A Novel Tool for the Assessment and Validation of Acceleration Methods for Solving the Neutron Transport Equation

by

Joshua Stephen Rehak

A dissertation submitted in partial satisfaction of the

requirements for the degree of

Doctor of Philosophy

in

Engineering – Nuclear Engineering

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Associate Professor Rachel Slaybaugh, Chair
Associate Professor Massimiliano Fratoni
Professor Per-Olof Persson

Summer 2021

A Novel Tool for the Assessment and Validation of Acceleration Methods for Solving the Neutron Transport Equation

Abstract

A Novel Tool for the Assessment and Validation of Acceleration Methods for Solving the Neutron Transport Equation

by

Joshua Stephen Rehak

Doctor of Philosophy in Engineering – Nuclear Engineering

University of California, Berkeley

Associate Professor Rachel Slaybaugh, Chair

The Boltzmann Transport Equation describes the behavior of the population of neutrons in nuclear systems. Solving this equation is therefore of great interest to researchers designing future generations of nuclear reactors among many other applications. Solving the neutron transport equation using computers requires careful discretization of the phase space and iterative methods to converge to a solution. These methods can be slow to converge, often due to material properties in systems of interest. Highly scattering media, for example, are often used in reactor designs and can cause many methods to take arbitrarily long to converge. To combat computational inefficiencies, researchers modify the iteration schemes using a broad class of algorithms called acceleration methods. Implementing, assessing, and validating acceleration methods is necessary but challenging for researchers. A particular challenge is confirming whether an acceleration method is actually improving the simulation the way we expect. Computational tools are generally designed for solving the problem of interest, not for assessing the solving process itself.

We present the Bay Area Radiation Transport (BART) code, a computational tool designed with the researcher as the end-user in mind. This code is designed to relieve some of the burden of implementing novel acceleration methods. It leverages modern coding practices to minimize the amount of code that must be modified to implement new methods and aims to make clear where these modifications need to be made. This both simplifies implementation and makes comparison across methods easier. Once implemented, the code provides a high-quality environment for testing the new method. The design of the code isolates modifications, providing a good comparison to a base case as well as other acceleration methods. Making this comparison is supported by the inclusion of a robust instrumentation system.

Developers are empowered to collect and extract data of any type from anywhere in the solving process with ease. This data collection can then be used to assess and validate implemented methods. Importantly, these data can interrogate whether the problems are being accelerated where and how the methods are designed to provide acceleration. Typically,

developers do not have the ability to see if a method is actually doing what we think, we only measure compute time and iteration count; BART provides much more information about what is happening. Finally, the code is designed with comprehensive testing to provide a reproducible and trusted environment to researchers.

The code itself is robust, with the ability to solve angular and scalar formulations of the transport equation in one, two, and three dimensions. We demonstrate a level-symmetric-like Gaussian quadrature implemented for solving angular formulations and show that it accurately integrates the spherical harmonics.

We present two acceleration methods, the two-grid (TG) and nonlinear diffusion acceleration (NDA) methods. The TG method is designed to accelerate the Gauss-Seidel (GS) iteration process in the presence of large amounts of upscattering. We demonstrate the effectiveness of the method using the BART code in one, two, and three dimensions. The effectiveness is shown by a reduction in total GS iterations by a larger factor than is required for the method to be efficient. We also demonstrate the benefits of the BART code, showing more rapid convergence of the scattering source using the code's unique instrumentation. The NDA method is designed to accelerate convergence by converging diffusive error modes more rapidly. We demonstrate effectiveness by reducing total iterations in one, and three dimensions. For the one-dimensional case, we demonstrate a significant reduction in the diffusive error modes using the BART Fourier analysis instrumentation.

We will examine the goals of the BART code and how the design meets these goals. By examining two acceleration methods and analyzing them using the data we can collect using this new tool, we will show the benefits of this novel code to the broader research community. The BART code changes how people are able to implement, assess, and validate acceleration methods. The ease of use and new information enables the development of new and better methods so we can design and build better nuclear systems.

To Mom, Dad, Jake, Christie, and Everly.

Contents

List of Figures

List of Tables

Acknowledgments

I am forever in debt to all the people that made this dissertation possible, and I cannot begin to thank them enough.

I would first like to thank my advisor Rachel N. Slaybaugh for her advice, support, patience, and friendship. Thank you for taking me on after an unconventional path brought me to gradurate school, and providing the wise counsel and insights I needed to not just survive but to thrive. Thank you as well to Per-Olof Persson and Massimiliano Fratoni for your guidance in completing this project and for serving on both my qualifying exam and dissertation committees. This work would not be possible without generous funding from the Nuclear Regulatory Commission[1] and Department of Energy.[2]

Next, I would like to thank all my fellow students and collaborators that made navigating graduate school possible. I could not have done this without our frequent discussions both professional and personal. An exhaustive list could never be made but I would like to particularly thank, in pseudo-random order[3], Joey Kabel, Sam Olivier, Vanessa Goss, Sami Lewis, Christopher Poresky, Mario Ortega, Marissa Ramirez de Chanlatte, Mitch Negus, and Milos Atz.

I thank my wonderful family for supporting me through the ups and downs of graduate school. Thank you to my mom and dad for the constant and endless love, and supporting the paths I've walked, even those difficult or far from home. I couldn't have done this without the love of the whole family, the Jillys, Nicole and Mike Wachell, and Jake and Tessa. Finally, I thank my wonderful wife and other half Christie for her patience, love and support. You made this roller coaster ride possible and worthwhile.

[1]This dissertation was prepared by Joshua Stephen Rehak under award NRC-HQ-84-14-G-0042 from the Nuclear Regulatory Commission. The statements, finding, conclusions, and recommendations are those of the author(s) and do not necessarily reflect the view of the US Nuclear Regulatory Commission.

[2]Acknowledgment: "This material is based upon work supported by the Department of Energy Office of Nuclear Energy under Award Number DE-NE0008661."

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

[3]The order of these names were randomized using the Python 3.8.10 `random.shuffle` function using a lexicographically sorted list of the names as strings and the current time and date as the seed (`202109101306`). For a docker container appropriate for reproducing this listing, please contact me.

Chapter 1

Introduction

Humanity is unique among the species on Earth, not least of all because its existential crises are of its own making. For the second half of the 20^{th} century, the cold war between the United States of America and the USSR ignited a nuclear arms race that threatened to destroy both countries, and most of humanity in the balance. Now, in the 21^{st} century, the threat of sudden destruction at the hands of a third has been replaced by the threat of slow destruction at the hands of the many. The global pollution crisis and the climate change it is driving is more insidious than nuclear war, a hard-to-identify destruction driven by a thousand small decisions. History is not without a sense of irony, as one of the greatest tools we have to fight this pollution crisis is nuclear energy.

Ever increasing demand for energy drives the global pollution crisis by creating an incentive for the use of fossil fuels and other carbon-releasing power sources. Transnational corporations focused on the short term need for profit ignore the long-term consequences of their actions, leaving governments and non-governmental-organizations to take action. Solutions to many of the effects of global climate change will require a large amount of energy: caring for and housing climate refugees, growing food in lands that have become less arable, and building carbon neutral industries. To avoid the positive feedback loop, we require the development and deployment of power sources that do not contribute to the carbon in the atmosphere, in operation or fabrication. Renewable power sources like solar and wind can meet many of our energy needs, but are dynamic and depend greatly on environmental conditions. Static power sources are still a necessity, a need most optimally met by a zero-carbon energy source, such as nuclear energy.

The gigawatt-scale mega-project nuclear reactors of the 20^{th} century are not sufficient to meet the challenges of global climate change. The largest resource required by these projects is one we do not have in bulk: time. As climate conditions change more rapidly, the spatial distribution of energy needs will change with it. Important factors support that small modular reactors (SMRs) and other advanced concept reactors may be the key to the future of nuclear energy. First, smaller projects that are quicker to deploy and require exponentially less materials are better equipped to meet current and future needs. Second, advanced reactors may also find a better fit in smart-grid technology that is a key part in maximizing the use of dynamic renewable energy sources. Finally, branching out from the water-cooled reactor designs of the 1950s and using modern fault detection systems can produce designs that are passively safe, increasing public acceptance of the technology. For these reasons, and others, It is imperative that we as a species support the development of these advanced reactors to meet this crisis.

A major need at the core of the development of advanced reactors is the use of computer modeling. A steady energy-creating nuclear reaction requires precision in both design and operation. The chain reaction is sustained by a population of neutrons that interact with a fuel material (e.g. uranium or plutonium) causing the release of both energy and more neutrons. In general, a constant neutron population size produces a steady amount of reactions and therefore a steady amount of energy. Some major factors complicate the situation and make modeling the neutron population difficult. First, the geometry of the reactor is very important and can be complex, making the path and possible escape of neutrons complicated. Second, many reactors require the neutrons to interact with a material that reduces their energy before they can interact with the fuel; a process known as moderation. Finally, models must be able to solve problems with sufficient accuracy, to ensure safety and enable licensing. The complicated problem of modeling the neutron population in a nuclear reactor has lead to the development of two broad categories of methods.

The first is Monte Carlo (MC) methods. These methods use the statistical sampling of random numbers to solve numeric problems; they are used in a broad variety of fields from biology to finance. The use of MC methods to simulate the physics and geometry of a nuclear chain reaction was first developed at Los Alamos National Lab (LANL) in the 1940s to develop the first nuclear weapons. Since that time, the methods have become widespread, complex, and increasingly powerful and precise. Now, many labs and Universities that study nuclear engineering maintain or develop their own MC code that seeks to further the art. Monte Carlo methods have many benefits, including the ability to run the simulation longer or with a more powerful computer (or both) to generate increasingly precise solutions. The downside being the exact same: that increasingly precise solutions require running the simulation longer or with a more powerful computer. In some cases, even running a simulation for an arbitrarily long amount of time will not result in a solution with enough precision. These cases require researchers to make more efficient algorithms, or use other data to improve the algorithm. These methods are not the focus of this dissertation; we will rather focus on the second broad class, deterministic methods.

Deterministic methods are those that confront the mathematical equations that describe the neutron population directly. While MC methods treat the neutrons as individual particles, deterministic methods instead consider the neutrons as a distributed population. The equation used was developed by Ludwig Boltzmann in the 1800s to describe thermodynamic systems, but is equally applicable to neutrons. Unlike MC methods, the precision of a deterministic method cannot be improved arbitrarily. The precision and quality of the solve is limited by our ability to translate a mathematical equation into a form that can be solved numerically. This is no small task, as the equation takes into account the whole range of neutron population position, velocity, and energy, all of which can span multiple orders of magnitude. Some extremely simplified versions of the equation can be solved easily, even by hand. However, a level of precision needed to ensure the safety of a novel reactor design requires a computer to solve a much more complex version and requires an assortment of mathematical tools.

Numerical methods to solve complex equations like the neutron transport equation are an active and fertile area of research. Ever since the development of the first computer algorithms, researchers have been optimizing and developing methods for solving abstract mathematical equations in the concrete digital world of a computer. Deterministic methods utilize the best of these algorithms to solve the neutron transport equation in a reactor design quickly and with high precision, but encounter limitations. In some cases, an algorithm may take an arbitrarily long amount of time to solve if the reactor uses certain materials or geometric configurations. Mathematical improvements to algorithms, called *acceleration methods*, work to make the solve faster or, sometimes, possible at all. Unfortunately, algorithms and acceleration methods that work well for some reactors may not work for others, and it may take time to find optimized methods for novel designs. It is also sometimes unclear if combinations of acceleration methods improve or detract from the ability to solve the neutron transport equation. We have a metaphorical toolbox of modeling algorithms and acceleration methods that we have experience working with, but we do not always know which will work best for novel reactor designs. If we want to support the modeling and development of advanced reactors, we need a way to assess which tools will be best, especially for reactors where we have less experience and intuition. This dissertation presents an environment for performing such an assessment.

The BART code presented in this dissertation is designed to provide a testing ground for deterministic acceleration methods, a new and valuable contribution to reactor simulation. This code is unlike codes designed to solve the neutron transport problem quickly and provide a solution at the end. Instead, the data of concern is collected *during* the solve algorithms itself, to provide insight into how the methods are really working as well as whether they are effective and for what reasons. Existing codes may present difficulties when collecting this data as they are not built for this purpose, resulting in large scale or intrusive modifications. The BART code is designed to make the data collection process easy and minimize impact to the rest of the solve. Designed with the developer in mind, the code is modular so that portions can be swapped out, while maintaining the rest of the code identical. This enables isolation of the modifications to assess their effectiveness.

Overall, the combination of improved data collection and modular design will provide researchers with a testing ground for developing and investigating novel methods and combinations of methods. In the past, people have implemented methods and use a few basic indicators to determine whether the methods are working the way they expect without having the data to truly support the conclusions. A main driver for the BART code is that we can get detailed data throughout a solve to validate our mathematical hypotheses. To some extent this is enabled by modern computing languages and software development tools. Moreover, this is motivated by increasing access to powerful computing resources combined with the need to rapidly, repeatedly, and accurately model reactors we have much less experience with than light water reactors. The following chapters describe the philosophy, implementation, and testing of BART:

Chapter 2: Background This chapter will present the neutron transport equation and the schemes used to discretize the phase space for solving. Finite element methods, central to the operation of the BART code, will be discussed. Two second-order formulations, the diffusion and self-adjoint angular-flux (SAAF) formulations, will be derived.

Chapter 3: Iterative Methods and Acceleration Schemes This chapter describes the iterative methods used to solve the discretized transport equation and the convergence challenges that are present in these methods. Two acceleration schemes: TG and NDA will be described.

Chapter 4: Assessment of Acceleration Methods This chapter discusses the challenges associated with implementing acceleration methods in novel or existing codes. In addition, we discuss how to define and assess the effectiveness of acceleration methods. Finally, we outline the challenges associated with validating whether the methods are effective for the reasons expected.

