord + r'\g<5>', sentence)

def getPosition(file, search):
    paragraph = 1
    fh = open(file,'r')
    fh.seek(0)
    try:
        for line in fh:
            line = line.rstrip()
            if line == '': paragraph += 1
            words = line.split()
            try:
                if (paragraph == search.paragraph and
                    words[search.find-1] == search.value):
                    return words[search.change-1]
            except IndexError: pass
    finally:
        if fh is not None:
            fh.close()
    return None

# MOVECONTROLDEVICE moves the control device based on searchList to -position-
#     which is specified as a float.

def moveControlDevice(file, searchList, position):
    paragraph = 1
    newfile = []
    for line in open(file,'r'):
        line = line.rstrip()
        if line == '': paragraph += 1
        words = line.split()
        for item in searchList:
            try:
                if (paragraph == item.paragraph and
                    words[item.find-1] == item.value):
                    line = replaceWord(line, item.change, str(position))
            except IndexError: pass
        newfile.append(line + '\n')
    f = open(file,'w')
```

```
    f.writelines(newfile)

def changeCycles(file, cycles):
    newfile = []
    for line in open(file,'r'):
        words = line.split()
        try:
            if words[0] == 'kcode':
                oldCycles = words[4]
                line = replaceWord(line, 5, str(cycles))
        except IndexError: pass
        newfile.append(line)
    f = open(file,'w')
    f.writelines(newfile)
    return oldCycles

def runMCNP(file, cycles):
    if cycles > 80:
        continueFile = open('inpcont','w')
        continueFile.writelines(['continue\n','kcode 3J %i\n' % cycles])
        continueFile.close()
        subprocess.call(['rm srcho srchm'], shell=True,
                        stderr=open('/dev/null','w'))
        subprocess.call(['mcnp', 'cn', 'i=inpcont', 'o=srcho',
                         'r=srchr', 'm=srchm', 'tasks 6'])
        subprocess.call(['rm inpcont srctq'], shell=True,
                        stderr=open('/dev/null','w'))
    else:
        subprocess.call(['rm srch?'], shell=True, stderr=open('/dev/null','w'))
        subprocess.call(['mcnp', 'i=' + file, 'o=srcho', 'r=srchr',
                         'm=srchm', 'tasks 6'])
    # Need a way of checking that the mcnp run completed successfully

def writeOutput(str, lines=1):
    output = open('output_file','a')
    str += '\n'*lines
    output.write(str)
    output.flush()

# GETKEFF retrieves the final keff and uncertainty from an MCNP run as recorded
#     in the mctal output file.

def getKeff(mctal):
    words = open(mctal,'r').readlines()[-2].split()
    return (eval(words[1]), eval(words[2]))

def raiseError(msg):
    message = ' '.join(('ERROR: '+msg).split())
    for line in textwrap.wrap(message, width=80, subsequent_indent=' '*7):
        print(line)

#-------------------------------------------------------------------------------
#---------------------------- COMMAND LINE OPTIONS -----------------------------
#-------------------------------------------------------------------------------

if __name__ == '__main__':
    from optparse import OptionParser
    usage = 'usage: %prog [options] mcnpfile'
    p = OptionParser(usage=usage)
    p.add_option('-c', '--control', action='store', dest='controlFile',
                 default='control_input', metavar='FILE',
                 help='read control device definition from FILE')
    p.add_option('-o', '--original', action='store_true', dest='useOriginal',
                 default=False, help='use original file instead of a copy')
    p.add_option('-t', '--tolerence', action='store', dest='tolerance',
                 default=0.002, metavar='TOL', type='float',
                 help='specify tolerence, TOL, on keff')
    p.add_option('-m', '--move-device', action='store_true', dest='moveDevice',
                 default=False, help='move control device only')
    (options, args) = p.parse_args()
    if not args:
```

```
    p.print_help()
else:
    main(args[0], options.controlFile, options)
```

Appendix B: Graphical User Interface Source Code

B.1 interface.pyw

```
#!/usr/bin/env python

import sys
from PyQt4.QtCore import *
from PyQt4.QtGui import *
from gui.mainWindow import MainWindow

if __name__ == "__main__":
        app = QApplication(sys.argv)
        form = MainWindow()
        form.show()
        app.exec_()
```

B.2 gui/mainWindow.py

```
#!/usr/bin/env python

from __future__ import division

import pickle
import platform
import os

from math import *
from mcodeFM import __version__
from data.classes import *
import fileIO

from PyQt4.QtCore import *
from PyQt4.QtGui import *
from gui.listElementDialog import ListElementDialog
from gui.listMaterialDialog import ListMaterialDialog
from gui.editPathDialog import EditPathDialog
from gui.editTimeDialog import EditTimeDialog
import gui.geometry as geometry


class MainWindow(QMainWindow):

    def __init__(self, parent=None):
        super(MainWindow, self).__init__(parent)
        self.setWindowTitle("MCODE Fuel Management Interface")

#-------------------------------------------------------------------------------
#------------------------- INTERFACE INITIALIZATION ----------------------------
#-------------------------------------------------------------------------------

        # Create tab widget and associated tabs
        self.main = QWidget()

        # Create time selection
        timeLabel = QLabel("Select Time:")
        self.timeComboBox = QComboBox()
        self.timeComboBox.setMinimumContentsLength(25)
        self.timeButton = QPushButton("Edit Times")
        topLayout = QHBoxLayout()
        topLayout.addWidget(timeLabel)
```

```
topLayout.addStretch()
topLayout.addWidget(self.timeComboBox)
topLayout.addWidget(self.timeButton)

# Create summary path view
self.scene = QGraphicsScene(self)
self.scene.setSceneRect(-250,-250,500,500)
self.view = QGraphicsView()
self.view.setRenderHint(QPainter.Antialiasing)
self.view.setMinimumSize(500,500)
self.view.setHorizontalScrollBarPolicy(Qt.ScrollBarAlwaysOff)
self.view.setVerticalScrollBarPolicy(Qt.ScrollBarAlwaysOff)
self.view.setScene(self.scene)
self.addPathButton = QPushButton("Add Path")
self.editPathButton = QPushButton("Edit Path")
self.removePathButton = QPushButton("Remove Path")
buttonLayout = QVBoxLayout()
buttonLayout.addWidget(self.addPathButton)
buttonLayout.addWidget(self.editPathButton)
buttonLayout.addWidget(self.removePathButton)
buttonLayout.addStretch()
viewLayout = QHBoxLayout()
viewLayout.addWidget(self.view)
viewLayout.addLayout(buttonLayout)

# Setup main widget
line = QFrame(self.main)
line.setFrameShape(QFrame.HLine)
line.setFrameShadow(QFrame.Sunken)
layout = QVBoxLayout()
layout.addLayout(topLayout)
layout.addWidget(line)
layout.addLayout(viewLayout)
self.main.setLayout(layout)
self.setCentralWidget(self.main)

# Add menu items
self.menubar = QMenuBar(self)
self.menuFile = QMenu("&File",self.menubar)
self.menuEdit = QMenu("&Edit",self.menubar)
self.menuHelp = QMenu("&Help",self.menubar)
self.setMenuBar(self.menubar)
self.menubar.addActions([self.menuFile.menuAction(),
                         self.menuEdit.menuAction(),
                         self.menuHelp.menuAction()])

self.actionNewMITR = QAction("&New MITR Run",self)
self.actionNewRun = QAction("New Run from Template...",self)
self.actionNewRun.setDisabled(True)
self.actionOpenRun = QAction("&Open Existing Run...",self)
self.actionSaveRun = QAction("&Save Run...",self)
self.actionExportRun = QAction("&Export Run...",self)
self.actionExit = QAction("E&xit",self)
self.menuFile.addActions([self.actionNewMITR,self.actionNewRun,
                          self.actionOpenRun,self.actionSaveRun,
                          self.actionExportRun, self.actionExit])
self.menuFile.insertSeparator(self.actionExit)
self.actionElements = QAction("&Elements",self)
self.actionMaterials = QAction("&Materials",self)
self.menuEdit.addActions([self.actionElements,self.actionMaterials])
self.actionAbout = QAction("&About",self)
self.menuHelp.addActions([self.actionAbout])

# Menu Signals
self.connect(self.actionNewMITR, SIGNAL("triggered()"), self.newMitrRun)
self.connect(self.actionOpenRun, SIGNAL("triggered()"), self.openRun)
self.connect(self.actionSaveRun, SIGNAL("triggered()"), self.saveRun)
self.connect(self.actionExportRun, SIGNAL("triggered()"), self.exportRun)
self.connect(self.actionExit, SIGNAL("triggered()"), self.close)
self.connect(self.actionElements, SIGNAL("triggered()"), self.editElements)
self.connect(self.actionMaterials, SIGNAL("triggered()"), self.editMaterials)
```

```
        self.connect(self.actionAbout, SIGNAL("triggered()"), self.about)

        # Run Editor Signals
        self.connect(self.timeButton, SIGNAL("clicked()"), self.editTimes)
        self.connect(self.scene, SIGNAL("selectionChanged()"), self.refreshView)
        self.connect(self.timeComboBox, SIGNAL("currentIndexChanged(int)"),
                     self.drawLocations)
        self.connect(self.addPathButton, SIGNAL("clicked()"), self.addPath)
        self.connect(self.editPathButton, SIGNAL("clicked()"), self.editPath)
        self.connect(self.removePathButton, SIGNAL("clicked()"), self.removePath)

        # Perform initial loading
        QTimer.singleShot(0, self.initialLoad)

    def initialLoad(self):
        self.run = RunData()
        self.paths = Container()
        self.elements = Container()
        self.materials = Container()

        self.populateTimes()
        self.drawLocations()
        self.refreshView()


#-------------------------------------------------------------------------------
#---------------------------- RUN EDITOR FUNCTIONS  ----------------------------
#-------------------------------------------------------------------------------


    def editTimes(self):
        form = EditTimeDialog(self.run)
        index = self.timeComboBox.currentIndex()
        if form.exec_():
            self.populateTimes()
            self.timeComboBox.setCurrentIndex(index)

    def populateTimes(self):
        self.timeComboBox.clear()
        if self.run.fixedTimes:
            if self.run.time:
                for i, time in enumerate(self.run.time):
                    self.timeComboBox.addItem(
                        "Cycle {0}: {1:.1f} days - {2:.1f} days".format(
                        i+1, 0 if i==0 else self.run.time[i-1], time))
            else:
                self.timeComboBox.addItem("Cycle 1: 0.0 days -")
        else:
            self.timeComboBox.addItems(
                ["Cycle "+str(item+1) for item in range(self.run.nCycles)])

    def drawLocations(self):
        self.scene.clear()
        self.locations = []
        time = self.timeComboBox.currentIndex()
        for i, vals in enumerate(geometry.MITR_pairs):
            location = geometry.Location(
                geometry.MITR_ids[i], geometry.MITR_points, vals[0], vals[1])
            location.setAcceptHoverEvents(True)
            for path in self.paths:
                if time - path.time < 0: continue
                try:
                    currentLocation = path.locations[time-path.time]
                    if currentLocation == location.uid:
                        location.hasPath = True
                        label = self.scene.addText(path.name)
                        index = geometry.MITR_ids.index(currentLocation)
                        pair = geometry.MITR_pairs[index]
                        label.translate(pair[0][0], pair[0][1])
                        label.translate(geometry.MITR_offset[pair[1]][0],
                                        geometry.MITR_offset[pair[1]][1])
```

```
                    label.setZValue(1)
                except IndexError: pass
            self.locations.append(location)
            self.scene.addItem(location)
            location.refresh()

    def refreshView(self):
        for location in self.locations:
            location.refresh()
        if self.scene.selectedItems():
            self.addPathButton.setEnabled(True)
            self.editPathButton.setEnabled(True)
            self.removePathButton.setEnabled(True)
        else:
            self.addPathButton.setDisabled(True)
            self.editPathButton.setDisabled(True)
            self.removePathButton.setDisabled(True)


    def addPath(self):
        location = self.scene.selectedItems()[-1]
        if location.hasPath:
            QMessageBox.warning(self, "Can't Add Path", "This location "
                                "already has a path for this cycle.")
            return
        time = self.timeComboBox.currentIndex()
        path = Path("{0} ({1:0d})".format(location.uid, time), [location.uid], time)
        form = EditPathDialog(path, self.elements, self.materials, self.run.time)
        if form.exec_():
            self.paths.addItem(path)
            self.drawLocations()

    def editPath(self):
        location = self.scene.selectedItems()[-1]
        time = self.timeComboBox.currentIndex()
        for path in self.paths:
            if (time-path.time < 0 or
                time-path.time > len(path.locations)-1):
                continue
            if path.locations[time-path.time] == location.uid:
                form = EditPathDialog(path, self.elements,
                                      self.materials, self.run.time)
                if form.exec_():
                    pass
        self.drawLocations()

    def removePath(self):
        if (QMessageBox.question(self, "Remove Path",
                                 "Remove selected path(s)?",
                                 QMessageBox.Yes|QMessageBox.No) ==
            QMessageBox.No):
            return
        for location in self.scene.selectedItems():
            time = self.timeComboBox.currentIndex()
            for path in self.paths:
                try:
                    if path.locations[time-path.time] == location.uid:
                        self.paths.removeItem(path)
                except IndexError: pass
        self.drawLocations()


#---------------------------------------------------------------------------
#---------------------------- MENU FUNCTIONS -------------------------------
#---------------------------------------------------------------------------


    def saveChanges(self):
        msgBox = QMessageBox(self)
        msgBox.setWindowTitle("Save Changes?")
        msgBox.setIcon(QMessageBox.Question)
```

```
        msgBox.setText("The run has been modified.")
        msgBox.setInformativeText("Do you want to save your changes?")
        msgBox.setStandardButtons(QMessageBox.Save | QMessageBox.Discard |
                                  QMessageBox.Cancel)
        msgBox.setDefaultButton(QMessageBox.Save)
        choice = msgBox.exec_()
        if choice == QMessageBox.Cancel:
            return False
        elif choice == QMessageBox.Save:
            if not self.saveRun():
                return False
        return True

    def newMitrRun(self):
        if self.saveChanges():
            self.initialLoad()

    def openRun(self):
        filename = QFileDialog.getOpenFileName(self, "Load MCODE Run", "./",
                                               "MCODE Runs (*.run)")
        fh = None
        if not filename.isEmpty():
            try:
                fh = open(filename, "rb")
                self.run = pickle.load(fh)
                self.paths = pickle.load(fh)
                self.elements = pickle.load(fh)
                self.materials = pickle.load(fh)
                for element in self.elements:
                    for node in element.materials:
                        for material in self.materials:
                            if element.materials[node].uid == material.uid:
                                del element.materials[node]
                                element.materials[node] = material
            finally:
                if fh is not None:
                    fh.close()
            self.populateTimes()
            self.drawLocations()

    def saveRun(self):
        filename = QFileDialog.getSaveFileName(self, "Save MCODE Run", "./",
                                               "MCODE Runs (*.run)")
        fh = None
        try:
            if filename[-4] != '.':
                filename += '.run'
            fh = open(filename, "wb")
            pickle.dump(self.run, fh)
            pickle.dump(self.paths, fh)
            pickle.dump(self.elements, fh)
            pickle.dump(self.materials, fh)
            return True
        except:
            return False
        finally:
            if fh is not None:
                fh.close()

    def exportRun(self):
        # Check if all paths have elements assigned
        for path in self.paths:
            if not path.element:
                QMessageBox.critical(self, "Elements not assigned!", "All paths "
                                     "do not have fuel elements assigned.")
                return

        # Export run
        filename = QFileDialog.getSaveFileName(
            self, "Save MCODE-FM Input", "./", "MCODE-FM Input (*)")
        fh = None
```

```
        elementsSaved = set()
        list2str = lambda list: " ".join([str(i) for i in list])
        try:
            fh = open(filename,"w")
            # Write run data
            fh.write("fixed %s\n" % self.run.fixedTimes)
            fh.write("nCycles %s\n" % self.run.nCycles)
            fh.write("time " + list2str(self.run.time) + "\n")
            fh.write("power " + list2str(
                [self.run.power * i for i in self.run.powerFraction]) + "\n")
            fh.write("\n")
            # Write path data
            for path in self.paths:
                fh.write("path " + list2str(path.locations) + "\n")
                fh.write("element %s\n" % path.element.uid)
                fh.write("time %s\n" % path.time)
                fh.write("flip " + list2str(path.flip) + "\n")
                fh.write("rotate " + list2str(path.rotate) + "\n\n")
                # Write element for path if not written
                if path.element not in elementsSaved:
                    fh.write("element %s\n" % path.element.uid)
                    fh.write("plates %s\n" % path.element.plates)
                    fh.write("axial %s\n" % path.element.axial)
                    fh.write("radial " + list2str(path.element.radial) + "\n")
                    for i in range(len(path.element.radial)):
                        for j in range(path.element.axial):
                            mat = path.element.materials[(i+1,j+1)]
                            fh.write("material %s %s %s\n" % (
                                i+1, j+1, mat.density))
                            for k in range(len(mat.nuclides)):
                                fh.write(list2str(mat.nuclides[k]))
                                fh.write("\n" if (k+1) % 4 == 0 else " ")
                            if (k+1) % 4 != 0: fh.write("\n")
                    elementsSaved.add(path.element)
                    fh.write("\n")
        finally:
            if fh is not None:
                fh.close()

    def editElements(self):
        form = ListElementDialog(self.elements,self.materials)
        if form.exec_():
            pass

    def editMaterials(self):
        form = ListMaterialDialog(self.materials,self.elements)
        if form.exec_():
            pass

    def about(self):
        QMessageBox.about(self, "About MCODE Fuel Management Interface",
                          """<b>MCODE Fuel Management Interface</b> v %s
                          <p>Copyright &copy; 2009 Paul K. Romano.
                          All Rights Reserved.
                          <p>Python %s -- Qt %s -- PyQt %s on %s""" %
                          (__version__, platform.python_version(),QT_VERSION_STR,
                           PYQT_VERSION_STR, platform.system()))

    def close(self):
        if self.saveChanges():
            QMainWindow.close(self)
```

B.3 gui/listElementDialog.py