Chapter 5: The BART Code and Design This chapter provides an overview of the design goals of the BART code. In addition, we describe how the particular design features of BART meet these goals and enable it to be an important and novel tool for future researchers.

Chapter 6: Assessing Acceleration Methods using BART In this chapter we discuss the implementation details of two acceleration methods in BART: TG and NDA. We then use the tools provided by BART to assess and validate the effectiveness of these methods.

Chapter 7: Conclusions and Future Work This chapter will cover lessons learned and conclusions drawn from the BART project. We will discuss existing issues and a future path forward for the code.

Chapter 2

Background

In this chapter, we will cover some of the mathematical tools used to quantify the propagation and generation of neutrons in a steady state system. We will begin with the time independent Boltzmann transport equation, a balance equation that governs the production and loss of neutrons in such a system. Next, we will discuss the required discretization of the phase space of this equation. This includes a discussion of the finite element method and its application. The approximations and assumptions we use will enable us to ultimately solve the transport equation for complex systems. Finally, we will discuss two formulations of the transport equation: the diffusion equation and the SAAF equation. These will be used extensively in future Chapters, and so we will derive the forms appropriate for the finite element method as well.

2.1 Boltzmann Transport Equation

The Boltzmann transport equation to describes the population of neutrons in a steady-state system. To use this equation, we will ignore two quantum-mechanics properties of neutrons. First, we will assume that that all neutrons are point particles. This naturally holds for high energy neutrons with very small de Broglie wavelengths. There are very few neutrons within the populations of interest that have low enough energy that their wave-like nature is important and thus this assumption has minimal impact on accuracy. Second, we will assume that all neutrons can be exactly described by both a position and a velocity. The uncertainty in knowing these values simultaneously is ignored. Since we are looking for average, aggregate behavior of the neutron population, this assumption also has minimal impact on accuracy.

For a given neutron, the position is described by the vector $\vec{r}$ and the velocity is

$$\vec{v} = v(E)\hat{\Omega} \ ,$$

where $v(E) = \sqrt{2E/m}$ is the speed, m is particle mass, $E \in [0, \infty]$ is particle energy, and $\hat{\Omega}$ is a directional unit vector such that $|\hat{\Omega}| = 1$. The $\hat{\Omega}$ vector can also be described in polar coordinates,

$$\hat{\Omega} = \begin{bmatrix} \sin(\theta)\cos(\varphi) \\ \sin(\theta)\sin(\varphi) \\ \cos(\theta) \end{bmatrix} .$$

The entire phase space as described is shown in Fig. 2.1. In a three-dimensional (3D) domain, there are six degrees of freedom for a neutron: three position, two direction, and one energy. As neutrons propagate through matter, they interact with nuclei and are scattered

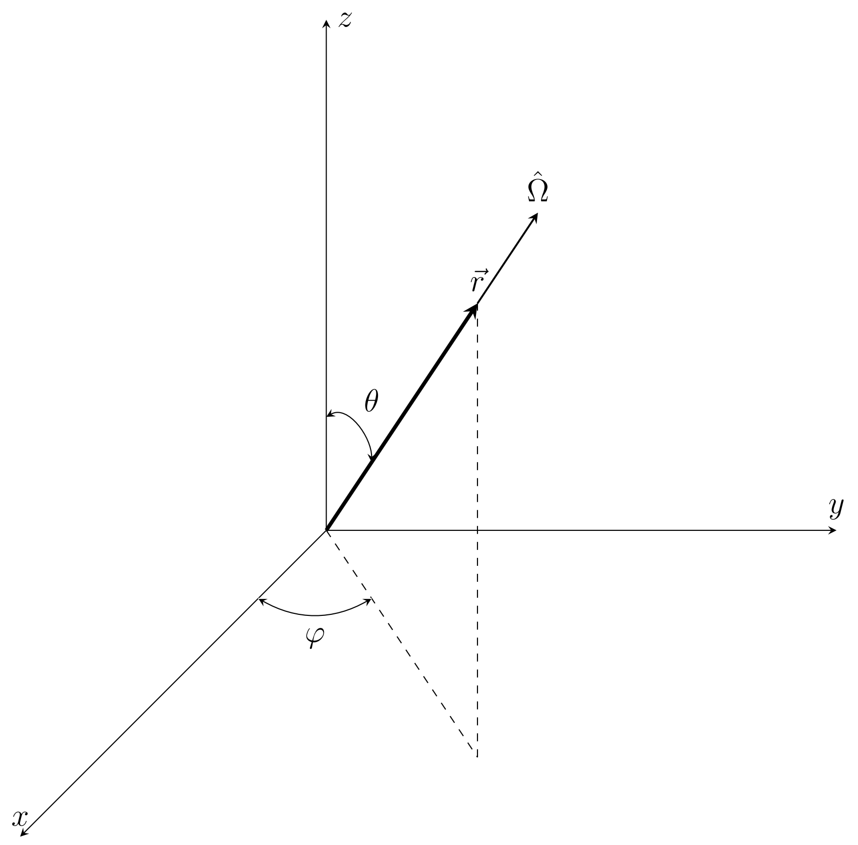


Figure 2.1: Neutron phase space in three-dimensional position with two-dimensional direction vector.

or absorbed. For finite regions, neutrons will eventually stream out through a boundary if they are not absorbed. Scattering events may change the velocity (in magnitude, direction, or both) of a neutron but ultimately leave the number of neutrons unchanged. Absorption events remove a neutron from the population but some events, such as fission or $(n, 2n)$, may produce secondary neutrons.

A description of the steady-state neutron population that includes these effects is the linear Boltzmann transport equation,

$$\begin{aligned}&\left[\hat{\Omega}\cdot\nabla+\Sigma_t(\vec{r},E)\right]\psi(\vec{r},E,\hat{\Omega})\\&\quad=\int_0^\infty dE'\int_{4\pi}d\hat{\Omega}'\Sigma_s(\vec{r},E'\to E,\hat{\Omega}'\to\hat{\Omega})\psi(\vec{r},E',\hat{\Omega}')\\&\quad+\chi(E)\int_0^\infty dE'\nu(E')\Sigma_f(\vec{r},E')\int_{4\pi}d\hat{\Omega}'\psi(\vec{r},E',\hat{\Omega}')+Q(\vec{r},E,\hat{\Omega})\ ,\end{aligned}\tag{2.1}$$

where linearity is based on the assumption that neutron-neutron interactions are ignored[1]. The angular neutron flux $\psi(\vec{r},E,\hat{\Omega})$ is the variable for which we are solving and gives the number of neutrons per unit area per steradian per unit energy; the other terms are defined in Table 2.1. A full derivation of the transport equation and a more in depth description of

Table 2.1: Quantities and terms in the linear Boltzmann transport equation.

Term	Description
$\Sigma_t(\vec{r},E)$	Total cross-section at $\vec{r}$ for neutrons with energy E
$\Sigma_s(\vec{r},E'\to E,\hat{\Omega}'\cdot\hat{\Omega})$	Differential scattering cross-section at $\vec{r}$ that scatters neutrons from energy E' to E and direction of motion $\hat{\Omega}'$ to $\hat{\Omega}$
$\Sigma_f(\vec{r},E')$	Fission cross-section at $\vec{r}$ for neutrons with energy E'
$\nu(E')$	Average neutrons created by fission caused by a neutron of energy E'
$\chi(E)$	Probability that a neutron of energy E will be created by a fission reaction
$Q(\vec{r},E,\hat{\Omega})$	A fixed source at $\vec{r}$ that emits neutrons of energy E in direction $\hat{\Omega}$

all the terms can be found in many standard texts [1, 2, 3].

The linear transport equation is often expressed in *operator form*

$$\mathcal{H}\psi=(\mathcal{S}+\mathcal{F})\,\psi+Q\ ,\tag{2.2}$$

where

$$\begin{aligned}\mathcal{H}f(\vec{r},E,\hat{\Omega})&=\left[\hat{\Omega}\cdot\nabla+\Sigma_t(\vec{r},E)\right]f(\vec{r},E,\hat{\Omega})\\\mathcal{S}f(\vec{r},E,\hat{\Omega})&=\int_0^\infty dE'\int_{4\pi}d\hat{\Omega}'\Sigma_s(\vec{r},E'\to E,\hat{\Omega}'\to\hat{\Omega})f(\vec{r},E',\hat{\Omega})\\\mathcal{F}f(\vec{r},E,\hat{\Omega})&=\chi(E)\int_0^\infty dE'\nu(E')\Sigma_f(\vec{r},E')\int_{4\pi}d\hat{\Omega}'f(\vec{r},E',\hat{\Omega}')\end{aligned}$$

are the transport operator, scattering operator, and fission operator, respectively.

[1]Interactions between neutrons would be proportional to the angular flux of both the incident and interacted-upon neutrons, which are members of the same population. This leads to a term proportional to the square of the angular flux, and thus non-linearity.

2.1.1 Criticality

Many fission reactions result in the production of one or more neutrons, creating a chain-reaction with the potential of a self-sustaining equilibrium state. In an equilibrium situation where, on average, every lost neutron is replaced by a fission neutron the rate of change of the population size is zero, and thus at a critical point; the system is called "critical." If more neutrons are created than lost, the system is "supercritical" and will grow exponentially. Conversely a net loss of neutrons will cause the population to decay away exponentially in a "subcritical" system.

Calculating the criticality state of a system is of great practical interest. This is particularly important in nuclear reactor design, where control of the criticality is vital to both steady-state operation, power changes, and accident studies. To this end, we introduce a factor, k-effective, that synthetically weighs the average neutrons created per fission: ν/k. Solving the system assuming an equilibrium state will yield a value for this factor. If it is unity, the each neutron produces one new neutron and the system is critical. Deviation from unity indicates a deviation from critical. If $k > 1$, the more than one neutron is produced by each neutron, indicating that the system is supercritical. A similar line of reasoning for cases where $k < 1$ leads to the following summary for k-effective,

$$k \begin{cases} < 1, & \text{subcritical,} \\ = 1, & \text{critical,} \\ > 1 & \text{supercritical.} \end{cases}$$

Expressing the k-effective version of the transport equation,

$$\begin{aligned} \left[\hat{\Omega} \cdot \nabla + \Sigma_t(\vec{r}, E)\right] \psi(\vec{r}, E, \hat{\Omega}) \\ = \int_0^\infty dE' \int_{4\pi} d\hat{\Omega}' \Sigma_s(\vec{r}, E' \to E, \hat{\Omega}' \to \hat{\Omega}) \psi(\vec{r}, E', \hat{\Omega}') \\ + \frac{\chi(E)}{k} \int_0^\infty dE' \nu(E') \Sigma_f(\vec{r}, E') \int_{4\pi} d\hat{\Omega}' \psi(\vec{r}, E', \hat{\Omega}')\ , \end{aligned} \tag{2.3}$$

and in operator form,

$$\mathcal{H}\psi = \left(\mathcal{S} + \frac{1}{k}\mathcal{F}\right)\psi,$$

we see that this is an eigenvalue problem with eigenvalues k corresponding to angular flux eigenvectors ψ. The k-eigenvalue problem is defined without an external source, $Q(\vec{r}, E, \hat{\Omega})$, which has been dropped from the formulation. On physical grounds, we will assume that there exists at least one eigenvalue k with a nonnegative eigenfunction [2]. In most cases there will in fact be an infinite number of values of k, with the largest corresponding to the positive eigenvalue, k_0. The remaining k values will be ordered such that $k_0 > k_1 > k_2 \ldots$. Other eigenvalue formulations of the transport equation exist but are outside the scope of this work [4].

2.2 Discretization and Expansion in Angle and Energy

Solving a differential equation like the Boltzmann transport equation deterministically requires a discretization of the phase space. Through this process, we will convert the infinite degrees of freedom of a continuous system into a finite-dimensional space that we are able to solve. Even with discretization, solving the equation with a high amount of accuracy often requires an extremely large number of degrees of freedom. The use of supercomputers makes very large problems possible, with possible discretizations having up to 100 million spatial cells, 256 angles per octant, and over 200 energy groups [5]. Discretization of energy and angle will be discussed in this section, and the discretization of space for the use of finite element methods will be discussed in the next section.

2.2.1 Energy Discretization

The energy solution space is defined by

$$\mathbb{E} = \{E : E \in \mathbb{R}, 0 < E \leq E_{\max}\}$$

where $E_{\max}$ is some chosen maximum energy. A good choice for this value is the maximum energy at which we expect to find neutrons in our system. Typically, this is determined by the maximum energy of neutrons generated by fission events. We will divide this space into G non-overlapping intervals that cover the entire space, $\{E_0, \ldots, E_{G-1}\}$ such that $\mathbb{E} = \bigcup_{i=0}^{G-1} E_i$. The size and location of these groups will be informed by the physics of the problem. The intervals are also by convention indexed such that lower numbers indicate higher energy values, $E_g > E_{g+1}$.

The next step is to integrate Eq. (2.3) over the energy domain,

$$\begin{aligned}\int_0^{E_{\max}} &\left[\hat{\Omega} \cdot \nabla + \Sigma_t(\vec{r}, E)\right] \psi(\vec{r}, E, \hat{\Omega}) dE \\ &= \int_0^{E_{\max}} dE \int_0^{E_{\max}} dE' \int_{4\pi} d\hat{\Omega}' \Sigma_s(\vec{r}, E' \to E, \hat{\Omega}' \to \hat{\Omega}) \psi(\vec{r}, E', \hat{\Omega}') \\ &+ \frac{1}{k} \int_0^{E_{\max}} dE\, \chi(E) \int_0^{E_{\max}} dE' \nu(E') \Sigma_f(\vec{r}, E') \int_{4\pi} d\hat{\Omega}' \psi(\vec{r}, E', \hat{\Omega}') \; . \qquad (2.4)\end{aligned}$$

We will assume that the integral can be expressed as a summation of integrals across each energy group,[2]

$$\int_0^{E_{\max}} = \sum_{g=0}^{G-1} \int_{E_{g+1}}^{E_g} dE \; .$$

[2]Formally, we will be using a Petrov-Galerkin scheme in which our test function is unity within the energy group interval and zero everywhere else.