```
#!/usr/bin/env python

import os
```

```
from PyQt4.QtCore import *
from PyQt4.QtGui import *
from gui.editElementDialog import EditElementDialog

from data.classes import *
import fileIO


class ListElementDialog(QDialog):

    def __init__(self, elements, materials, parent=None):
        super(ListElementDialog, self).__init__(parent)

        self.elements = elements
        self.materials = materials

        # Create widgets
        self.elementsTree = QTreeWidget()
        self.elementsTree.setColumnCount(1)
        self.elementsTree.setHeaderLabels(["Fuel Elements"])
        self.elementsTree.setSelectionMode(
            QAbstractItemView.ExtendedSelection)
        self.addElementButton = QPushButton("Add Element")
        self.editElementButton = QPushButton("Edit Element")
        self.removeElementButton = QPushButton("Remove Element")
        self.loadElementButton = QPushButton("Load Element...")
        okButton = QDialogButtonBox(QDialogButtonBox.Ok)

        # Create layout
        buttonLayout = QVBoxLayout()
        buttonLayout.addWidget(self.addElementButton)
        buttonLayout.addWidget(self.editElementButton)
        buttonLayout.addWidget(self.removeElementButton)
        buttonLayout.addWidget(self.loadElementButton)
        buttonLayout.addStretch()
        buttonLayout.addWidget(okButton)
        layout = QHBoxLayout()
        layout.addWidget(self.elementsTree)
        layout.addLayout(buttonLayout)
        self.setLayout(layout)

        # Create Signals
        self.connect(self.addElementButton, SIGNAL("clicked()"),
                     self.addElement)
        self.connect(self.editElementButton, SIGNAL("clicked()"),
                     self.editElement)
        self.connect(self.removeElementButton, SIGNAL("clicked()"),
                     self.removeElement)
        self.connect(self.loadElementButton, SIGNAL("clicked()"),
                     self.loadElement)
        self.connect(okButton, SIGNAL("accepted()"), self.accept)


        # Perform initial loading
        self.setWindowTitle("Elements")
        self.populateElements()

#-------------------------------------------------------------------------------
#---------------------- FUEL ELEMENT EDITOR FUNCTIONS --------------------------
#-------------------------------------------------------------------------------


    def populateElements(self, selectedElement = None):
        # Sort elements by name
        sortedNames = []
        sortedElements = []
        for element in self.elements:
            sortedNames.append(str(element.name))
        sortedNames.sort()
        for name in sortedNames:
```

```
            for element in self.elements:
                if element.name == name:
                    sortedElements.append(element)
        # Create item for each element
        selected = None
        self.elementsTree.clear()
        for element in sortedElements:
            item = QTreeWidgetItem(self.elementsTree, [element.name])
            item.setData(0, Qt.UserRole, QVariant(QString(element.uid)))
            if selectedElement is not None and selectedElement == element.uid:
                selected = item
        if selected is not None:
            selected.setSelected(True)
            self.elementsTree.setCurrentItem(selected)

    def currentElement(self):
        item = self.elementsTree.currentItem()
        if item is None:
            return None
        return self.elements.getItem(str(item.data(0,Qt.UserRole).toString()))

    def addElement(self):
        element = Element("Enter name here", radial=[1,4,14,17,18])
        form = EditElementDialog(element, self.materials)
        if form.exec_():
            self.elements.addItem(element)
            self.populateElements()

    def editElement(self):
        element = self.currentElement()
        if element is not None:
            form = EditElementDialog(element, self.materials)
            if form.exec_():
                self.populateElements(element.uid)

    def removeElement(self):
        if (QMessageBox.question(self, "Remove Element",
                                 "Remove selected element(s)?",
                                 QMessageBox.Yes|QMessageBox.No) ==
            QMessageBox.No):
            return
        for item in self.elementsTree.selectedItems():
            element = self.elements.getItem(
                str(item.data(0,Qt.UserRole).toString()))
            self.elements.removeItem(element)
        self.populateElements()

    def loadElement(self):
        filename = str(QFileDialog.getOpenFileName(
            self, "Load datafile", "./", "Datafiles (data)"))
        if not filename:
            return

        # Load element data from 'data'
        fh = open(filename,"r")
        line = fh.readline()
        currentElements = set()
        while line != '':
            words = line.split()
            (location, plates, axial) = (words[1], eval(words[2]), eval(words[3]))
            radial = eval(fh.readline())
            data = eval(fh.readline())
            name = "{0}_{1}".format(filename,location)
            newElement = Element(name,plates,axial,radial)
            newElement.data = data
            for node in newElement.data:
                newElement.materials[node] = []
            currentElements.add(newElement)
            self.elements.addItem(newElement)
            line = fh.readline()
        fh.close()
```

```
        # Determine whether to use beginning or end of cycle data
        useEnd = QMessageBox.question(self, "Beginning or end?",
                                      "Use beginning or end of cycle data?",
                                      "Beginning","End")
        directory = os.path.dirname(filename)
        step = 0
        if useEnd:
            while True:
                file = "{0}/tmpdir11/mc{1:03d}".format(directory,step)
                if not os.path.exists(file):
                    file = oldfile
                    step = step - 1
                    break
                oldfile = file
                step += 1

        # Load material data from 'mc###i'
        file = "{0}/tmpdir11/mc{1:03d}i".format(directory,step)
        loadMat = fileIO.loadMCNPdata(file)
        for element in currentElements:
            for node in element.materials:
                num = str(element.data[node]['mat'])
                # -Note- Material name should be more descriptive
                material = Material("{0}{1:s}".format(id(element),node),
                                    loadMat[num]['density'],
                                    loadMat[num]['nuclides'])
                element.materials[node] = material
        self.populateElements()
```

B.4 gui/listMaterialDialog.py

```
#!/usr/bin/env python

from __future__ import division

import os

from PyQt4.QtCore import *
from PyQt4.QtGui import *
from gui.editMaterialDialog import *

from data.classes import *
import fileIO


GLOBAL, ELEMENT = ["Global Materials", "Fuel Element Materials"]


class ListMaterialDialog(QDialog):

    def __init__(self, materials, elements, parent=None):
        super(ListMaterialDialog, self).__init__(parent)

        self.elements = elements
        self.materials = materials

        # Create widgets
        self.materialsTree = QTreeWidget()
        self.materialsTree.setColumnCount(1)
        self.materialsTree.setHeaderLabels(["Materials"])
        self.materialsTree.setSelectionMode(
            QAbstractItemView.ExtendedSelection)
        self.materialsTree.setMinimumHeight(400)
        self.addMatButton = QPushButton("Add Material")
        self.editMatButton = QPushButton("Edit Material")
        self.removeMatButton = QPushButton("Remove Material")
```

```
        self.loadMatButton = QPushButton("Load Material...")
        self.groupMatButton = QPushButton("Group Materials")
        okButton = QDialogButtonBox(QDialogButtonBox.Ok)

        # Create layout
        buttonLayout = QVBoxLayout()
        buttonLayout.addWidget(self.addMatButton)
        buttonLayout.addWidget(self.editMatButton)
        buttonLayout.addWidget(self.removeMatButton)
        buttonLayout.addWidget(self.loadMatButton)
        buttonLayout.addStretch()
        buttonLayout.addWidget(okButton)
        layout = QHBoxLayout()
        layout.addWidget(self.materialsTree)
        layout.addLayout(buttonLayout)
        self.setLayout(layout)

        # self.actionLoadMDB = QAction("Load Materials Database...",self)
        # self.actionSaveMDB = QAction("Save Materials Database...",self)
        # self.menuFile.addActions([self.actionLoadMDB,self.actionSaveMDB])
        # self.connect(self.actionLoadMDB, SIGNAL("triggered()"), self.loadMaterialDB)
        # self.connect(self.actionSaveMDB, SIGNAL("triggered()"), self.saveMaterialDB)

        # Create Signals
        self.connect(self.addMatButton, SIGNAL("clicked()"),
                     self.addMaterial)
        self.connect(self.editMatButton, SIGNAL("clicked()"),
                     self.editMaterial)
        self.connect(self.removeMatButton, SIGNAL("clicked()"),
                     self.removeMaterial)
        self.connect(self.loadMatButton, SIGNAL("clicked()"),
                     self.loadMaterial)
        self.connect(okButton, SIGNAL("accepted()"), self.accept)

        self.setWindowTitle("Materials")
        self.populateMaterials()

    def populateMaterials(self, selectedMaterial=None):
        elementSet = set()
        globalSet = set()
        for element in self.elements:
            for material in element.materials.values():
                elementSet.add(material)
        for material in self.materials:
            globalSet.add(material)

        elementSet = elementSet.symmetric_difference(
            elementSet.intersection(globalSet))
        self.elementDict = {}
        self.globalDict = {}
        for material in elementSet:
            self.elementDict[material.uid] = material
        for material in globalSet:
            self.globalDict[material.uid] = material