Using this, Eq. (2.4) becomes G equations of the form,

$$\begin{aligned}\left[\hat{\Omega}\cdot\nabla + \Sigma_t^g(\vec{r})\right]\psi_g(\vec{r},\hat{\Omega})dE = &\sum_{g'=0}^{G-1}\int_{4\pi} d\hat{\Omega}'\Sigma_s^{g'\to g}(\vec{r},\hat{\Omega}'\to\hat{\Omega})\psi_{g'}(\vec{r},\hat{\Omega}') \\ &+ \frac{\chi_g}{k}\sum_{g'=0}^{G-1}\nu_{g'}\Sigma_f^{g'}(\vec{r})\int_{4\pi} d\hat{\Omega}'\psi_{g'}(\vec{r},\hat{\Omega}')\ , \qquad (2.5)\end{aligned}$$

for $g = 0,\dots,G-1$ where,

$$\begin{aligned}\psi_g(\vec{r},\hat{\Omega}) &= \int_{E_{g+1}}^{E_g}\psi(\vec{r},E,\hat{\Omega})dE \\ \Sigma_t^g(\vec{r}) &= \int_{E_{g+1}}^{E_g}\Sigma_t(\vec{r},E)dE \\ \Sigma_s^{g'\to g}(\vec{r},\hat{\Omega}'\to\hat{\Omega}) &= \int_{E_{g'+1}}^{E_{g'}}dE'\int_{E_{g+1}}^{E_g}dE\ \Sigma_s(\vec{r},E'\to E,\hat{\Omega}'\to\hat{\Omega}) \\ \nu_g &= \int_{E_{g+1}}^{E_g}\nu(E)dE \\ \chi_g &= \int_{E_{g+1}}^{E_g}\chi(E)dE \\ \Sigma_f^g(\vec{r}) &= \int_{E_{g+1}}^{E_g}\Sigma_f(\vec{r},E)dE\ .\end{aligned}$$

This form of the neutron transport equation (NTE), Eq. (2.5), gives the G multigroup equations. The number of groups can vary greatly depending on the amount of computational power at hand, and the problem of interest. For the simplest problems, few groups may suffice. For example, a simple pressurized water reactor problem may only require two groups: a high energy (fast) group for neutrons born in fission events, and a low energy (thermal) group for neutrons that cause these events after losing their energy by scattering. Modern deterministic codes provide the option to use many more groups for more complex analysis. For example, the SCALE code system [6] developed at Oak Ridge National Laboratory provides a number of group structures, including one with 252 groups. The specifics of discretization of the energy spectrum and modifications to cross-sections are outside the scope of this work.

2.2.2 Scattering Expansion

The handling of the scattering operator is of particular concern. Fission reactions release a large amount of energy, some of which is given to the neutrons released by the reaction that will go on to cause further reactions. These energetic fast neutrons are not likely to immediately cause fission in some problems of interest. In thermal reactor problems, for example, we expect these fast neutrons to scatter and reduce their energy. Only after reaching a very low energy after many scattering events do we expect these neutrons to cause more fission reactions to occur, continuing the chain reaction. This makes careful handling of the scattering operator of particular interest.

The scattering reaction rate, as seen in the scattering operator,

$$\Sigma_s(\vec{r}, E' \to E, \hat{\Omega}' \to \hat{\Omega})\psi(\vec{r}, \hat{\Omega}', E') \ ,$$

provides the rate of generation of neutrons of energy E and direction of motion $\hat{\Omega}$, due to incoming neutrons of energy E' and direction of motion $\hat{\Omega}'$. We assume that the scattering is axially symmetrical: the actual incoming and outgoing directions of motion do not matter, only the angle between them, $\mu_0 = \hat{\Omega}' \cdot \hat{\Omega} = \cos(\theta)$. For many systems of interest this assumption is valid and introduces minimal error.

The scattering cross-section is complex and, depending on the material and physical layout of the system, neutrons can scatter with varying degrees of anisotropy. We therefore expand the scattering cross-section using a functional expansion that enables us to represent these scattering modes numerically. The most commonly used way to accomplish this is to expand the scattering operator using the Legendre polynomials. This is equivalent to expanding the scattering operator using the spherical harmonics [1] with $m = 0$, which is an orthogonal set of functions that solve Laplace's equation on a sphere. In addition, as shown in Appendix A, the spherical harmonics are eigenfunctions of the scattering operator.

Thus, we expand the multigroup scattering differential cross-section as an infinite sum of the Legendre polynomials,

$$\Sigma_s^{g'\to g}(\vec{r}, \hat{\Omega}' \cdot \hat{\Omega}) = \sum_{\ell=0}^{\infty} \frac{2\ell+1}{4\pi} \Sigma_{s\ell}^{g'\to g}(\vec{r}) P_\ell(\hat{\Omega}' \cdot \hat{\Omega}) \ ,$$

where the scattering cross-section moments are given by

$$\Sigma_{s\ell}^{g'\to g}(\vec{r}) = 2\pi \int_{4\pi} \Sigma_s^{g'\to g}(\vec{r}, \hat{\Omega}' \cdot \hat{\Omega}) P_\ell(\hat{\Omega}' \cdot \hat{\Omega}) d\hat{\Omega} \ .$$

Practically, we must bound expansion when solving systems numerically. We typically choose a maximum value of $\ell_{\max} \in [3, 5]$, which efficiently captures the dominant modes of scattering. In the special case of only isotropic scattering, only the 0th moment of the scattering cross-section is nonzero. In this case, Eq. (2.5) reduces to

$$\begin{aligned}\left[\hat{\Omega} \cdot \nabla + \Sigma_t^g(\vec{r})\right] \psi_g(\vec{r}, \hat{\Omega}) dE = &\sum_{g'=0}^{G-1} \Sigma_{s0}^{g'\to g}(\vec{r}) \phi_{g'}(\vec{r}) \\ &+ \frac{\chi_g}{k} \sum_{g'=0}^{G-1} \nu_{g'} \Sigma_f^{g'}(\vec{r}) \phi_{g'}(\vec{r}) + Q_g(\vec{r}, \hat{\Omega}) \ , \end{aligned} \tag{2.6}$$

where

$$\phi_{g'}(\vec{r}) = \int_{4\pi} d\hat{\Omega}' \psi_{g'}(\vec{r}, \hat{\Omega}') \ .$$

2.2.3 Angle Discretization

The angular solution domain for the transport equation is the unit sphere,

$$\mathbf{\Omega} = \left\{ \hat{\Omega} \in \mathbb{R}^2 : ||\hat{\Omega}||_2 = 1 \right\} \, .$$

To discretize this space we will use a collocation method commonly referred to in this context as the *discrete ordinates method* [1]. We enforce the solution to the transport equation along a particular set of angles. There must be enough points to sweep all of the unit sphere, the number of which varies with the dimension in which the problem is being solved as well as the physics properties of the system. In addition, symmetry in the choice of angles is usually desired for two reasons. First, so that no particular direction or axis is preferred. Second, some reflective boundary conditions will require angles to have a reflected counterpart. Using a quadrature set in this way is known as an S_N method, where N is the order of the quadrature set.

In addition, these angles and corresponding integration weights are chosen such that they form a quadrature set that we can use to integrate the angular flux on the unit sphere to calculate the scalar flux. For a quadrature set with N quadrature points,

$$\phi_g(\vec{r}) = \int_{4\pi} \psi_g(\vec{r}, \hat{\Omega}) d\hat{\Omega} \approx \sum_{n=0}^{N-1} w_n \psi_g(\hat{\Omega}_n) \, ,$$

where $(w_n, \hat{\Omega}_n)$ are the ordinate and weight pairs in the quadrature set. The quadrature set should also integrate exactly the spherical harmonics and Legendre polynomials. This will enable calculation of scattering moments from the previous section, as well as the Legendre moments of the scalar flux.

A frequently used quadrature set for the discrete ordinates method is the *level-symmetric set*. Use of this set has been historically very successful and a full description can be found in many sources [2, 1]. Although this set fits all the requirements described above for a quadrature set, the ordinate and weight values are not procedurally generated. These values must be provided in a table and therefore do not provide the ability for an arbitrary order quadrature set. Ideally, a quadrature set could be generated for any arbitrary order while still meeting the requirements above. One set that accomplishes both of these is a level-symmetric-like Gaussian quadrature set, described in Chapter 5.

Using this or any other quadrature set results in a set of N equations, where N is the number of quadrature points in our set. For example, the isotropic scattering multigroup equation shown in Eq. (2.6) is now,

$$\begin{aligned}\left[\hat{\Omega}_n \cdot \nabla + \Sigma_t^g(\vec{r})\right] \psi_g^n(\vec{r}, \hat{\Omega}_n) dE &= \sum_{g'=0}^{G-1} \Sigma_{s0}^{g' \to g}(\vec{r}) \phi_{g'}(\vec{r}) \\ &\quad + \frac{\chi_g}{k} \sum_{g'=0}^{G-1} \nu_{g'} \Sigma_f^{g'}(\vec{r}) \phi_{g'}(\vec{r}) \, . \qquad (2.7)\end{aligned}$$

Accounting for more angles and energy groups will improve the resolution and accuracy of our solution, an incentive to using as many of each as possible. Unfortunately, for every energy group we must solve all the angles, resulting in scaling of $\mathcal{O}(N \times G)$. The multiplicative nature of the number of equations is further exacerbated by discretizing the spatial domain of the phase space. This is a more complex process and depends greatly on the means by which we will solve the transport equation.

2.3 Discretization of Space

Choices of spatial discretization are motivated by the methodology one will use to solve the problem. In this section, we will motivate the discretization of differential equations to enable the use of a finite element method (FEM). The derivations and methodology in this section can be found in many FEM books, including *Numerical Solution of Partial Differential Equations by the Finite Element Method* by Claes Johnson [7].

To describe the FEM methodology, we will consider a one-dimensional, bounded Poisson problem

$$-\frac{d^2u(x)}{dx^2} = 1, \forall x \in (0,1) \ , \tag{2.8}$$

with boundary conditions,

$$u(0) = u(1) = 0 \ . \tag{2.9}$$

This problem has an exact solution

$$u(x) = \frac{x}{2}\left(1 - x\right) \ , \tag{2.10}$$

but we will assume this is unknown and seek an approximate solution $\hat{u}(x)$. One method to approximate the solution to Eq. (2.8) is to impose the average of our approximation over the interval by integrating

$$-\int_0^1 \frac{d^2\hat{u}(x)}{dx^2} dx = \int_0^1 dx \ . \tag{2.11}$$

A better average will be achieved by a weighted average achieved by using a well-chosen weighting function $v(x)$,

$$-\int_0^1 \frac{d^2\hat{u}(x)}{dx^2} v(x) dx = \int_0^1 v(x) \ dx \ . \tag{2.12}$$

Choice of this weighing function will have a large influence on the quality of our approximation.

Galerkin methods are those that chose $v(x)$ from the same solution space as our approximation. For the Poisson problem of this form, one good approximation is the sine function. To this end, we can seek our approximate solution in the space,

$$V = \{k \sin(\pi x) : k \in \mathbb{R}\} \ ,$$

where the weighing function is also a member of this space. The choice of weighing function is arbitrary, and therefore our approximation must hold for all possible values.

We will seek an approximate solution $\hat{u}(x) \in V$ by plugging it into Eq. (2.12) and requiring that it hold for all possible values of $v(x)$,

$$-\int_0^1 \frac{d^2}{dx^2}\left[k\sin(\pi x)\right] v(x)dx = \int_0^1 v(x)\ dx, \quad \forall v \in V\ . \tag{2.13}$$

As the choice of $v(x)$ is arbitrary, we will chose the simplest version, $v(x) = \sin(\pi x)$ and solve for k:

$$\int_0^1 k\pi^2 \sin^2(\pi x) v(x)dx = \int_0^1 \sin(\pi x)\ dx. \tag{2.14}$$

This ultimately yields a value of $k = 0.129$, which is an excellent approximation of the Poisson equation.

2.3.1 Basis Functions

In the previous section, we described approximating the Poisson equation using a Galerkin method and a set of good basis functions. The choice of basis functions was motivated by an *a priori* understanding of what functions would provide a good approximation. In most cases, we do not have this information and will need to arbitrarily choose a set of good basis functions to use for our Galerkin method.

We must first discretize the solution space. We will divide our solution space, $\mathcal{D}$, into a triangulation[3] of a finite number of non-overlapping elements that cover the entire space,

$$T_h = \{K_0, K_1, \ldots, K_{H-1}\}\,, \quad \text{s.t. } \mathcal{D} = \bigcup_{i=0}^{H-1} K_i\ .$$

We now define a space for our solution as a space of continuous functions that are piecewise linear on each element,

$$V_h = \left\{v \in C^0(\mathcal{D}) : v|_K \in \mathbb{P}_1(K) \forall K \in T_h,\ v(\partial\mathcal{D}) = f(\partial\mathcal{D})\right\}\ , \tag{2.15}$$

where $C^0(\mathcal{D})$ is the space of continuous functions on our solution domain, and $\mathbb{P}_1(K)$ is the space of polynomials that are linear on the element K, and $f(\partial\mathcal{D})$ is a function describing our boundary conditions (only Dirichlet boundary conditions are shown here for simplicity).

If our solution is a member of this space, it is linear on each element and is therefore uniquely described by the values of the function on the vertices of all the elements in our triangulation. If our triangulation of H elements has M total vertices, our approximate solution $u_h(\vec{r})$ is therefore

$$u_h(\vec{r}) = \sum_{i=0}^{M} u_i \varphi_i(\vec{r})\ , \tag{2.16}$$

[3]We will use the term triangulation to describe the discretization of the solution space into non-overlapping elements, even when they are not triangular in shape.

where u_i is the value of our approximate solution at vertex i and $\varphi_i(\vec{r})$ is a function that is unity at that vertex, and zero everywhere else. If the position of vertex j is $\vec{r}_j$,

$$\varphi_i(\vec{r}_j) = \delta_{ij}, \quad u_h(\vec{r}_i) = u_i, \quad \forall i, j \in 0, \ldots, M \ .$$

These functions $\varphi_i(\vec{r})$ form a basis for our solution space V_h. An example basis function is shown in Fig. 2.2. Now equipped with a solution space on which to seek our solution and a basis to describe this space, we will return to our general Galerkin method.

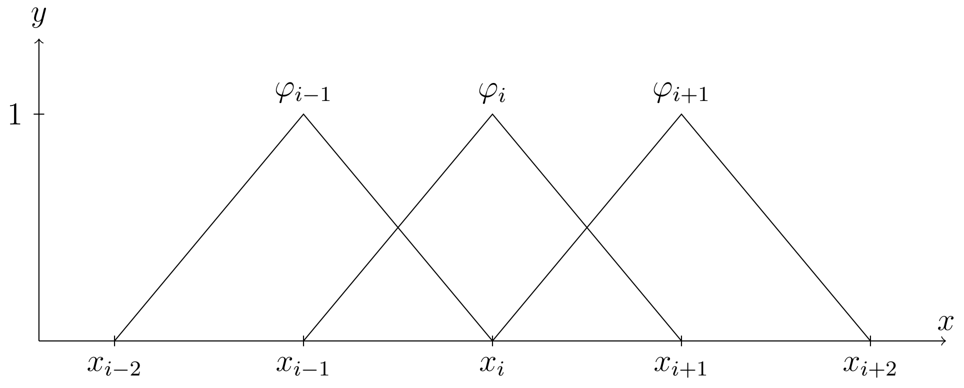


Figure 2.2: One-dimensional basis functions, $\varphi(x)$.

2.3.2 The Weak Formulation

The Galerkin formulation for our model problem, Eq. (2.8), uses the weighted average form of Eq. (2.12) and our solution space from Eq. (2.15). For this one-dimensional case with the given boundary conditions, we will be seeking an approximate solution $u_h(x)$ in the space,

$$V_h = \left\{ v \in C^0([0,1]) : v|_K \in \mathbb{P}_1(K) \forall K \in T_h, \ v(0) = v(1) = 0 \right\} \ ,$$

such that

$$-\int_0^1 \frac{d^2 u_h(x)}{dx^2} v(x) dx = \int_0^1 v(x) \ dx, \quad \forall v(x) \in V_h \ .$$

Our choice of $v(x)$ is arbitrary, as this must hold for all possible functions in our solution space. We will therefore pick the simplest value: our weighing function is equal to our basis, $v(x) = \varphi(x)$,

$$-\int_0^1 \frac{d^2 u_h(x)}{dx^2} \varphi(x) dx = \int_0^1 \varphi(x) \ dx \ . \tag{2.17}$$

We will first use Green's Formula,

$$\oint_{\partial\mathcal{D}} v(\vec{r}) \hat{n} \cdot \nabla u(\vec{r}) ds = \int_{\mathcal{D}} \nabla v(\vec{r}) \cdot \nabla u(\vec{r}) \ d\vec{r} + \int_{\mathcal{D}} v(\vec{r}) \nabla^2 u(\vec{r}) d\vec{r} \ , \tag{2.18}$$

which in one-dimension on our domain simplifies to integration by parts,

$$\int_0^1 \frac{d^2u(x)}{d^2x} v(x) dx = \left[\frac{du(x)}{dx} v(x)\right]_0^1 - \int_0^1 \frac{du(x)}{dx}\frac{dv(x)}{dx} dx \; . \tag{2.19}$$

We plug the expansion of Eq. (2.19) into our formulation Eq. (2.17),

$$\left[\frac{du_h(x)}{dx}\varphi(x)\right]_0^1 - \int_0^1 \frac{du_h(x)}{dx}\frac{d\varphi(x)}{dx} dx = \int_0^1 \varphi(x)\; dx \; .$$

By our definition of the space, all functions including the basis go to zero on the boundaries, $\varphi(0) = \varphi(1) = 0$, eliminating the first term and leaving us with

$$\int_0^1 \frac{du_h(x)}{dx}\frac{d\varphi(x)}{dx} dx = \int_0^1 \varphi(x)\; dx \; .$$

This is the weak formulation of our problem that provides a weak solution. Our original problem required the second derivative of $u(x)$ to exist and be continuous, while this formulation only requires that of the first derivative. This is what makes this formulation weak.

For our given triangulation T, we have M equations, one for each vertex basis function, $\varphi_j = \varphi(\vec{r}_j)$,

$$\int_0^1 \frac{du_h(x)}{dx}\frac{d\varphi_j}{dx} dx = \int_0^1 \varphi_j\; dx, \quad j = 0, 1, \ldots, M-1 \; .$$

We can also substitute in our definition of $u_h(x)$ as a linear combination of the basis functions from Eq. (2.16),

$$\sum_{i=0}^{M-1} u_i \int_0^1 \frac{d\varphi_i}{dx}\frac{d\varphi_j}{dx} dx = \int_0^1 \varphi_j\; dx, \quad j = 0, 1, \ldots, M-1 \; . \tag{2.20}$$

To simplify notation, we will use inner product notation to indicate integration of the problem domain,

$$\begin{aligned} \Big(u, v\Big)_{\mathcal{D}} &\equiv \int_{\mathcal{D}} u\; v\; d\vec{r} \\ \Big\langle u, v\Big\rangle_{\partial\mathcal{D}} &\equiv \oint_{\partial\mathcal{D}} u\; v\; ds \; , \end{aligned}$$

and prime to indicate spatial derivatives.

Using this notation for our model problem, Eq. (2.20) becomes

$$\sum_{i=0}^{M-1} u_i \left(\varphi_i', \varphi_j'\right)_{\mathcal{D}} = (1, \varphi_j)_{\mathcal{D}} \, , \quad j = 0, \ldots, M-1 \; .$$

We now note that this is a linear system of M equations and M unknowns (the values of u_i), that can be represented in matrix form as,

$$\mathbf{A}\vec{u} = \vec{b}\ ,$$

where $\mathbf{A}$ is an $M \times M$ matrix and vectors $\vec{b}$ and $\vec{u}$ are $M \times 1$ vectors with entries,

$$\begin{aligned} \mathbf{A}_{ij} &= \left(\varphi_i', \varphi_j'\right)_{\mathcal{D}} \\ \vec{b}_i &= (1, \varphi_j)_{\mathcal{D}} \\ \vec{u}_i &= u_i\ . \end{aligned}$$

Consequently, we note that this matrix and vector are only dependent on the solution space basis functions and the formulation of our problem. In addition, unless the indices i and j are on the same element, at least one of the basis functions will be zero. The matrix $\mathbf{A}$ is therefore quite sparse and the sparsity pattern will depend on the indexing and spatial dimension of the problem.

2.3.3 Assembling the System Matrix

Explicitly creating the matrix $\mathbf{A}$ and vector $\vec{b}$ can be complicated, especially if the mesh is irregular and complex. One practical way to build the matrix is to iterate over all the cells in the triangulation and calculate the value of $\mathbf{A}_{ij}$ and $\vec{b}_i$ at each local degree of freedom. These local cell values are added to the system matrix by mapping the local index to the global system index. Multiple cells share degrees of freedom, and each will contribute to that degree of freedom's value in the system matrix.

To calculate the local contributions, we must integrate the local basis functions. We will accomplish this computationally by using a quadrature rule, such as Gaussian quadrature, on the cell. The basis functions will be evaluated on cell K for each quadrature point consisting of an ordinate and a weight,

$$\int_K f(\vec{r})d\vec{r} \approx \sum_{q=0}^{Q-1} w_q f(\vec{r}_q)\ ,$$

where $\vec{r}_q$ are the ordinates, and w_q are the weights. Note that this is a spatial quadrature that integrates exactly on each cell and necessarily is different from the quadrature over the unit sphere used in the angular discretization.

For square cells, we will perform this integration on a reference cell on the domain $[0,1]^d$ where d is the spatial dimension of the triangulation. We will then map the value back to the actual cell K using a mapping function,

$$f_K : [0,1]^d \mapsto K \in R^d\ .$$

The quadrature formulation will then take the form,

$$\int_K f(\vec{r})d\vec{r} \approx \sum_{q=0}^{Q-1} w_q f(\vec{r}_q)|\mathbf{J}(f_K)|\ ,$$

where $|\mathbf{J}(f_K)|$ is the determinant of the Jacobian matrix of the transformation. A consequence of this is that the integration only needs to be calculated once for the reference cell and then mapped to each cell's local contribution for stamping on the system matrix.

2.3.4 General Form

Our description of the finite element method has been limited to our model problem, but it can be applied to more generalized partial differential equations (PDEs). This is important for application to problems that are not the model problem, such as the NTE. We will examine a generalized time-independent PDE with derivatives up to second order, noting the general names for each term. Note that this is not a specific equation, but the application of a FEM to a generalized equation.

$$\underbrace{-D\nabla^2 u(\vec{r})}_{\text{Diffusion term}} + \underbrace{\hat{\Omega}\cdot\nabla u(\vec{r})}_{\text{Convection term}} + \underbrace{\Sigma u(\vec{r})}_{\text{Reaction term}} = \underbrace{Q(\vec{r})}_{\text{Source term}} \quad , \quad \forall \vec{r} \in \mathcal{D} \ ,$$

with boundary conditions,

$$\begin{aligned} \hat{n}\cdot\nabla u(\vec{r}) + \hat{n}\cdot u = f(\vec{r}), &\quad \forall \vec{r} \in \partial\mathcal{D}_N \\ u(\vec{r}) = f_D(\vec{r}), &\quad \forall \vec{r} \in \partial\mathcal{D}_D \ , \end{aligned}$$

where $\mathcal{D}_N$ and $\mathcal{D}_D$ are the portions of the domain boundary with Neumann and Dirichlet boundary conditions, respectively, with respective functional forms $f(\vec{r})$ and $f_D(\vec{r})$.

From this generalized form we can define many of the well-known PDEs: the Laplace equation only has the diffusion term, the Poisson equation (our model problem) adds a source term, and the Helmholtz equation adds a reaction term on to that. Our NTE is a convection-reaction equation with a source term, but in the first-order form lacks the diffusion term. Adding time dependence allows us to also include time-dependent forms of PDEs like the wave equation and heat equation, which is beyond the scope of this work.

A major benefit of discretizing diffusive problems using the FEM is that they, in general, result in sparse symmetric positive-definite system matrices. This enables the use faster and more efficient solving algorithms, even for complex or unstructured meshes. Different diffusive, or second-order, forms of the transport equation have been developed. These include the even- and odd-parity formulation [1], the diffusion formulation, the weighted least-square formulation [8], and the self-adjoint angular-flux formulation [9]. We will examine the discretization of the diffusion and SAAF formulations in the following sections.

2.4 Diffusion Equation

The diffusion equation is a well known and understood second-order form of the transport equation. This formulation treats neutron propagation as diffusive, much like the heat equation. A full derivation and asymptotic analysis of the diffusion equation can be found in many standard texts [1, 2, 3]. The process of deriving the diffusion formulation in general involves integrating the transport equation over angle, and using a form of Fick's law as a closure for the current,

$$\vec{J}_g(\vec{r}) = -D_g(\vec{r})\nabla \cdot \phi_g(\vec{r}) \ , \tag{2.21}$$

where $\vec{J}_g$ is the current and D_g is the diffusion coefficient, each for group g. The current represents a net rate at which neutrons are passing through a unit area in a specific direction. The diffusion coefficient is related to the total cross-section and the first moment of the scattering cross-section,

$$D_g(\vec{r}) \ \frac{1}{3\left(\Sigma_t^g(\vec{r}) - \Sigma_{s1}^g(\vec{r})\right)} \ .$$

Using this, the standard form of the k-eigenvalue multigroup diffusion equation is

$$\left[-D_g(\vec{r})\nabla^2 + \Sigma_t^g(\vec{r})\right]\phi_g(\vec{r}) = \sum_{g'=0}^{G-1} \Sigma_{s0}^{g'\to g}\phi_{g'}(\vec{r}) + \frac{\chi_g}{k}\sum_{g'=0}^{G-1} \nu_{g'}\Sigma_f^{g'}\phi_{g'}(\vec{r}) \ . \tag{2.22}$$

We will now derive the weak form of the diffusion equation for solving using finite element methods. In doing so, we will prepare for iterative solving techniques, moving the within-group scattering term to the left-hand-side, as the other groups will be treated as a source term from the previous iteration. This will be discussed further in Chapter 3

2.4.1 Weak Form of the Diffusion Equation

We begin with Eq. (2.22) and bring the within-group scattering term to the right hand side, then multiply through on the left by an arbitrary spatial weight function, $v(\vec{r})$, and integrate over the spatial domain.