        # Sort materials by name
        sortedGlobalNames = []
        sortedGlobalMaterials = []
        sortedElementNames = []
        sortedElementMaterials = []
        for material in self.globalDict.values():
            sortedGlobalNames.append(str(material.name))
        sortedGlobalNames.sort()
        for name in sortedGlobalNames:
            for material in self.globalDict.values():
                if material.name == name:
                    sortedGlobalMaterials.append(material)
        for material in self.elementDict.values():
            sortedElementNames.append(str(material.name))
        sortedElementNames.sort()
        for name in sortedElementNames:
```

```
            for material in self.elementDict.values():
                if material.name == name:
                    sortedElementMaterials.append(material)

        # Create item for each material
        selected = None
        self.materialsTree.clear()
        parent = QTreeWidgetItem(self.materialsTree, [GLOBAL])
        for material in sortedGlobalMaterials:
            item = QTreeWidgetItem(parent, [material.name])
            item.setData(0, Qt.UserRole, QVariant(material.uid))
            self.materialsTree.expandItem(parent)
        parent = QTreeWidgetItem(self.materialsTree, [ELEMENT])
        for material in sortedElementMaterials:
            item = QTreeWidgetItem(parent, [material.name])
            item.setData(0, Qt.UserRole, QVariant(material.uid))
            self.materialsTree.expandItem(parent)

    def currentMaterial(self):
        item = self.materialsTree.currentItem()
        if item is None:
            return None
        if item.parent().text(0) == GLOBAL:
            return self.materials.getItem(
                str(item.data(0,Qt.UserRole).toString()))
        if item.parent().text(0) == ELEMENT:
            return self.elementDict[str(item.data(0,Qt.UserRole).toString())]
        return None

    def addMaterial(self):
        newMaterial = Material("Enter name here")
        form = EditMaterialDialog(newMaterial)
        if form.exec_():
            self.materials.addItem(newMaterial)
            self.populateMaterials()

    def editMaterial(self):
        material = self.currentMaterial()
        if material is not None:
            form = EditMaterialDialog(material)
            if form.exec_():
                self.populateMaterials(material.uid)

    def removeMaterial(self):
        if (QMessageBox.question(self, "Remove Material",
                                 "Remove selected material(s)?",
                                 QMessageBox.Yes|QMessageBox.No) ==
            QMessageBox.No):
            return
        for item in self.materialsTree.selectedItems():
            try:
                material = self.materials.getItem(
                    str(item.data(0,Qt.UserRole).toString()))
                self.materials.removeItem(material)
            except KeyError: pass
        self.populateMaterials()

    def loadMaterial(self):
        filename = str(QFileDialog.getOpenFileName(
            self, "Open MCNP File", "./", "All Files (*)"))
        file = os.path.basename(filename)
        loadMat = fileIO.loadMCNPdata(filename)
        for mat in loadMat:
            newMaterial = materials.Material(file + " - " + mat,
                                             loadMat[mat]['density'],
                                             loadMat[mat]['nuclides'])
            self.materials.addItem(newMaterial)
        self.populateMaterials()

    def loadMaterialDB(self):
        filename = QFileDialog.getOpenFileName(self, "Load Material Database", "./",
```

```
                                                "Material Databases (*.mmd)")
        fh = None
        if not filename.isEmpty():
            try:
                fh = open(filename, "rb")
                materialDB = cPickle.load(fh)
                self.materials = materials.MaterialContainer()
                for material in materialDB:
                    self.materials.addItem(material)
            finally:
                if fh is not None:
                    fh.close()
                self.populateMaterials()
        self.tabWidget.setCurrentWidget(self.materialsEditor)

    def saveMaterialDB(self):
        filename = QFileDialog.getSaveFileName(self, "Save Material Database", "./",
                                                "Material Databases (*.mmd)")
        fh = None
        try:
            if filename[-4] != '.':
                filename += '.mmd'
            fh = open(filename, "wb")
            cPickle.dump(self.materials, fh)
        finally:
            if fh is not None:
                fh.close()
```

B.4 gui/editElementDialog.py

```
#!/usr/bin/env python

from __future__ import division

from math import *
from data.classes import *

from PyQt4.QtCore import *
from PyQt4.QtGui import *
from gui.editMaterialDialog import *
import gui.geometry as geometry


length = 280
width = length*cos(pi/6)

class EditElementDialog(QDialog):

    def __init__(self, element, materials, parent=None):
        super(EditElementDialog, self).__init__(parent)

        self.element = element
        self.materials = materials

        # Create view
        self.scene = QGraphicsScene(self)
        self.scene.setSceneRect(-150,-225,300,450)
        self.view = QGraphicsView()
        self.view.setRenderHint(QPainter.Antialiasing)
        self.view.setMinimumSize(300,450)
        self.view.setHorizontalScrollBarPolicy(Qt.ScrollBarAlwaysOff)
        self.view.setVerticalScrollBarPolicy(Qt.ScrollBarAlwaysOff)
        self.view.setDragMode(QGraphicsView.RubberBandDrag)
        self.view.setRubberBandSelectionMode(Qt.IntersectsItemShape)
        self.view.setScene(self.scene)

        # Create mesh options
```

```
nameLabel = QLabel("Name:")
self.nameLineEdit = QLineEdit()
self.nameLineEdit.setAlignment(Qt.AlignHCenter)
nameLayout = QHBoxLayout()
nameLayout.addWidget(nameLabel)
nameLayout.addWidget(self.nameLineEdit)
platesLabel = QLabel("Number of Plates:")
self.platesComboBox = QComboBox()
self.platesComboBox.addItems(["15","18"])
axialLabel = QLabel("Axial Nodes:")
self.axialSpinBox = QSpinBox()
self.axialSpinBox.setMinimum(1)
self.axialSpinBox.setValue(self.element.axial)
optionsLayout = QHBoxLayout()
optionsLayout.addWidget(platesLabel)
optionsLayout.addWidget(self.platesComboBox)
optionsLayout.addWidget(axialLabel)
optionsLayout.addWidget(self.axialSpinBox)

# Create node/material selection
hline = QFrame(self)
hline.setFrameShape(QFrame.HLine)
hline.setFrameShadow(QFrame.Sunken)
nodeLabel = QLabel("Current Node:")
self.nodeComboBox = QComboBox()
materialLabel = QLabel("Current Material:")
self.materialComboBox = QComboBox()
self.materialComboBox.setMinimumContentsLength(20)
gridLayout = QGridLayout()
gridLayout.addWidget(nodeLabel,0,0)
gridLayout.addWidget(self.nodeComboBox,0,1)
gridLayout.addWidget(materialLabel,1,0)
gridLayout.addWidget(self.materialComboBox,1,1)
self.groupButton = QPushButton("Group Radial")
self.ungroupButton = QPushButton("Ungroup Radial")
self.editMaterialButton = QPushButton("Edit Material")
self.allMaterialButton = QPushButton("Apply Material to All")
buttonLayout = QGridLayout()
buttonLayout.addWidget(self.groupButton,0,0)
buttonLayout.addWidget(self.ungroupButton,1,0)
buttonLayout.addWidget(self.editMaterialButton,0,1)
buttonLayout.addWidget(self.allMaterialButton,1,1)
buttonBox = QDialogButtonBox(QDialogButtonBox.Ok|QDialogButtonBox.Cancel)
sideLayout = QVBoxLayout()
sideLayout.addLayout(nameLayout)
sideLayout.addLayout(optionsLayout)
sideLayout.addWidget(hline)
sideLayout.addLayout(gridLayout)
sideLayout.addLayout(buttonLayout)
sideLayout.addStretch()
sideLayout.addWidget(buttonBox)

# Create overall layout
layout = QHBoxLayout()
layout.addWidget(self.view)
layout.addLayout(sideLayout)
self.setLayout(layout)

# Set connections
self.connect(self.platesComboBox,
             SIGNAL("currentIndexChanged(int)"), self.platesChanged)
self.connect(self.groupButton, SIGNAL("clicked()"), self.groupSelected)
self.connect(self.ungroupButton, SIGNAL("clicked()"), self.ungroupSelected)
self.connect(self.scene, SIGNAL("selectionChanged()"), self.update)
self.connect(self.axialSpinBox, SIGNAL("valueChanged(int)"), self.axialChanged)
self.connect(self.nodeComboBox, SIGNAL("activated(int)"),
             self.axialChanged)
self.connect(self.materialComboBox, SIGNAL("activated(int)"),
             self.changeMaterial)
self.connect(self.editMaterialButton, SIGNAL("clicked()"),
             self.editMaterial)
```

```
        self.connect(self.allMaterialButton, SIGNAL("clicked()"),
                     self.applyMaterialToAll)
        self.connect(buttonBox, SIGNAL("accepted()"), self.accept)
        self.connect(buttonBox, SIGNAL("rejected()"), self.reject)

        self.setWindowTitle("Edit Fuel Element")
        self.initialLoad()

    def initialLoad(self):
        self.axialNode = 1

        index = self.platesComboBox.findText(QString(str(self.element.plates)))
        self.platesComboBox.setCurrentIndex(index)
        self.createPlates()

        localSet = set()
        self.materialComboBox.addItems(["None","New material..."])
        for material in self.element.materials.values():
            localSet.add(material)
        for material in self.materials:
            localSet.add(material)
        self.localMaterials = Container()
        for material in localSet:
            self.localMaterials.addItem(material)
            self.materialComboBox.addItem(
                material.name, QVariant(material.uid))

        self.nameLineEdit.setText(self.element.name)

    def createPlates(self):
        self.scene.clear()
        self.plates = {}
        points = [(0,0), (width,length/2),
                  (width,length/2-length/self.element.plates),
                  (0,-length/self.element.plates)]
        position = length/2*sin(pi/6)
        for i in range(self.element.plates):
            plate = geometry.Location(i, points, (-width/2, position), 0)
            plate.setAcceptHoverEvents(True)
            self.plates[i] = plate
            self.scene.addItem(plate)
            position -= length/self.element.plates
        self.update()

    def update(self):
        node = 0
        colors = QColor.colorNames()
        for i in range(self.element.plates):
            if i+1 > self.element.radial[node]:
                node += 1
            self.plates[i].defaultColor = QColor(colors[node+16])
            self.plates[i].refresh()

        nodes = self.selectedNodes()
        if nodes:
            low = nodes[0]
            high = nodes[-1]
            if low == high:
                rangeString = str(low) + "R, "
            else:
                rangeString = str(low) + "-" + str(high) + "R, "
            if self.element.materials.has_key((low,self.axialNode)):
                index = self.materialComboBox.findText(
                    self.element.materials[(low,self.axialNode)].name)
                self.materialComboBox.setCurrentIndex(index)
            else:
                self.materialComboBox.setCurrentIndex(0)
        else:
            rangeString = ""
            self.materialComboBox.setCurrentIndex(0)
        self.nodeComboBox.clear()
```

```
        self.nodeComboBox.addItems([rangeString + str(i+1) + "A"
                                    for i in range(self.element.axial)])
        self.nodeComboBox.setCurrentIndex(self.axialNode-1)

    def selectedNodes(self):
        nodes = set()
        if self.scene.selectedItems():
            for plate in self.scene.selectedItems():
                for node, group in enumerate(self.element.nodes()):
                    if plate.uid in group: nodes.add(node)
        return [i+1 for i in list(nodes)]

    def groupSelected(self):
        if self.scene.selectedItems():
            index = [plate.uid for plate in self.scene.selectedItems()]
            low = min(index)
            high = max(index)
            for node in self.element.radial[::-1]:
                if low <= node and node-1 <= high:
                    self.element.radial.remove(node)
            if low > 0: self.element.radial.append(low)
            self.element.radial.append(high+1)
            self.element.radial.sort()
            self.update()

    def ungroupSelected(self):
        if self.scene.selectedItems():
            for plate in self.scene.selectedItems():
                index = plate.uid
                if index not in self.element.radial and index > 1:
                    self.element.radial.append(index)
                if index+1 not in self.element.radial:
                    self.element.radial.append(index+1)
            self.element.radial.sort()
            self.update()

    def platesChanged(self):
        self.element.plates = self.platesComboBox.currentText().toInt()[0]
        for node in self.element.radial[::-1]:
            if node >= self.element.plates: self.element.radial.remove(node)
        self.element.radial.append(self.element.plates)
        self.element.radial.sort()
        self.createPlates()

    def axialChanged(self):
        self.element.axial = self.axialSpinBox.value()
        self.axialNode = self.nodeComboBox.currentIndex()+1
        self.update()

    def changeMaterial(self):
        index = self.materialComboBox.currentIndex()
        if index == 0:
            for node in self.selectedNodes():
                self.element.materials.pop((node,self.axialNode),None)
        else:
            if index == 1:
                material = Material("Enter name here")
                form = EditMaterialDialog(material)
                if not form.exec_():
                    del material
                    return
                self.localMaterials.addItem(material)
                self.materialComboBox.addItem(
                    material.name, QVariant(material.uid))
            else:
                uid = str(self.materialComboBox.itemData(index).toString())
                material = self.localMaterials.getItem(uid)
            for node in self.selectedNodes():
                self.element.materials[(node,self.axialNode)] = material
        self.update()
```

```
    def editMaterial(self):
        index = self.materialComboBox.currentIndex()
        if index > 1:
            uid = str(self.materialComboBox.itemData(index).toString())
            material = self.localMaterials.getItem(uid)
            form = EditMaterialDialog(material)
            if form.exec_():
                self.localMaterials.addItem(material)
                self.materialComboBox.setItemText(index,material.name)
                self.update()

    def applyMaterialToAll(self):
        if (QMessageBox.question(self, "Apply Material",
                                 "Apply Material to all Nodes?",
                                 QMessageBox.Yes|QMessageBox.No) ==
            QMessageBox.No):
            return
        index = self.materialComboBox.currentIndex()
        if index > 1:
            uid = str(self.materialComboBox.itemData(index).toString())
            material = self.localMaterials.getItem(uid)
            for axial in range(self.element.axial):
                for radial in range(len(self.element.radial)):
                    self.element.materials[(radial+1,axial+1)] = material
            self.update()

    def accept(self):
        self.element.axial = self.axialSpinBox.value()
        self.element.plates = self.platesComboBox.currentText().toInt()[0]
        self.element.name = self.nameLineEdit.text()
        QDialog.accept(self)
```

B.6 gui/editMaterialDialog.py

```
#!/usr/bin/env python

from PyQt4.QtCore import *
from PyQt4.QtGui import *
from data.classes import *

class EditMaterialDialog(QDialog):

    def __init__(self, material=None, parent=None):
        super(EditMaterialDialog, self).__init__(parent)

        self.material = material
        self.nuclideTable = QTableWidget()
        self.nuclideTable.setColumnCount(2)
        self.nuclideTable.setHorizontalHeaderLabels(
            ["Nuclide", "Atom/Weight Fraction"])

        nameLabel = QLabel("Material Name:")
        self.nameLineEdit = QLineEdit()
        self.nameLineEdit.setAlignment(Qt.AlignHCenter)
        densityLabel = QLabel("MCNP Density:")
        self.densityLineEdit = QLineEdit()
        self.densityLineEdit.setAlignment(Qt.AlignHCenter)
        addNuclideButton = QPushButton("&Add Nuclide")
        removeNuclideButton = QPushButton("&Remove Nuclide")
        okButton = QPushButton("&Ok")

        gridLayout = QGridLayout()
        gridLayout.addWidget(nameLabel,0,0)
        gridLayout.addWidget(self.nameLineEdit,0,1)
        gridLayout.addWidget(densityLabel,1,0)
        gridLayout.addWidget(self.densityLineEdit,1,1)
        buttonLayout = QHBoxLayout()
        buttonLayout.addWidget(addNuclideButton)
        buttonLayout.addWidget(removeNuclideButton)
```

```
        buttonLayout.addStretch()
        buttonLayout.addWidget(okButton)
        layout = QVBoxLayout()
        layout.addLayout(gridLayout)
        layout.addWidget(self.nuclideTable)
        layout.addLayout(buttonLayout)
        self.setLayout(layout)

        self.connect(addNuclideButton, SIGNAL("clicked()"), self.addNuclide)
        self.connect(removeNuclideButton, SIGNAL("clicked()"), self.removeNuclide)
        self.connect(okButton, SIGNAL("clicked()"), self.accept)
        self.connect(self.nuclideTable, SIGNAL("cellChanged(int,int)"),
                     self.tableItemChanged)

        self.setWindowTitle("Material Definition")
        QTimer.singleShot(0, self.initialLoad)

    def initialLoad(self):
        self.nameLineEdit.setText(self.material.name)
        self.nameLineEdit.selectAll()
        self.densityLineEdit.setText(self.material.density)
        self.populateTable()

    def populateTable(self):
        self.nuclideTable.clearContents()
        self.nuclideTable.setRowCount(len(self.material.nuclides))
        for row in range(len(self.material.nuclides)):
            self.nuclideTable.setItem(row, 0, QTableWidgetItem(
                str(self.material.nuclides[row][0])))
            self.nuclideTable.setItem(row, 1, QTableWidgetItem(
                str(self.material.nuclides[row][1])))
        self.nuclideTable.resizeColumnsToContents()

    def tableItemChanged(self, row, column):
        row = self.nuclideTable.currentRow()
        column = self.nuclideTable.currentColumn()
        if column == 0:
            self.material.nuclides[row] = (
                str(self.nuclideTable.currentItem().text()),
                self.material.nuclides[row][1])
        elif column == 1:
            self.material.nuclides[row] = (
                self.material.nuclides[row][0],
                str(self.nuclideTable.currentItem().text()))
        self.nuclideTable.resizeColumnsToContents()


    def addNuclide(self):
        self.material.nuclides.append(('','1.0'))
        self.populateTable()
        self.nuclideTable.setFocus()
        self.nuclideTable.setCurrentCell(self.nuclideTable.rowCount()-1,0)
        self.nuclideTable.editItem(self.nuclideTable.currentItem())

    def removeNuclide(self):
        index = self.nuclideTable.currentRow()
        self.material.nuclides.pop(index)
        self.populateTable()

    def accept(self):
        self.material.name = self.nameLineEdit.text()
        self.material.density = self.densityLineEdit.text()
        QDialog.accept(self)
```

B.7 gui/editPathDialog.py