$$\Big(v(\vec{r}), -D_g(\vec{r})\nabla^2\phi_g(\vec{r})\Big)_{\mathcal{D}} + \Big(v(\vec{r}), \Sigma_r^g(\vec{r})\phi_g(\vec{r})\Big)_{\mathcal{D}} = \\ \Big(v(\vec{r}), \sum_{g'\neq g}\Sigma_{s0}^{g'\to g}\phi_{g'}(\vec{r})\Big)_{\mathcal{D}} + \Big(v(\vec{r}), \frac{\chi_g}{k}\sum_{g'=0}^{G-1}\nu_{g'}\Sigma_f^{g'}\phi_{g'}(\vec{r})\Big)_{\mathcal{D}}$$

where $\Sigma_r^g(\vec{r}) = \Sigma_t^g(\vec{r}) - \Sigma_{s0}^{g\to g}(\vec{r})$ is the removal cross-section and the inner product notation is as described in the previous section. Applying Green's formula, Eq. (2.18), to the first term gives

$$\Big(\nabla v(\vec{r}), D_g(\vec{r})\nabla\phi_g(\vec{r})\Big)_{\mathcal{D}} - \Big\langle v(r), \hat{n}\cdot D_g(\vec{r})\nabla\phi_g(\vec{r})\Big\rangle_{\partial\mathcal{D}} + \Big(v(\vec{r}), \Sigma_r^g(\vec{r})\phi_g(\vec{r})\Big)_{\mathcal{D}} = \\ \Big(v(\vec{r}), \sum_{g'\neq g}\Sigma_{s0}^{g'\to g}\phi_{g'}(\vec{r})\Big)_{\mathcal{D}} + \Big(v(\vec{r}), \frac{\chi_g}{k}\sum_{g'=0}^{G-1}\nu_{g'}\Sigma_f^{g'}\phi_{g'}(\vec{r})\Big)_{\mathcal{D}} \ ,$$

where $\hat{n}$ is the outward normal on the boundary. Using our Fick's law definition in Eq. (2.21), and the definition of net current we can say

$$\vec{J}_{\text{net}}(\vec{r}) \equiv \hat{n} \cdot \vec{J}(\vec{r}) = -\hat{n} \cdot D_g(\vec{r})\nabla\phi_g(\vec{r}) \ ,$$

and then substitute the net current into the boundary term:

$$\Big(\nabla v(\vec{r}), D_g(\vec{r})\nabla\phi_g(\vec{r})\Big)_{\mathcal{D}} + \Big\langle v(r), \vec{J}_{\text{net}}(\vec{r})\Big\rangle_{\partial\mathcal{D}} + \Big(v(\vec{r}), \Sigma_r^g(\vec{r})\phi_g(\vec{r})\Big)_{\mathcal{D}} = \\ \Big(v(\vec{r}), \sum_{g'\neq g}\Sigma_{s0}^{g'\to g}\phi_{g'}(\vec{r})\Big)_{\mathcal{D}} + \Big(v(\vec{r}), \frac{\chi_g}{k}\sum_{g'=0}^{G-1}\nu_{g'}\Sigma_f^{g'}\phi_{g'}(\vec{r})\Big)_{\mathcal{D}} \ .$$

For reflective boundary conditions, this term is zero because all outgoing flux is matched by reflected incoming flux resulting in zero current. For vacuum boundary conditions, we need to enforce zero incoming flux. We will do so using Marshak boundary conditions, fully derived in Bell and Glasstone [2]. Using these boundary conditions, the requirement of zero incoming current is represented by

$$\hat{n} \cdot \vec{J}(\vec{r}) = \frac{1}{2}\phi(\vec{r}) \ . \tag{2.23}$$

Substituting this in to our weak form gives us the final weak form for the diffusion equation,

$$\Big(\nabla v(\vec{r}), D_g(\vec{r})\nabla\phi_g(\vec{r})\Big)_{\mathcal{D}} + \Big\langle v(r), \frac{1}{2}\phi_g(\vec{r})\Big\rangle_{\partial\mathcal{D},\text{vacuum}} + \Big(v(\vec{r}), \Sigma_r^g(\vec{r})\phi_g(\vec{r})\Big)_{\mathcal{D}} = \\ \Big(v(\vec{r}), \sum_{g'\neq g}\Sigma_{s0}^{g'\to g}\phi_{g'}(\vec{r})\Big)_{\mathcal{D}} + \Big(v(\vec{r}), \frac{\chi_g}{k}\sum_{g'=0}^{G-1}\nu_{g'}\Sigma_f^{g'}\phi_{g'}(\vec{r})\Big)_{\mathcal{D}} \ .$$

2.4.2 Benefits of the Diffusion Equation

The diffusion equation has a number of benefits that we will leverage. The angular dependence has been integrated out, which significantly reduces the phase space required to solve the problem. The diffusive nature of the formulation means that problems with high scattering will solve much more quickly than the NTE. This has been leveraged in many acceleration methods to speed up the convergence of problems, discussed further in Chapter 3. The formulation does not have a convective term, and so the matrix generated by the weak formulation is symmetric and sparse, allowing us to solve with more efficient methods. Some of these structural benefits can be found in a formulation that retains angular information, such as the self-adjoint angular flux formulation, described in the next section.

2.5 Self-Adjoint Angular-Flux Equation

The SAAF equation [9] is a second-order self-adjoint form of the transport equation much like the well-understood even- and odd-parity forms. This particular form of the second-order transport equation is derived algebraically from the first-order form, and presents a set of benefits and drawbacks compared to the parity forms. One of the major benefits of second-order forms such as the SAAF formulation is that FEM can be more easily applied than first-order forms to unstructured meshes and those with re-entrant cells. In this section, we shall derive the SAAF equation and discuss the advantages and disadvantages of its use.

2.5.1 Derivation

We will derive the SAAF equation from a monoenergetic form of Eq. (2.1),

$$\hat{\Omega}\cdot\nabla\psi(\vec{r},\hat{\Omega}) + \Sigma_t\psi(\vec{r},\hat{\Omega}) = S\psi(\vec{r},\hat{\Omega}) + q(\vec{r},\hat{\Omega}) \;, \tag{2.24}$$

where the source q can be the eigenvalue term or a fixed source and the scattering term is the monoenergetic form of the differential cross-section

$$S\psi(\vec{r},\hat{\Omega}) = \int_{4\pi}\Sigma_s(\vec{r},\hat{\Omega}'\to\hat{\Omega})d\hat{\Omega}\psi(\vec{r},\hat{\Omega}') \;.$$

To derive the second-order form from the first-order form, we will solve for the angular flux as a function of the first-order term and substitute this back into the first-order term, resulting in a second-order formulation. First, we will re-order Eq. (2.24) to solve for the angular flux,

$$\begin{aligned}
\Sigma_t\psi(\vec{r},\hat{\Omega}) - S\psi(\vec{r},\hat{\Omega}) &= -\hat{\Omega}\cdot\nabla\psi(\vec{r},\hat{\Omega}) + q(\vec{r},\hat{\Omega}) \\
(\Sigma_t - S)\,\psi(\vec{r},\hat{\Omega}) &= -\hat{\Omega}\cdot\nabla\psi(\vec{r},\hat{\Omega}) + q(\vec{r},\hat{\Omega}) \\
\psi(\vec{r},\hat{\Omega}) &= -\left(\Sigma_t - S\right)^{-1}\hat{\Omega}\cdot\nabla\psi(\vec{r},\hat{\Omega}) + \left(\Sigma_t - S\right)^{-1}q(\vec{r},\hat{\Omega}) \;.
\end{aligned} \tag{2.25}$$

We then substitute Eq. (2.25) back into the first term of Eq. (2.24),

$$\begin{aligned}
\hat{\Omega}\cdot\nabla\bigg[-\left(\Sigma_t - S\right)^{-1}\hat{\Omega}\cdot\nabla\psi(\vec{r},\hat{\Omega}) + \left(\Sigma_t - S\right)^{-1}q(\vec{r},\hat{\Omega})\bigg] &+ \Sigma_t\psi(\vec{r},\hat{\Omega}) \\
&= S\psi(\vec{r},\hat{\Omega}) + q(\vec{r},\hat{\Omega}) \\
-\hat{\Omega}\cdot\nabla\left(\Sigma_t - S\right)^{-1}\hat{\Omega}\cdot\nabla\psi(\vec{r},\hat{\Omega}) + \left(\Sigma_t - S\right)\psi(\vec{r},\hat{\Omega}) & \\
&= q(\vec{r},\hat{\Omega}) - \hat{\Omega}\cdot\nabla\left(\Sigma_t - S\right)^{-1}q(\vec{r},\hat{\Omega}) \;.
\end{aligned} \tag{2.26}$$

Naturally, the second-order form of the transport equation will require two boundary conditions. For incoming flux, we will enforce a Dirichlet boundary condition, which is standard for the first-order form:

$$\psi(\vec{r},\hat{\Omega}) = f(\vec{r},\hat{\Omega}), \quad \vec{r}\in\partial\mathbb{D}, \quad \hat{\Omega}\cdot\hat{n} < 0 \;. \tag{2.27}$$

This is the term that will require the most modification depending on the type of boundary condition used. For vacuum boundaries, we will set $f(\vec{r}, \hat{\Omega}) = 0$ and for reflective boundaries $f(\vec{r}, \hat{\Omega}) = \psi(\vec{r}, \hat{\Omega}')$ where $\hat{\Omega}'$ is the reflected angle.

For the second boundary condition, we will enforce Robin boundary conditions on the outgoing flux in the form of the first-order form itself,

$$\hat{\Omega} \cdot \nabla \psi(\vec{r}, \hat{\Omega}) + \Sigma_t \psi(\vec{r}, \hat{\Omega}) = S\psi(\vec{r}, \hat{\Omega}) + q(\vec{r}, \hat{\Omega}), \quad \vec{r} \in \partial\mathbb{D}, \quad \hat{\Omega} \cdot \hat{n} > 0 \,. \tag{2.28}$$

For S_N calculations, we will derive the SAAF equation starting with the multigroup form of the transport equation with isotropic scattering. We start with Eq. (2.7) with the fission term represented as q,

$$\left[\hat{\Omega}_n \cdot \nabla + \Sigma_t^g(\vec{r})\right] \psi_g^n(\vec{r}, \hat{\Omega}_n) dE = \sum_{g'=0}^{G-1} \Sigma_{s0}^{g' \to g}(\vec{r}) \phi_{g'}(\vec{r}) + q_g^n(\vec{r}, \hat{\Omega}_n) \,.$$

Following the same steps as the derivation for the monoenergetic equation, we get the form of the SAAF equation appropriate for multigroup S_N calculations,

$$\begin{aligned} -\hat{\Omega}_n \cdot \nabla \frac{1}{\Sigma_t} \hat{\Omega}_n \cdot \nabla \psi_g^n(\vec{r}, \hat{\Omega}_n) + \Sigma_t \psi_g^n(\vec{r}, \hat{\Omega}_n) & \\ = \sum_{g'=0}^{G-1} \Sigma_{s0}^{g' \to g} \phi_{g'}(\vec{r}) + q_g^n(\vec{r}, \hat{\Omega}_n) - \hat{\Omega}_n \cdot \nabla \frac{\sum_{g'=0}^{G-1} \Sigma_{s0}^{g' \to g} \phi_{g'}(\vec{r}) + q_g^n(\vec{r}, \hat{\Omega}_n)}{\Sigma_t} \,. & \end{aligned} \tag{2.29}$$

2.5.2 Properties of the SAAF Equation

Many important properties of the SAAF form of the transport equation are fully described by Morel and McGhee [9]; some of the important properties will be summarized here. The SAAF equation can also be derived as a version of the weighted least-square transport equation with a weighing factor of Σ_t^{-1} [8], so the properties of these formulations are also applicable.

As discussed in Section 2.3, second-order forms of the transport equation like the SAAF equation lead to sparse, symmetric matrices when using standard finite-element spatial discretization, enabling the use of more efficient solvers. This contrasts with the block dense lower-triangular matrix formed by a first-order form of the transport equation. The block dense matrix pattern lends itself to stepping through the mesh and solving cells sequentially, implicitly inverting the transport operator in a method called sweeping. Using a FEM with a second-order form enables us to use unstructured meshes and re-entrant cells, which are challenges for a sweeping method. In addition, unlike other second-order forms of the transport equation, the full angular flux is the unknown in the SAAF equation. Calculating the full angular flux when using other forms such as the even- and odd-parity equations can be challenging, and the SAAF formulation avoids the issue entirely.

A disadvantage of the SAAF equation is the presence of the inverse of the total cross-section. This makes solving the equation in voids, where the total cross-section is zero, impossible. There are techniques that can be applied to make the SAAF and other forms of the least-squares transport equation compatible with void regions [8, 10], but we will not discuss them here.

2.5.3 Weak Form of the SAAF Equation

To solve the SAAF equation with a finite element method, we must derive the weak form, as described in the previous section. We begin with the self-adjoint angular flux equation appropriate for S_N calculations, Eq.(2.29), and drop the angle and group index, and reintroduce the scattering operator for conciseness:

$$-\hat{\Omega}\cdot\nabla\frac{1}{\Sigma_t}\hat{\Omega}\cdot\nabla\psi(\vec{r},\hat{\Omega})+\Sigma_t\psi(\vec{r},\hat{\Omega})=S\psi(\vec{r},\hat{\Omega})+q(\vec{r},\hat{\Omega})-\hat{\Omega}\cdot\nabla\frac{S\psi(\vec{r},\hat{\Omega})+q(\vec{r},\hat{\Omega})}{\Sigma_t}\,. \tag{2.30}$$

Multiplying through on the left by an arbitrary spatial weight function $v(\vec{r})$ and integrating over the angular and spatial domain, we get

$$\begin{aligned}\int d\hat{\Omega}\int_{\mathcal{D}}d\vec{r}\,v(\vec{r})\Big(-\hat{\Omega}\cdot\nabla\frac{1}{\Sigma_t}\hat{\Omega}\cdot\nabla\psi(\vec{r},\hat{\Omega})\Big)+\int d\hat{\Omega}\int_{\mathcal{D}}d\vec{r}\,v(\vec{r})\Sigma_t\psi(\vec{r},\hat{\Omega})\\=\int d\hat{\Omega}\int_{\mathcal{D}}d\vec{r}\,v(\vec{r})\left(S\psi(\vec{r},\hat{\Omega})+q(\vec{r},\hat{\Omega})-\hat{\Omega}\cdot\nabla\frac{S\psi(\vec{r},\hat{\Omega})+q(\vec{r},\hat{\Omega})}{\Sigma_t}\right).\end{aligned} \tag{2.31}$$