```
#!/usr/bin/env python

from __future__ import division
```

```
from PyQt4.QtCore import *
from PyQt4.QtGui import *
import gui.geometry as geometry
from gui.editElementDialog import EditElementDialog

from data.classes import Element

class EditPathDialog(QDialog):

    def __init__(self, path, elements, materials, time, parent=None):
        super(EditPathDialog, self).__init__(parent)

        self.path = path
        self.elements = elements
        self.materials = materials
        self.time = time

        # Create side panel
        nameLabel = QLabel("Path Name:")
        self.nameLineEdit = QLineEdit()
        self.nameLineEdit.setAlignment(Qt.AlignHCenter)
        elementLabel = QLabel("Element:")
        self.elementComboBox = QComboBox()
        self.pathList = QListWidget()
        self.pathList.setSelectionMode(QAbstractItemView.ExtendedSelection)
        self.upButton = QPushButton("Up")
        self.removeButton = QPushButton("X")
        self.downButton = QPushButton("Down")
        gridLayout = QGridLayout()
        gridLayout.addWidget(nameLabel,0,0)
        gridLayout.addWidget(self.nameLineEdit,0,1)
        gridLayout.addWidget(elementLabel,1,0)
        gridLayout.addWidget(self.elementComboBox,1,1)
        buttonLayout = QHBoxLayout()
        buttonLayout.addWidget(self.upButton)
        buttonLayout.addWidget(self.removeButton)
        buttonLayout.addWidget(self.downButton)
        sideLayout = QVBoxLayout()
        sideLayout.addLayout(gridLayout)
        sideLayout.addWidget(self.pathList)
        sideLayout.addLayout(buttonLayout)

        # Create view and top layout
        self.scene = QGraphicsScene(self)
        self.scene.setSceneRect(-250,-250,500,500)
        self.view = QGraphicsView()
        self.view.setRenderHint(QPainter.Antialiasing)
        self.view.setMinimumSize(500,500)
        self.view.setHorizontalScrollBarPolicy(Qt.ScrollBarAlwaysOff)
        self.view.setVerticalScrollBarPolicy(Qt.ScrollBarAlwaysOff)
        self.view.setScene(self.scene)
        topLayout = QHBoxLayout()
        topLayout.addLayout(sideLayout)
        topLayout.addWidget(self.view)

        # Create lower buttons and layout
        self.flipButton = QPushButton("Flip Element")
        self.rotateButton = QPushButton("Rotate Element")
        buttonBox = QDialogButtonBox(QDialogButtonBox.Ok|QDialogButtonBox.Cancel)
        lowerLayout = QHBoxLayout()
        lowerLayout.addWidget(self.flipButton)
        lowerLayout.addWidget(self.rotateButton)
        lowerLayout.addWidget(buttonBox)
        line = QFrame(self)
        line.setFrameShape(QFrame.HLine)
        line.setFrameShadow(QFrame.Sunken)
        layout = QVBoxLayout()
        layout.addLayout(topLayout)
        layout.addWidget(line)
        layout.addLayout(lowerLayout)
```

```
        self.setLayout(layout)

        # Set connections
        self.connect(self.scene, SIGNAL("selectionChanged()"), self.addLocation)
        self.connect(self.upButton, SIGNAL("clicked()"), self.moveLocationUp)
        self.connect(self.removeButton, SIGNAL("clicked()"), self.removeLocation)
        self.connect(self.downButton, SIGNAL("clicked()"), self.moveLocationDown)
        self.connect(self.elementComboBox, SIGNAL("activated(int)"),
                     self.changeElement)
        self.connect(self.flipButton, SIGNAL("clicked()"), self.flipElement)
        self.connect(self.rotateButton, SIGNAL("clicked()"), self.rotateElement)
        self.connect(buttonBox, SIGNAL("accepted()"), self.accept)
        self.connect(buttonBox, SIGNAL("rejected()"), self.reject)


        self.setWindowTitle("Edit Path")
        self.initialLoad()

    def initialLoad(self):
        self.nameLineEdit.setText(self.path.name)
        self.nameLineEdit.selectAll()
        # Sort elements by name
        sortedNames = []
        sortedElements = []
        for element in self.elements:
            sortedNames.append(str(element.name))
        sortedNames.sort()
        for name in sortedNames:
            for element in self.elements:
                if element.name == name:
                    sortedElements.append(element)
        # Create item for each element
        self.elementComboBox.addItems(["None","New element..."])
        for element in sortedElements:
            self.elementComboBox.addItem(
                element.name, QVariant(element.uid))
        # Create locations
        self.locations = {}
        for i, vals in enumerate(geometry.MITR_pairs):
            uid = geometry.MITR_ids[i]
            location = geometry.Location(uid, geometry.MITR_points, vals[0], vals[1])
            location.setAcceptHoverEvents(True)
            self.locations[uid] = location
            self.scene.addItem(location)
        self.update()

    def update(self):
        # Update list of locations in path
        self.pathList.clear()
        n = len(self.path)
        for location in self.locations.values():
            location.defaultColor = Qt.lightGray
        for i, id in enumerate(self.path.locations):
            self.locations[id].defaultColor = QColor(i/n*255,(n-i)/n*255,0)
            try:
                itemString = "{0} days - {1} days ---> {2}".format(
                    0.0 if i==0 and self.path.time==0 else
                    self.time[self.path.time+i-1],
                    self.time[self.path.time+i], id)
            except IndexError:
                itemString = "No Time Specified ---> %s" % (id)
            if i in self.path.flip:
                if i in self.path.rotate:
                    itemString += " (Flip,Rotate)"
                else:
                    itemString += " (Flip)"
            else:
                if i in self.path.rotate:
                    itemString += " (Rotate)"
            self.pathList.addItem(itemString)
        for location in self.locations.values():
```

```
            location.refresh()

        # Update list of elements
        if self.path.element: # Change to finding uid
            index = self.elementComboBox.findText(self.path.element.name)
            self.elementComboBox.setCurrentIndex(index)
            self.elementComboBox.setPalette(QPalette())
        else:
            self.elementComboBox.setCurrentIndex(0)
            self.elementComboBox.setPalette(QPalette(Qt.red))

    def addLocation(self):
        if self.scene.selectedItems():
            location = self.scene.selectedItems()[-1]
            self.path.locations.append(location.uid)
            self.update()

    def removeLocation(self):
        toDelete = []
        for item in self.pathList.selectedItems():
            self.pathList.setCurrentItem(item)
            toDelete.append(self.pathList.currentRow())
        toDelete.sort()
        toDelete.reverse()
        for index in toDelete:
            del self.path.locations[index]
        self.scene.clearSelection()
        self.update()

    def moveLocationUp(self):
        if not self.pathList.selectedItems(): return
        index = self.pathList.currentRow()
        if index > 0:
            self.path.locations = self.path.locations[:index-1] + \
                                  [self.path.locations[index]] + \
                                  [self.path.locations[index-1]] + \
                                  self.path.locations[index+1:]
            self.update()
            self.pathList.setItemSelected(self.pathList.item(index-1), True)
            self.pathList.setCurrentRow(index-1)

    def moveLocationDown(self):
        if not self.pathList.selectedItems(): return
        index = self.pathList.currentRow()
        if index < self.pathList.count()-1:
            self.path.locations = self.path.locations[:index] + \
                                  [self.path.locations[index+1]] + \
                                  [self.path.locations[index]] + \
                                  self.path.locations[index+2:]
            self.update()
            self.pathList.setItemSelected(self.pathList.item(index+1), True)
            self.pathList.setCurrentRow(index+1)

    def changeElement(self):
        index = self.elementComboBox.currentIndex()
        if index == 0:
            self.path.element = None
        elif index == 1:
            element = Element("Enter name here", radial=[1,4,14,17,18])
            form = EditElementDialog(element, self.materials)
            if not form.exec_():
                del element
                return
            self.elements.addItem(element)
            self.elementComboBox.addItem(
                element.name, QVariant(element.uid))
            self.path.element = element
        else:
            uid = str(self.elementComboBox.itemData(index).toString())
            element = self.elements.getItem(uid)
            self.path.element = element
```