First, we will collect all the terms in Eq. (2.31) on one side of the equal sign and group the terms with $\hat{\Omega}\cdot\nabla$,

$$\begin{aligned}\int d\hat{\Omega}\int_{\mathcal{D}}d\vec{r}\,v(\vec{r})\Big(-\hat{\Omega}\cdot\nabla\frac{1}{\Sigma_t}\hat{\Omega}\cdot\nabla\psi(\vec{r},\hat{\Omega})+\hat{\Omega}\cdot\nabla\frac{S\psi(\vec{r},\hat{\Omega})+q(\vec{r},\hat{\Omega})}{\Sigma_t}\Big)\\+\int d\hat{\Omega}\int_{\mathcal{D}}d\vec{r}\,v(\vec{r})\Sigma_t\psi(\vec{r},\hat{\Omega})-\int d\hat{\Omega}\int_{\mathcal{D}}d\vec{r}\,v(\vec{r})\Big(S\psi(\vec{r},\hat{\Omega})+q(\vec{r},\hat{\Omega})\Big)=0\,.\end{aligned} \tag{2.32}$$

The inner product is preserved under commutation and association, so we will reverse the order and factor out the del operator from this combined term,

$$\begin{aligned}\int d\hat{\Omega}\int_{\mathcal{D}}d\vec{r}\,v(\vec{r})\left(-\hat{\Omega}\cdot\nabla\frac{1}{\Sigma_t}\hat{\Omega}\cdot\nabla\psi(\vec{r},\hat{\Omega})+\hat{\Omega}\cdot\nabla\frac{S\psi(\vec{r},\hat{\Omega})+q}{\Sigma_t}\right)=\\\int d\hat{\Omega}\int_{\mathcal{D}}d\vec{r}\,v(\vec{r})\left(-\nabla\cdot\hat{\Omega}\frac{1}{\Sigma_t}\hat{\Omega}\cdot\nabla\psi(\vec{r},\hat{\Omega})+\nabla\cdot\hat{\Omega}\frac{S\psi(\vec{r},\hat{\Omega})+q}{\Sigma_t}\right)=\\\int d\hat{\Omega}\int_{\mathcal{D}}d\vec{r}\,v(\vec{r})\left[\nabla\cdot\left(-\hat{\Omega}\frac{1}{\Sigma_t}\hat{\Omega}\cdot\nabla\psi(\vec{r},\hat{\Omega})+\hat{\Omega}\frac{S\psi(\vec{r},\hat{\Omega})+q}{\Sigma_t}\right)\right].\end{aligned}$$

Next, we use the following vector identity to expand this term,

$$f(\nabla \cdot g) = \nabla \cdot (fg) - g \cdot (\nabla f)\,,$$

resulting in two terms,

$$\int d\hat{\Omega} \int_{\mathcal{D}} d\vec{r} \Bigg[\nabla \cdot v(\vec{r}) \left(-\hat{\Omega}\frac{1}{\Sigma_t}\hat{\Omega} \cdot \nabla\psi(\vec{r},\hat{\Omega}) + \hat{\Omega}\frac{S\psi(\vec{r},\hat{\Omega}) + q}{\Sigma_t} \right) \\ + \nabla v(\vec{r}) \cdot \left(\hat{\Omega}\frac{1}{\Sigma_t}\hat{\Omega} \cdot \nabla\psi(\vec{r},\hat{\Omega}) - \hat{\Omega}\frac{S\psi(\vec{r},\hat{\Omega}) + q}{\Sigma_t} \right) \Bigg]\,.$$

We then apply Green's theorem to the first term,

$$\int \nabla \cdot \vec{f}\, d\vec{r} = \int_{\partial\mathcal{D}} \hat{n} \cdot \vec{f}\, ds\,,$$

where $\partial\mathcal{D}$ is the surface of the region $\mathcal{D}$, $\hat{n}$ is the surface normal, and ds is the differential distance along the surface. This leaves us with the two terms,

$$\int d\hat{\Omega} \int_{\partial\mathcal{D}} ds\ (\hat{n} \cdot v(\vec{r})) \left(-\hat{\Omega}\frac{1}{\Sigma_t}\hat{\Omega} \cdot \nabla\psi(\vec{r},\hat{\Omega}) + \hat{\Omega}\frac{S\psi(\vec{r},\hat{\Omega}) + q}{\Sigma_t} \right) \\ + \int d\hat{\Omega} \int_{\mathcal{D}} d\vec{r}\, \nabla v(\vec{r}) \cdot \left(\hat{\Omega}\frac{1}{\Sigma_t}\hat{\Omega} \cdot \nabla\psi(\vec{r},\hat{\Omega}) - \hat{\Omega}\frac{S\psi(\vec{r},\hat{\Omega}) + q}{\Sigma_t} \right)\,.$$

Factoring the $\hat{\Omega}$ out of the first term,

$$\int d\hat{\Omega} \int_{\partial\mathcal{D}} ds \left(\hat{n} \cdot \hat{\Omega}\right) v(\vec{r}) \left(-\frac{1}{\Sigma_t}\hat{\Omega} \cdot \nabla\psi(\vec{r},\hat{\Omega}) + \frac{S\psi(\vec{r},\hat{\Omega}) + q}{\Sigma_t} \right) \\ + \int d\hat{\Omega} \int_{\mathcal{D}} d\vec{r}\, \nabla v(\vec{r}) \cdot \left(\hat{\Omega}\frac{1}{\Sigma_t}\hat{\Omega} \cdot \nabla\psi(\vec{r},\hat{\Omega}) - \hat{\Omega}\frac{S\psi(\vec{r},\hat{\Omega}) + q}{\Sigma_t} \right)\,,$$

we can see that it is identical to the first-order form of the transport equation. That is,

$$-\frac{1}{\Sigma_t}\hat{\Omega} \cdot \nabla\psi(\vec{r},\hat{\Omega}) + \frac{S\psi(\vec{r},\hat{\Omega}) + q}{\Sigma_t}\bigg|_{\partial\mathcal{D}} = \psi(\vec{r}_b,\hat{\Omega}) \equiv \psi_b(\vec{r},\hat{\Omega}), \quad \forall \vec{r}_b \in \partial\mathcal{D}\,.$$

Substituting in this term leaves us with,

$$\int d\hat{\Omega} \int_{\partial\mathcal{D}} ds \left(\hat{n} \cdot \hat{\Omega}\right) v(\vec{r}) \psi_b(\vec{r},\hat{\Omega}) \\ + \int d\hat{\Omega} \int_{\mathcal{D}} d\vec{r}\, \nabla v(\vec{r}) \cdot \left(\hat{\Omega}\frac{1}{\Sigma_t}\hat{\Omega} \cdot \nabla\psi(\vec{r},\hat{\Omega}) - \hat{\Omega}\frac{S\psi(\vec{r},\hat{\Omega}) + q}{\Sigma_t} \right)\,.$$

We can use the commutative properties of the dot product again to obtain,

$$\begin{aligned}\int d\hat{\Omega}\int_{\partial\mathcal{D}} ds\left(\hat{n}\cdot\hat{\Omega}\right)v(\vec{r})\psi_b(\vec{r},\hat{\Omega})\\ +\int d\hat{\Omega}\int_{\mathcal{D}} d\vec{r}\left(\hat{\Omega}\cdot\nabla v(\vec{r})\right)\left(\frac{1}{\Sigma_t}\hat{\Omega}\cdot\nabla\psi(\vec{r},\hat{\Omega})-\frac{S\psi(\vec{r},\hat{\Omega})+q}{\Sigma_t}\right).\end{aligned}\tag{2.33}$$

We substitute the final expansion in Eq. (2.33) back into Eq. (2.32), replacing the first term and giving us the next step in the derivation of the weak form,

$$\begin{aligned}\int d\hat{\Omega}\int_{\partial\mathcal{D}} ds\left(\hat{n}\cdot\hat{\Omega}\right)v(\vec{r})\psi_b(\vec{r},\hat{\Omega})\\ +\int d\hat{\Omega}\int_{\mathcal{D}} d\vec{r}\left(\hat{\Omega}\cdot\nabla v(\vec{r})\right)\left(\frac{1}{\Sigma_t}\hat{\Omega}\cdot\nabla\psi(\vec{r},\hat{\Omega})-\frac{S\psi(\vec{r},\hat{\Omega})+q}{\Sigma_t}\right)\\ +\int d\hat{\Omega}\int_{\mathcal{D}} d\vec{r}\, v(\vec{r})\Sigma_t\psi(\vec{r},\hat{\Omega})-\int d\hat{\Omega}\int_{\mathcal{D}} d\vec{r}\, v(\vec{r})\left(S\psi(\vec{r},\hat{\Omega})+q\right)=0\,.\end{aligned}\tag{2.34}$$

As discussed in Sec. 2.3, we consider a triangulation of our spatial domain, divided into non-overlapping elements,

$$T_h=\{K_1,K_2,\dots K_M\}\quad\text{such that}\quad\bigcup_{K\in T_h}K=\mathcal{D}\,.$$

We will seek an approximate solution ψ_h in the following space:

$$V_h=\left\{v\in C^0(\mathcal{D})\,:\,v|_K\in\mathbb{P}_p(K)\,\forall K\in T_h\right\}\,.$$

We insert this approximate solution into the SAAF weak form Eq. (2.36),

$$\psi_h(\vec{r},\hat{\Omega})=\sum_{i=1}^{N}\psi_i(\hat{\Omega})\varphi(\vec{r}_i\,)\,,\tag{2.35}$$

and complete the Galerkin formulation by seeking $\psi_h\in V_h$ such that,

$$\begin{aligned}\int d\hat{\Omega}\int_{\partial\mathcal{D}} ds\left(\hat{n}\cdot\hat{\Omega}\right)v_h(\vec{r})\psi_b(\vec{r},\hat{\Omega})\\ +\int d\hat{\Omega}\int_{\mathcal{D}} d\vec{r}\left(\hat{\Omega}\cdot\nabla v_h(\vec{r})\right)\left(\frac{1}{\Sigma_t}\hat{\Omega}\cdot\nabla\psi_h(\vec{r},\hat{\Omega})-\frac{S\psi_h(\vec{r},\hat{\Omega})+q}{\Sigma_t}\right)\\ +\int d\hat{\Omega}\int_{\mathcal{D}} d\vec{r}\, v_h(\vec{r})\Sigma_t\psi_h(\vec{r},\hat{\Omega})-\int d\hat{\Omega}\int_{\mathcal{D}} d\vec{r}\, v_h(\vec{r})\left(S\psi_h(\vec{r},\hat{\Omega})+q\right)=0\,,\end{aligned}\tag{2.36}$$

for all $v_h \in V_h$. Included in the possible values of v_h is the most straightforward choice: that the weighing functions are equal to the basis functions, $v_h(\vec{r}) = \varphi_j(\vec{r})$, for $j = 1, \ldots, N$. Plugging this into Eq. (2.36) gives us N equations of the form,

$$\begin{aligned}\int d\hat{\Omega} \int_{\partial\mathcal{D}} ds \left(\hat{n}\cdot\hat{\Omega}\right) \varphi_j(\vec{r})\psi_b(\vec{r},\hat{\Omega}) \\ + \int d\hat{\Omega}\int_{\mathcal{D}} d\vec{r}\left(\hat{\Omega}\cdot\nabla\varphi_j(\vec{r})\right)\left(\frac{1}{\Sigma_t}\hat{\Omega}\cdot\nabla\psi_h(\vec{r},\hat{\Omega}) - \frac{S\psi_h(\vec{r},\hat{\Omega})+q}{\Sigma_t}\right) \\ + \int d\hat{\Omega}\int_{\mathcal{D}} d\vec{r}\,\varphi_j(\vec{r})\Sigma_t\psi_h(\vec{r},\hat{\Omega}) - \int d\hat{\Omega}\int_{\mathcal{D}} d\vec{r}\,\varphi_j(\vec{r})\left(S\psi_h(\vec{r},\hat{\Omega})+q\right) = 0\ ,\end{aligned}$$

for $j = 1, \ldots N$.

The next step is to substitute in our approximate solution from Eq. (2.35). We will not substitute this in for the scattering terms, $S\psi$, because our solving methodology will treat these as constants that are not dependent on the angular flux that we are solving for. In practice, our iterative methods will calculate these terms from the previous iteration. In addition, we must split the boundary term into expressions for incoming and out-going fluxes. The incoming flux will be defined by our choice of boundary conditions, and the outgoing flux will be equal to the angular flux itself,

$$\psi_b(\vec{r},\hat{\Omega}) = \begin{cases}\psi_{\text{inc}}(\vec{r},\hat{\Omega}) & \hat{n}\cdot\hat{\Omega} < 0 \\ \psi_h(\vec{r},\hat{\Omega}) & \hat{n}\cdot\hat{\Omega} > 0\end{cases}\ . \tag{2.37}$$

Splitting the boundary term gives us,

$$\begin{aligned}&\int_{\left(\hat{n}\cdot\hat{\Omega}\right)<0} d\hat{\Omega}\int_{\partial\mathcal{D}} ds\left(\hat{n}\cdot\hat{\Omega}\right)\varphi_j(\vec{r})\psi_{\text{inc}}(\vec{r},\hat{\Omega}) + \int_{\left(\hat{n}\cdot\hat{\Omega}\right)>0} d\hat{\Omega}\int_{\partial\mathcal{D}} ds\left(\hat{n}\cdot\hat{\Omega}\right)\varphi_j(\vec{r})\psi_h(\vec{r},\hat{\Omega}) \\ &+ \int d\hat{\Omega}\int_{\mathcal{D}} d\vec{r}\left(\hat{\Omega}\cdot\nabla\varphi_j(\vec{r})\right)\left(\frac{1}{\Sigma_t}\hat{\Omega}\cdot\nabla\psi_h(\vec{r},\hat{\Omega}) - \frac{S\psi_h(\vec{r},\hat{\Omega})+q}{\Sigma_t}\right) \\ &+ \int d\hat{\Omega}\int_{\mathcal{D}} d\vec{r}\,\varphi_j(\vec{r})\Sigma_t\psi_h(\vec{r},\hat{\Omega}) - \int d\hat{\Omega}\int_{\mathcal{D}} d\vec{r}\,\varphi_j(\vec{r})\left(S\psi_h(\vec{r},\hat{\Omega})+q\right) = 0, \quad j = 1,\ldots N\ .\end{aligned}$$

Or, with the approximate solution substituted in, and the summation factored out from each term,

$$\begin{aligned}\sum_{i=1}^{N}\Bigg[&\int_{\left(\hat{n}\cdot\hat{\Omega}\right)<0} d\hat{\Omega}\int_{\partial\mathcal{D}} ds\left(\hat{n}\cdot\hat{\Omega}\right)\varphi_j(\vec{r})\psi_{\text{inc}}(\vec{r},\hat{\Omega}) + \int_{\left(\hat{n}\cdot\hat{\Omega}\right)>0} d\hat{\Omega}\int_{\partial\mathcal{D}} ds\left(\hat{n}\cdot\hat{\Omega}\right)\varphi_j(\vec{r})\varphi_i(\vec{r})\psi_i(\hat{\Omega}) \\ &+ \int d\hat{\Omega}\int_{\mathcal{D}} d\vec{r}\left(\hat{\Omega}\cdot\nabla\varphi_j(\vec{r})\right)\left(\frac{1}{\Sigma_t}\hat{\Omega}\cdot\nabla\varphi_i(\vec{r})\psi_i(\hat{\Omega}) - \frac{S\psi_h(\vec{r},\hat{\Omega})+q}{\Sigma_t}\right) \\ &+ \int d\hat{\Omega}\int_{\mathcal{D}} d\vec{r}\,\varphi_j(\vec{r})\Sigma_t\varphi_i(\vec{r})\psi_i(\hat{\Omega}) - \int d\hat{\Omega}\int_{\mathcal{D}} d\vec{r}\,\varphi_j(\vec{r})\left(S\psi_h(\vec{r},\hat{\Omega})+q\right)\Bigg] = 0\,,\end{aligned}$$

for $i, j = 1, \dots N$.