```
        self.update()

    def flipElement(self):
        for item in self.pathList.selectedItems():
            index = self.pathList.row(item)
            if index not in self.path.flip:
                self.path.flip.add(index)
            else:
                self.path.flip.remove(index)
        self.update()

    def rotateElement(self):
        for item in self.pathList.selectedItems():
            index = self.pathList.row(item)
            if index not in self.path.rotate:
                self.path.rotate.add(index)
            else:
                self.path.rotate.remove(index)
            currentItem = item
        self.update()

    def accept(self):
        self.path.name = self.nameLineEdit.text()
        QDialog.accept(self)
```

B.8 gui/editTimeDialog.py

```
#!/usr/bin/env python

from __future__ import division

from PyQt4.QtCore import *
from PyQt4.QtGui import *

class EditTimeDialog(QDialog):

    def __init__(self, runData, parent=None):
        super(EditTimeDialog, self).__init__(parent)

        self.run = runData

        # Create top widgets
        powerLabel = QLabel("Power (W):")
        self.powerLineEdit = QLineEdit()
        cycleLabel = QLabel("Number of Cycles:")
        self.cycleSpinBox = QSpinBox()
        self.cycleSpinBox.setValue(1)
        self.cycleSpinBox.setMinimum(1)
        self.fixedRadio = QRadioButton("Change fuel at fixed times")
        self.controlRadio = QRadioButton("Change fuel based on control device")

        # Create top layout
        gridLayout = QGridLayout()
        gridLayout.addWidget(powerLabel,0,0)
        gridLayout.addWidget(self.powerLineEdit,0,1)
        gridLayout.addWidget(cycleLabel,1,0)
        gridLayout.addWidget(self.cycleSpinBox,1,1)
        radioLayout = QVBoxLayout()
        radioLayout.addWidget(self.fixedRadio)
        radioLayout.addWidget(self.controlRadio)
        topLine = QFrame(self)
        topLine.setFrameShape(QFrame.VLine)
        topLine.setFrameShadow(QFrame.Sunken)
        topLayout = QHBoxLayout()
        topLayout.addLayout(radioLayout)
        topLayout.addWidget(topLine)
        topLayout.addLayout(gridLayout)

        # Create table and buttons
```

```
        self.timeTable = QTableWidget()
        self.timeTable.setColumnCount(2)
        self.timeTable.setHorizontalHeaderLabels(["Time (days)","Power (%)"])
        self.addTimeButton = QPushButton("&Add Time")
        self.removeTimeButton = QPushButton("&Remove Time")
        self.okButton = QPushButton("&Ok")

        # Create overall layout
        buttonLayout = QVBoxLayout()
        buttonLayout.addWidget(self.addTimeButton)
        buttonLayout.addWidget(self.removeTimeButton)
        buttonLayout.addStretch()
        buttonLayout.addWidget(self.okButton)
        bottomLayout = QHBoxLayout()
        bottomLayout.addWidget(self.timeTable)
        bottomLayout.addLayout(buttonLayout)
        layout = QVBoxLayout()
        layout.addLayout(topLayout)
        layout.addLayout(bottomLayout)
        self.setLayout(layout)

        # Set signals
        self.connect(self.fixedRadio, SIGNAL("clicked()"), self.toggleTime)
        self.connect(self.controlRadio, SIGNAL("clicked()"), self.toggleTime)
        self.connect(self.addTimeButton, SIGNAL("clicked()"), self.addTime)
        self.connect(self.removeTimeButton, SIGNAL("clicked()"), self.removeTime)
        self.connect(self.okButton, SIGNAL("clicked()"), self.accept)
        self.connect(self.timeTable, SIGNAL("cellChanged(int,int)"), self.tableItemChanged)

        self.setWindowTitle("Edit Times")
        self.initialLoad()

    def toggleTime(self):
        if self.fixedRadio.isChecked():
            self.cycleSpinBox.setDisabled(True)
            self.timeTable.setDisabled(False)
            self.run.fixedTimes = True
        else:
            self.cycleSpinBox.setDisabled(False)
            self.timeTable.setDisabled(True)
            self.run.fixedTimes = False

    def initialLoad(self):
        self.fixedRadio.setChecked(self.run.fixedTimes)
        self.controlRadio.setChecked(not self.run.fixedTimes)
        self.toggleTime()
        self.cycleSpinBox.setValue(self.run.nCycles)

        self.powerLineEdit.setText(str(self.run.power))
        self.populateTable()

    def populateTable(self, selected=None):
        self.timeTable.clearContents()
        self.timeTable.setRowCount(len(self.run.time))
        for row in range(len(self.run.time)):
            self.timeTable.setItem(
                row, 0, QTableWidgetItem(str(self.run.time[row])))
            self.timeTable.setItem(
                row, 1, QTableWidgetItem(str(self.run.powerFraction[row]*100)))

    def tableItemChanged(self, row, column):
        row = self.timeTable.currentRow()
        column = self.timeTable.currentColumn()
        if column == 0:
            self.run.time[row] = self.timeTable.currentItem().text().toFloat()[0]
        elif column == 1:
            self.run.powerFraction[row] = float(self.timeTable.currentItem().text())/100

    def addTime(self, selected=None):
        self.run.time.append('')
        self.run.powerFraction.append('')
```

```
        self.populateTable()

        self.timeTable.setFocus()
        self.timeTable.setCurrentCell(self.timeTable.rowCount()-1,0)
        self.timeTable.editItem(self.timeTable.currentItem())

    def removeTime(self):
        index = self.timeTable.currentRow()
        self.run.time.pop(index)
        self.run.powerFraction.pop(index)
        self.populateTable()

    def accept(self):
        self.run.nCycles = self.cycleSpinBox.value()
        self.run.power = self.powerLineEdit.text().toFloat()[0]
        QDialog.accept(self)
```

B.9 gui/geometry.py

```
#!/usr/bin/env python

from __future__ import division

from math import *
from PyQt4.QtCore import *
from PyQt4.QtGui import *

class Location(QGraphicsItem):

    def __init__(self, uid, points, translation, rotation):
        super(Location, self).__init__()
        self.setFlags(QGraphicsItem.ItemIsSelectable)

        self.uid = uid
        self.points = points
        self.translation = translation
        self.rotation = rotation
        self.defaultColor = Qt.lightGray
        self.color = self.defaultColor
        self.hasPath = False

        self.path = QPainterPath()
        self.path.addPolygon(QPolygonF([QPointF(x,y) for x,y in points]))
        self.path.closeSubpath()
        self.translate(translation[0], translation[1])
        self.rotate(rotation)

    def boundingRect(self):
        return self.path.boundingRect()

    def shape(self):
        return self.path

    def paint(self, painter, option, widget=None):
        painter.setPen(Qt.black)
        painter.setBrush(QBrush(self.color))
        painter.drawPath(self.path)

    def refresh(self):
        try:
            if self.isSelected():
                self.color = Qt.red
            elif self.hasPath:
                self.color = Qt.darkGray
            else:
                self.color = self.defaultColor
            self.update()
        except RuntimeError: pass
```

```
    def hoverEnterEvent(self, event):
        self.color = Qt.gray
        self.update()

    def hoverLeaveEvent(self, event):
        self.refresh()

class Label(QGraphicsTextItem):

    def __init__(self, text, points, translation, rotation):
        super(Label, self).__init__()



length = 75
width = length*cos(pi/6)

MITR_points = [(0,0), (width,length/2), (width,-length/2), (0,-length)]

MITR_pairs = [((0,0),30), ((0,0),150),((0,0),270), # A-ring

              ((length,0),90), ((length/2,width),90), ((-length/2,width),90), # B-ring
              ((-length/2,width),210), ((-length,0),210), ((-length/2,-width),210),
              ((-length/2,-width), -30), ((length/2,-width), -30), ((length,0), -30),

              ((2*length,0),90), ((3/2*length,width),90), ((length,2*width),90), #C-ring
              ((0,2*width),90), ((-length,2*width),90), ((-length,2*width),210),
              ((-3/2*length,width),210), ((-2*length,0),210), ((-3/2*length,-width),210),
              ((-length,-2*width),210), ((-length,-2*width),-30), ((0,-2*width),-30),
              ((length,-2*width),-30), ((3/2*length,-width),-30), ((2*length,0),-30)]

MITR_ids = ['A1','A2','A3', # A-ring
            'B1','B2','B3','B4','B5','B6','B7','B8','B9',# B-ring
            'C1','C2','C3','C4','C5','C6','C7','C8','C9',# C-ring
            'C10','C11','C12','C13','C14','C15'] # C-ring

MITR_offset = {30: (0.3*length,0), 150: (-length/2,0.4*width),
               270: (-length/2,-0.6*width), 90: (0,0.4*width),
               210: (-0.7*length,0), -30: (0,-0.6*width)}
```

Appendix C: Example Files

C.1 Example of mcodeFM_input