We therefore have the weak form of the SAAF equation that forms a system of the form,

$$\mathbf{A}\vec{\Psi} = \vec{b}$$

where $\mathbf{A} \in \mathbb{R}^{N\times N}$, $\vec{b} \in \mathbb{R}^N$, and $\vec{\Psi} = \left[\vec{\psi}_0(\hat{\Omega}), \dots, \vec{\psi}_N(\hat{\Omega})\right]^{\mathrm{T}}$. The entries of the left-hand side matrix are,

$$\begin{aligned}\mathbf{A}\vec{\Psi}(i,j) = &\int_{\left(\hat{n}\cdot\hat{\Omega}\right)>0} d\hat{\Omega} \int_{\partial\mathcal{D}} ds \left(\hat{n}\cdot\hat{\Omega}\right) \varphi_j(\vec{r})\varphi_i(\vec{r})\psi_i(\hat{\Omega}) \\ &+ \int d\hat{\Omega} \int_{\mathcal{D}} d\vec{r} \left(\hat{\Omega}\cdot\nabla\varphi_j(\vec{r})\right) \left(\frac{1}{\Sigma_t}\hat{\Omega}\cdot\nabla\varphi_i(\vec{r})\psi_i(\hat{\Omega})\right) \\ &+ \int d\hat{\Omega} \int_{\mathcal{D}} d\vec{r}\, \varphi_j(\vec{r})\Sigma_t\varphi_i(\vec{r})\psi_i(\hat{\Omega})\,,\end{aligned}$$

and the right-hand side vector has entries,

$$\begin{aligned}\vec{b}(i) = &-\int_{\left(\hat{n}\cdot\hat{\Omega}\right)<0} d\hat{\Omega} \int_{\partial\mathcal{D}} ds \left(\hat{n}\cdot\hat{\Omega}\right) \varphi_i(\vec{r})\psi_{\text{inc}}(\vec{r},\hat{\Omega}) \\ &+ \int d\hat{\Omega} \int_{\mathcal{D}} d\vec{r} \left(\hat{\Omega}\cdot\nabla\varphi_i(\vec{r})\right) \left(\frac{S\psi_h(\vec{r},\hat{\Omega}) + q}{\Sigma_t}\right) \\ &+ \int d\hat{\Omega} \int_{\mathcal{D}} d\vec{r}\, \varphi_i(\vec{r}) \left(S\psi_h(\vec{r},\hat{\Omega}) + q\right)\,.\end{aligned}$$

We now have the SAAF equation in a form that can be assembled and solved. Armed with this derivation and applying the multigroup and discrete ordinates method, we can use iterative solving techniques. We will explore these in the following chapter.

2.6 Conclusion

In this chapter, we covered the Boltzmann neutron transport equation that describes the propagation of neutrons in our steady state system. We then introduced the k-eigenvalue formulation that describes the criticality of this system. To make the equation practical to solve, we described how to discretize this the phase space, including the development and the use of continuous finite element methods. Finally, we discussed two second-order formulations of the equation, the diffusion equation and the SAAF equation. We derived the weak forms of these formulations to support using finite element methods. In the following chapter, we will build on this discretization and formulation by introducing iterative methods to solve the transport equation. These methods will introduce their own challenges, which we will discuss and address with acceleration methods.

Chapter 3

Iterative Methods and Acceleration Schemes

The size and complexity of the discretized transport equation in the previous chapter prevents solving directly in most cases. Instead, we will apply iterative methods[11], a broad class of schemes that converge on the solution in a finite number of discrete steps. There are three types of iterative schemes that we will use in solving the transport equation. The first two go together to solve the space, energy, and angle part of the equations–we will call these fixed source solves as a fission source, if present, is treated as fixed. The third is an eigenvalue solver that is only necessary when looking for the criticality state of the systems. We will discuss the iterative methods we use. In this work we will only focus on two of the iterative schemes, the one for solving in energy and the one for solving the eigenvalue. Thus, we will discuss the convergence properties of these methods and two acceleration schemes designed to alleviate some of the convergence challenges.

3.1 Operator Form

We will apply the iterative methods in this chapter to the fully discretized transport equation. To simplify notation, we will summarize the discretization using the operator form, in which the parts of the discretized transport equation are expressed as matrices and vectors. Our discretizations and important parameters are summarized as,

- Energy discretized using multigroup discretization with G energy groups,
- Angle discretized using discrete ordinates using a quadrature set with N total angles,
- Space discretized using a continuous finite element method using a triangulation with M total degrees of freedom, and
- Scattering expansion using scattering operator expanded in L moments.

The full operator form of the discretized transport equation is

$$\mathbf{H}\vec{\psi} = \mathbf{M}\mathbf{S}\vec{\phi} + \frac{1}{k}\mathbf{M}\mathbf{F}\vec{\phi} \,, \tag{3.1}$$

where the terms are described below.

The matrix $\mathbf{H}$ is the discretized transport operator. It is block-diagonal, made up of $G \times N$ blocks, one for each energy group and angle. Each of these blocks is $M \times M$, representing the spatial discretization. The ordering within the matrix may vary, but the size of the matrix $\mathbf{H} \in \mathbb{R}^{m\times m}$ where $m = (G \times N \times M)$ will not. The block vector $\vec{\psi} \in \mathbb{R}^{m\times 1}$ contains the spatial solution for each angle and group. The scalar flux vector contains the angular flux moments (not the solution) along angular collocation points, and the size is therefore a function of L and not N: $\vec{\phi} \in \mathbb{R}^{n\times 1}$ where $n = (G \times L \times M)$. The scattering and fission operators operate on these moments: $\mathbf{S} \in \mathbb{R}^{n\times n}$, $\mathbf{F} \in \mathbb{R}^{n\times n}$. Finally, the matrix $\mathbf{M}$ converts angular flux moments into angular fluxes along the collocation angles of our discretization. This is called the moment-to-discrete operator and is of size $\mathbf{M} \in \mathbb{R}^{m\times n}$.

Related is the means by which we calculate the scalar flux using the angular flux moments,

$$\vec{\phi} = \mathbf{D}\vec{\psi} \,,$$

where $\mathbf{D} \in \mathbb{R}^{n\times m}$ is the discrete to moment operator. We can substitute this into Eq. (3.1) to see that the angular flux explicitly appears on both sides of the discretized equation,

$$\mathbf{H}\vec{\psi} = \mathbf{M}\mathbf{S}\mathbf{D}\vec{\psi} + \frac{1}{k}\mathbf{M}\mathbf{F}\mathbf{D}\vec{\psi} \,.$$

When describing the iterative schemes below, it is often beneficial to simplify the notation,

$$\mathbf{H}\vec{\psi} = \widetilde{\mathbf{S}}\vec{\psi} + \frac{1}{k}\widetilde{\mathbf{F}}\vec{\psi} \,, \tag{3.2}$$

where $\widetilde{\mathbf{X}} = \mathbf{M}\mathbf{X}\mathbf{D} \in \mathbb{R}^{m\times m}$ for a matrix $\mathbf{X} \in \mathbb{R}^{n\times n}$.

3.2 Solving Linear Equations

A general iterative method solves $\mathbf{A}\vec{x} = \vec{b}$ by iteratively performing

$$\vec{x}^{(k+1)} = \vec{x}^{(k)} + \mathbf{P}^{-1}\left(\vec{b} - \mathbf{A}\vec{x}^{(k)}\right) \,, \tag{3.3}$$

where $k \geq 0$ is an iteration index, and $\mathbf{P}$ is a preconditioner matrix chosen such that $\mathbf{P}^{-1}\mathbf{A} \approx \mathbf{I}$. Note that if $\mathbf{P}^{-1}$ is chosen to be exactly $\mathbf{A}^{-1}$, Eq. (3.3) reduces to solving the original equation. Choice of $\mathbf{P}$ defines the type of method,

- GS iteration chooses $\mathbf{P}$ equal to the lower-triangular portion of $\mathbf{A}$,

- Jacobi iteration chooses $\mathbf{P}$ equal to the diagonal portion of $\mathbf{A}$, and
- Richardson iteration chooses $\mathbf{P} = \frac{1}{\omega}\mathbf{I}$, where ω is a parameter chosen such that the iteration converges.

Methods that choose $\mathbf{P}$ as part of $\mathbf{A}$ are called matrix-splitting methods.

The error in step k of the method is defined,

$$e^{(k)} \equiv \vec{x} - \vec{x}^{(k)} = \mathbf{A}^{-1}\vec{b} - \vec{x}^{(k)} \ , \tag{3.4}$$

and is related to the error in the previous step using the iteration matrix $\mathbf{C}$,

$$e^{(k)} = \mathbf{C}e^{(k-1)} = \left(\mathbf{I} - \mathbf{P}^{-1}\mathbf{A}\right) e^{(k-1)} = \left(\mathbf{I} - \mathbf{P}^{-1}\mathbf{A}\right)^k e^{(0)} \ . \tag{3.5}$$

Convergence of the iterative method requires the error to go to zero as the number of iterations approaches infinity. Therefore, this is dependent on the norm of the iteration matrix $\mathbf{C}$. Taking the norm of both sides of Eq. (3.5), taking the limit, and using the Cauchy-Schwartz inequality we can say

$$\lim_{k\to\infty} \left\| e^{(k)} \right\| = \lim_{k\to\infty} \left\| \mathbf{C}^k e^{(0)} \right\| \leq \lim_{k\to\infty} \left\| \mathbf{C^k} \right\| \left\| e^{(0)} \right\| = \rho(\mathbf{C}) \left\| e^{(0)} \right\| \ ,$$

where $\rho(\mathbf{C})$ is the spectra radius of $\mathbf{C}$. The problem converges if and only if the spectral radius of $\mathbf{C}$ is less than unity.

3.2.1 Solving the Discretized Neutron Transport Equation

The discretized form of the Boltzmann transport equation as seen, e.g., in Eq. (2.29) is a set of single-group energy equations that are each only a function of space and angle. Each "within-group" equation is solved for that group's flux with an iterative method; these are often called the inner iterations. In this work, we use source iteration (SI) as the within-group solver. These single-group equations are often coupled together through scattering. We typically use a different iterative solver over energy groups, and these are the "multigroup" solver or the outer iterations. The outer iterations converge the angular flux over all groups. If there is no upscattering, only one outer iteration is required. We use GS as the multigroup solver.

The combination of inner and outer iterations is sufficient to solve fixed source problems. In systems with fission, the fission source is treated as a constant during the inner and outer iterations so it looks like a fixed source problem. The process of converging the fission source and the k-eigenvalue is the eigenvalue iteration. In this work, we use power-iteration (PI) for the eigenvalue solver. We therefore have three iterative schemes nested and working together.

3.3 Fixed Source Solvers

The first method for solving part of the transport equation is the well-known SI. This is a straightforward scheme for solving the scattering term of the transport equation using a stationary iterative method to solve the linear equation. In doing so, we will hold the fission source constant, leaving it to other methods to converge this term. A complete and thorough discussion of SI can be found in many texts and review papers [2, 12].