```
fixed True
nCycles 1
time 80.0 160.0 240.0
power 5000000.0 5000000.0 5000000.0

path A1 B1 C1
element c1152790-1c9a-408d-9ae8-7cd9258ef506
time 0
flip
rotate

element c1152790-1c9a-408d-9ae8-7cd9258ef506
plates 18
axial 1
radial 9 18
material 1 1 -17.02
92238.70c -0.71991 92235.70c -0.17775 92234.70c -0.00234 42000.66c -0.10
material 2 1 -17.02
92238.70c -0.71991 92235.70c -0.17775 92234.70c -0.00234 42000.66c -0.10

path B4 B5 C8
element c1152790-1c9a-408d-9ae8-7cd9258ef506
time 0
flip
rotate

path C11 C12 C13
element c1152790-1c9a-408d-9ae8-7cd9258ef506
time 0
flip
rotate
```

C.2 Example of control_input

```
surface 860 5
surface 864 5
surface 868 5
surface 872 4
surface 874 4
surface 876 4
range 0.0 50.0
```

C.3 Example of mcnp.sh

```
#!/bin/sh

keffsearch -c ../control_input -o $1          # Perform control blade search
keffsearch -c ../control_input -m input       # Move blade in MCODE file
keffsearch -c ../control_input -m ../skeleton # Move blade in skeleton
mcnp $1 $2 $3 tasks 6                         # Run MCNP
```

References

[1] Argonne National Laboratory. (2008, July) Reduced Enrichment for Research and Test Reactors. [Online]. http://www.rertr.anl.gov/

[2] U.S. Nuclear Regulatory Commission, "Limiting the Use of Highly Enriched Uranium in Domestically Licensed Research and Test Reactors," 10 Code of Federal Regulations Part 50, 1986.

[3] Daniel M Wachs, Curtis R Clark, and Randall J Dunavant, "Conceptual Process Description for the Manufacture of Low-Enriched Uranium-Molybdenum Fuel," Idaho National Laboratory, INL/EXT-08-13840, 2008.

[4] Government Accountability Office, "Action May Be Needed to Reassess the Security of NRC-Licensed Research Reactors," GAO-08-403, 2008.

[5] National Nuclear Security Administration. (2008, December) GTRI: Reducing Nuclear Threats. [Online]. http://nnsa.energy.gov/news/print/2262.htm

[6] Argonne National Laboratory, "U-Mo Fuels Handbook Version 1.0," Contract W-31-109-ENG-38, June 2006.

[7] U.S. Nuclear Regulatory Commission, "Safety Evaluation Report Related to the Evaluation of Low-Enriched Uranium Silicide-Aluminum Dispersion Fuel for Use in Non-Power Reactors," NUREG-1313, 1988.

[8] Tom H Newton Jr., "Development of a Low Enrichment Uranium Core for the MIT Reactor," Cambridge, MA, Ph.D. Thesis 2006.

[9] International Atomic Energy Agency. (2008, April) Nuclear Research Reactors in the World. [Online]. http://www.iaea.org/worldatom/rrdb/

[10] Nuclear Reactor Laboratory, "Safety Analysis Report for the MIT Research Reactor (MITR-III)," Massachusetts Institute of Technology, Cambridge, MA, NRC Accession ML053190384, 2000.

[11] MITR Staff, "MIT Reactor MITR-2 Fuel Element Assembly," MIT Nuclear Reactor Laboratory, Cambridge, MA, Drawing R3F-201-4, 1996.

[12] Yu-Chih Ko, "Thermal Hydraulics Analysis of the MIT Research Reactor in Support of a Low Enrichment Uranium (LEU) Core Conversion," Cambridge, MA, M.S. Thesis 2008.

[13] Y-12 National Security Complex, "The Y-12 Standard Specification Low Enriched Uranium Metal

Supply to Research and Test Reactors," Global Nuclear Security & Supply, Oak Ridge, Tennessee, Technical Report Y/GNSS/05-05 Revision 1, 2005.

[14] J. Rest, Y. S. Kim, G. L. Hofman, M. K. Meyer, and S. L. Hayes, "U-Mo Fuels Handbook," Argonne National Laboratory, June 2006.

[15] A. Addae, "The Reactor Physics of the Massachusetts Institute of Technology Research Reactor Redesign," Massachusetts Institute of Technology, Cambridge, MA, Ph.D. Thesis 1970.

[16] Andrew Kadak, "Fuel Management of the Redesigned MIT Research Reactor," Massachusetts Institute of Technology, Cambridge, MA, Ph.D. Thesis 1972.

[17] John A. Bernard Jr., "MITR-II Fuel Management, Core Depletion, and Analysis: Codes Developed for the Diffusion Theory Program CITATION," Massachusetts Institute of Technology, Cambridge, MA, PhD Thesis 1979.

[18] E. Redmond, J. Yanch, and O. Harling, "Monte Carlo Simulation of the MIT Research Reactor," *Nuclear Technology*, vol. 106, pp. 1-14, 1994.

[19] X-5 Monte Carlo Team, "MCNP - A General Monte Carlo N-Particle Transport Code, Version 5," Los Alamos National Laboratory report LA-UR-03-1987, October 3, 2005.

[20] Arne Olson, "A Users Guide for the REBUS-PC Code, Version 1.4," Argonne National Laboratory, Technical Report ANL/RERTR/TM-32, December 21, 2001.

[21] Thomas H. Newton Jr., Arne P. Olson, and John A. Stillman, "Reactor Core Design and Modeling of the MIT Research Reactor for Conversion to LEU," in *International Meeting on Reduced Enrichment for Research and Test Reactors*, Washington, DC, October 5-8, 2007.

[22] J. Deen et al., "WIMS-ANL User Manual Rev. 6," Argonne National Laboratory, Technical Report ANL/TD/TM99-07, February, 2004.

[23] P. Hejzlar, M. Driscoll, and M. Kazimi Z. Xu, "An Improved MCNP-ORIGEN Depletion Program (MCODE) and its Verification for High-Burnup Applications," , Seoul, South Korea, October 2002.

[24] A.G. Croff, "ORIGEN2: a Versatile Computer Code for Calculating the Nuclide Compositions and Characteristics of Nuclear Materials," *Nuclear Technology*, vol. 62, pp. 335-352, September 1983.

[25] R.L. Moore, B.G. Schnitzler, C.A. Wemple, R.S. Babcock, and D.E. Wessol, "MOCUP: MCNP-ORIGEN2 Coupled Utility Program," Idaho National Engineering Laboratory, INEL-95/0523, September, 1995.

[26] H.R. Trellue and D.I. Poston, "User's Manual, Version 2.0, for MONTEBURNS, Version 2.0," Los

Alamos National Laboratory, LA-UR-99-4999, September, 1999.

[27] D. Knott, B.H. Forssen, and M. Edenius, "CASMO-4, a Fuel Assembly Burnup Program, Methodology," Studsvik of America, Inc., Studsvik/SOA-95/2, 1995.

[28] C.M. Kang and R.O. Mosteller, "Incorporation of a Predictor-Corrector Depletion Capability into the CELL-2 Code," *Trans. Am. Nucl. Soc.*, vol. 45, pp. 729-731, November 1983.

[29] Nick Touran, "Graphical Automation of REBUS/MC**2 Fast Reactor Calculations," in *2008 American Nuclear Society Student Conference*, College Station, TX, p. on CDROM.

[30] Anselmo Peretto. (2009, March) py2exe.org. [Online]. http://www.py2exe.org/

[31] (2009, February) Nullsoft Scriptable Install System. [Online]. http://nsis.sourceforge.net/

[32] Paul K. Romano, Benoit Forget, and Thomas H. Newton Jr., "Extending MCODE Capabilities for Innovative Design Studies at the MITR," *Trans. Am. Nucl. Soc.*, vol. 99, pp. 659-660, 2008.

[33] R. E. Morrow, T. H. Trumbull, T. J. Donovan, and T. M. Sutton, "A keff Search Capability in MC21," in *Joint International Topical Meeting on Mathematics & Computation and Supercomputing in Nuclear Applications*, Monterey, CA, 2007, pp. on CD-ROM.

[34] Thomas H. Newton Jr., Paul K. Romano, and Benoit Forget, "REBUS and MCODE Burnup Modeling of the MITR for Conversion Studies," in *International Meeting on Reduced Enrichment for Research and Test Reactors*, Washington, DC, 2008.

[35] Oak Ridge National Laboratory. (April, 2009) SCALE Software for Nuclear Licensing and Safety Analyses. [Online]. http://www.ornl.gov/sci/scale/

[36] E. Fridman, E. Shwageraus, and A. Galperin, "Efficient Generation of One-Group Cross Sections for Coupled Monte Carlo Depletion Calculations," *Nuclear Science and Engineering*, vol. 159, no. 1, pp. 37-47, May 2008.

[37] David C. Carpenter, "A Comparison of Constant Power Depletion Algorithms," in *International Conference on Mathematics and Computational Methods & Reactor Physics*, Saratoga Springs, NY, 2009, pp. on CD-ROM.