We begin with the discretized transport equation in operator form, Eq. (3.2), with the fission source term replaced with a constant source term vector and the terms consolidated,

$$\left(\mathbf{H} - \widetilde{\mathbf{S}}\right)\vec{\psi} = \vec{Q} \, . \tag{3.6}$$

We now apply the iterative scheme of Eq. (3.3), introducing the iteration index (k),

$$\vec{\psi}^{(k+1)} = \vec{\psi}^{(k)} + \mathbf{P}^{-1}\left[\vec{Q} - (\mathbf{H} - \widetilde{\mathbf{S}})\vec{\psi}^{(k)}\right] \, . \tag{3.7}$$

The source iteration scheme is a matrix-splitting scheme, where we chose the preconditioner matrix $\mathbf{P} = \mathbf{H}$. The source iteration scheme reduces as,

$$\begin{aligned}
\vec{\psi}^{(k+1)} &= \vec{\psi}^{(k)} + \mathbf{P}^{-1}\left[\vec{Q} - (\mathbf{H} - \widetilde{\mathbf{S}})\vec{\psi}^{(k)}\right] \\
&= \vec{\psi}^{(k)} + \mathbf{H}^{-1}\left[\vec{Q} - (\mathbf{H} - \widetilde{\mathbf{S}})\vec{\psi}^{(k)}\right] \\
\mathbf{H}\vec{\psi}^{(k+1)} &= \mathbf{H}\vec{\psi}^{(k)} + \vec{Q} - (\mathbf{H} - \widetilde{\mathbf{S}})\vec{\psi}^{(k)} \\
&= \widetilde{\mathbf{S}}\vec{\psi}^{(k)} + \vec{Q} \, .
\end{aligned}$$

Using the source-iteration scheme, an initial guess for the scalar flux is chosen and used to calculate $\vec{\psi}^{(1)}$. This new value is then used to calculate the new scalar flux, $\vec{\phi}^{(1)}$ and the process is repeated. Repeated application of the scattering operator k times results in an angular flux that accounts for particles that have scattered at most $k-1$ times. Intuitively, this indicates that the iterative scheme will take many steps for problems with a large amount of scattering.

To solve the fully discretized neutron transport equation, we will solve groups individually in angle and space using the SI scheme. Once converged, we move on to the next group, using the flux from the previous group, if needed, in the scattering source term. This forward substitution is the GS iterative process. The convergence of the individual groups using SI is referred to as the inner iteration, and the GS iteration is referred to as the outer iteration. This process requires separating the problem into G equations of the form

$$\mathbf{H}_g\vec{\psi}_g^{(k+1)} = \mathbf{M}\sum_{g'=0}^{g}\mathbf{S}_{g'\to g}\vec{\phi}_{g'}^{(k+1)} + \mathbf{M}\sum_{g'=g+1}^{G-1}\mathbf{S}_{g'\to g}\vec{\phi}_{g'}^{(k)} + \vec{Q}_g \, . \tag{3.8}$$

The GS scheme is most efficient with minimal contribution from lower energy groups (higher g) to higher energy groups (lower g). This phenomenon, upscattering, requires more outer iterations to loop over those groups with upscattering contributions until all groups have converged. In this work we focus on speeding up the the energy part of the fixed source solves, though BART could easily be used to focus on accelerating the space-angle solves as well.

3.3.1 Convergence

The speed at which GS converges depends on the spectral radius of the iteration matrix. Fourier analysis of the infinite one-dimensional case [12, 11] reveals that the spectral radius is equal to the scattering ratio $c = \Sigma_s/\Sigma_t$. As a system becomes more and more dominated by scattering, the value of c approaches unity, and GS may take an arbitrarily long amount of time to converge. This matches with the physical intuition identified earlier: if each iteration results in a solution that takes into account neutrons that scatter an additional time, an arbitrary number of iterations may be required if the neutrons will scatter an arbitrary number of times.

This convergence property is not ideal for many problems of interest. Thermal reactor designs rely on high energy neutrons born from fission losing nearly all of their energy by scattering many times before being ultimately absorbed. These designs are built around causing neutrons to scatter without leaking or being absorbed, ideal circumstances for source iteration to converge extremely slowly. Additionally, some shielding problems may rely on a large amount of scattering, causing issues for source iteration. Finally, as discussed, significant upscattering can cause the GS iteration schemes to converge very slowly. These convergence issues have motivated the development of acceleration schemes to produce a more efficient iteration strategy.

3.3.2 Accelerating Fixed Source Solves

Approaches to accelerate SI and GS stretch back as far as the work of Kopp and Lebedev in the 1960s [13, 14]. Their schemes chose better preconditioners for source iteration that would improve the convergence properties for problems with a large amount of scattering. These methods are referenced as "synthetic methods," in that they do not solve the original problem (like a matrix splitting preconditioner would) but a different problem that is a good approximation. One such method that has wide applicability and use is the diffusion synthetic acceleration (DSA) scheme [15]. The diffusion synthetic acceleration (DSA) scheme uses a diffusion approximation to calculate a more accurate error term in Eq. (3.7). This is very successful in problems with a large amount of scattering where diffusion is a better description of the behavior of neutrons. Careful consideration must be paid to the discretization of the diffusion equation when using DSA to ensure consistency with the discretization of the transport equation that it is accelerating. Inconsistencies can cause instabilities in the scheme, ultimately preventing convergence. Other acceleration methods have been developed, including the nonlinear diffusion acceleration (NDA) [16, 8] and two-grid (TG) methods, discussed in the next sections.

3.4 Two-Grid Acceleration

As discussed in the previous section, the two-grid (TG) acceleration scheme was developed by Adams and Morel [17] to speed up the convergence of the GS iteration scheme. The method is similar to other schemes for efficient solving of PDEs that use multiple levels of discretization, called multigrid methods. These schemes use a fast and efficient course-grid solve of the PDE to provide a correction to the finer grid solve, sometimes using multiple grids of increasing coarseness and correcting on each level. This speeds up the convergence of smoother error modes that slow down the solve on the fine grid. For the TG method, the two grids are not spatial but angular: the fine grid is the standard angular solve and the course grid is a scalar diffusion solve. Specifically, the TG method was derived to accelerate the GS iteration in regions with a large amount of upscattering.

3.4.1 Derivation

Our derivation of the TG method begins with the exact operator form of the transport equation but for a single group g,

$$\mathbf{H}_g\vec{\psi}_g = \mathbf{M}\sum_{g'=0}^{G-1}\mathbf{S}_{g'\to g}\vec{\phi} + \vec{Q}_g \ . \tag{3.9}$$

and the GS iteration scheme,

$$\mathbf{H}_g\vec{\psi}_g^{(k+1)} = \mathbf{M}\sum_{g'=0}^{g}\mathbf{S}_{g'\to g}\vec{\phi}_{g'}^{(k+1)} + \mathbf{M}\sum_{g'=g+1}^{G-1}\mathbf{S}_{g'\to g}\vec{\phi}_{g'}^{(k)} + \vec{Q}_g \ . \tag{3.8, revisited}$$

Next, we subtract the iterative scheme Eq. (3.8) from the exact equation Eq. (3.9),

$$\mathbf{H}_g\mathbf{D}^{-1}\vec{\varepsilon}_g^{(k+1)} = \mathbf{M}\sum_{g'=0}^{g}\mathbf{S}_{g'\to g}\vec{\varepsilon}_{g'}^{(k+1)} + \mathbf{M}\sum_{g'=g+1}^{G-1}\mathbf{S}_{g'\to g}\vec{\varepsilon}_{g'}^{(k)} \ ,$$

where,

$$\vec{\varepsilon}_g^{(k)} = \vec{\phi}_g - \vec{\phi}_g^{(k)} \ , \tag{3.10}$$

is the scalar flux error in step k. Next, we will add and subtract an extra term to the right hand side,

$$\begin{aligned}\mathbf{H}_g\mathbf{D}^{-1}\vec{\varepsilon}_g^{(k+1)} = {} & \mathbf{M}\sum_{g'=0}^{g}\mathbf{S}_{g'\to g}\vec{\varepsilon}_{g'}^{(k+1)} + \mathbf{M}\sum_{g'=g+1}^{G-1}\mathbf{S}_{g'\to g}\vec{\varepsilon}_{g'}^{(k)} \\ & + \mathbf{M}\sum_{g'=g+1}^{G-1}\mathbf{S}_{g'\to g}\vec{\varepsilon}_{g'}^{(k+1)} - \mathbf{M}\sum_{g'=g+1}^{G-1}\mathbf{S}_{g'\to g}\vec{\varepsilon}_{g'}^{(k+1)} \ ,\end{aligned}$$

which will combine with our current summations,

$$\mathbf{H}_g\mathbf{D}^{-1}\vec{\varepsilon}_g^{(k+1)} = \mathbf{M}\sum_{g'=0}^{G-1}\mathbf{S}_{g'\to g}\vec{\varepsilon}_{g'}^{(k+1)} + \mathbf{M}\sum_{g'=g+1}^{G-1}\mathbf{S}_{g'\to g}\left(\vec{\varepsilon}_{g'}^{(k)} - \vec{\varepsilon}_{g'}^{(k+1)}\right) .$$

Using the definition of our error, Eq. (3.10),

$$\begin{aligned}\mathbf{H}_g\mathbf{D}^{-1}\vec{\varepsilon}_g^{(k+1)} &= \mathbf{M}\sum_{g'=0}^{G-1}\mathbf{S}_{g'\to g}\vec{\varepsilon}_{g'}^{(k+1)} + \mathbf{M}\sum_{g'=g+1}^{G-1}\mathbf{S}_{g'\to g}\left(\vec{\phi}_{g'}^{(k+1)} - \vec{\phi}_{g'}^{(k)}\right)\\ &= \mathbf{M}\sum_{g'=0}^{G-1}\mathbf{S}_{g'\to g}\vec{\varepsilon}_{g'}^{(k+1)} + \mathbf{M}\vec{R}_g^{(k+1)} ,\end{aligned}$$

where $\vec{R}_g^{(k+1)}$ is the isotropic residual.

Our first restriction is to reduce the angular resolution by only considering the scalar flux moments and approximating the transport operator using the diffusion operator, weighed by a diffusion coefficient. This is a natural step, given that our error and isotropic residuals are defined using only the scalar flux and gives

$$\left[-\nabla\cdot D_g\nabla + \mathbf{\Sigma}_t^g\right]\vec{\varepsilon}_g^{(k+1)} = \sum_{g'=0}^{G-1}\mathbf{\Sigma}_{s0}^{g'\to g}\vec{\varepsilon}_{g'}^{(k+1)} + \vec{R}_g^{(k+1)} , \tag{3.11}$$

where $\nabla\cdot D_g\nabla$ is the discretized diffusion operator and $\mathbf{\Sigma}_{s0}$ is the discretized zeroth moments of the scattering operator.

Our second restriction step is to collapse all energy groups into a single group. To define our restriction operator, we will assume that our error can be separated into functions of energy and space,

$$\vec{\varepsilon}_g^{(k)} = \vec{\varepsilon}^{(k)}\xi_g , \tag{3.12}$$

where $\vec{\varepsilon}$ is the spatial component of the error, and ξ_g is an energy-shape weighing function that serves as a prolongation operator. The norm of the total error should be conserved, so the values of ξ_g are normalized such that their sum is unity,

$$\sum_{g=0}^{G-1}\xi_g \equiv 1 .$$

The value of $\vec{\xi} \in \mathbb{R}^{G\times 1}$ is determined using an eigenvalue equation of the SI GS iteration matrix:

$$\left[\mathbf{\Sigma}_t - \mathbf{\Sigma}_{s0}^L - \mathbf{\Sigma}_{s0}^D\right]^{-1}\mathbf{\Sigma}_{s0}^U\vec{\xi} = \lambda\vec{\xi} ,$$

where $\mathbf{\Sigma}_t \in \mathbb{R}^{G\times G}$ is a diagonal matrix with the total cross-sections and $\mathbf{\Sigma}_{s0} \in \mathbb{R}^{G\times G}$ is the zeroth-moments of the scattering cross-sections divided into strictly upper (U), diagonal (D), and strictly lower (L) matrices. The values of ξ_g correspond to the entries of the normalized eigenvector $\vec{\xi}$ for the spectral radius of this problem.

We now apply a restriction operator to Eq. (3.11) by first substituting in Eq. (3.12) and then summing over all groups,

$$\left[-\nabla \cdot \langle D \rangle \nabla + \left\langle \dot{D} \right\rangle + \langle \Sigma_a \rangle\right] \vec{\varepsilon}^{(k+1)} = \left\langle \vec{R}^{(k+1)} \right\rangle \ , \tag{3.13}$$

where

$$\begin{aligned}
\langle D \rangle &= \sum_{g=0}^{G-1} D_g \xi_g \\
\left\langle \dot{D} \right\rangle &= \sum_{g=0}^{G-1} D_g \nabla \xi_g \\
\langle \Sigma_a \rangle &= \sum_{g=0}^{G-1} \left[\Sigma_t^g \xi_g - \sum_{g'=0}^{G-1} \Sigma_{s0}^{g' \to g} \xi_{g'} \right] \\
\left\langle \vec{R}^{(k+1)} \right\rangle &= \sum_{g=0}^{G-1} R_g^{(k+1)} \ .
\end{aligned}$$

For our discretization, we will assume that all cells have a single material and so we will neglect the term that includes the gradient of the energy-shape weighing function. This is the final level of restriction, and this is the equation that will be solved on the same spatial domain as the original problem but with greatly reduced angular and energy phase space. The formulation of the residual term and the calculation of the energy-shape weighting function are both based on groups that have upscattering. Therefore, it is expected that this method will speed up the convergence of our GS iterations in the presence of upscattering.

The full algorithm is now presented and shown in Fig. 3.1:

1. Solve the discretized GS,

$$\mathbf{H}_g \vec{\psi}_g^{(k+1/2)} = \mathbf{M} \sum_{g'=0}^{g} \mathbf{S}_{g' \to g} \vec{\phi}_{g'}^{(k+1/2)} + \mathbf{M} \sum_{g'=g+1}^{G-1} \mathbf{S}_{g' \to g} \vec{\phi}_{g'}^{(k)} + \vec{Q}_g \ ,$$

2. Calculate the collapsed one group isotropic residual,

$$\left\langle \vec{R}^{(k+1/2)} \right\rangle = \sum_{g=0}^{G-1} \sum_{g'=g+1}^{G-1} \Sigma_{s0}^{g' \to g} \left(\vec{\phi}_{g'}^{(k+1/2)} - \vec{\phi}_{g'}^{(k)} \right) \ ,$$

3. Solve the discretized restricted diffusion approximation for the error,

$$\left[-\nabla \cdot \langle D \rangle \nabla + \langle \Sigma_a \rangle\right] \vec{\varepsilon}^{(k+1/2)} = \left\langle \vec{R}^{(k+1/2)} \right\rangle \ ,$$

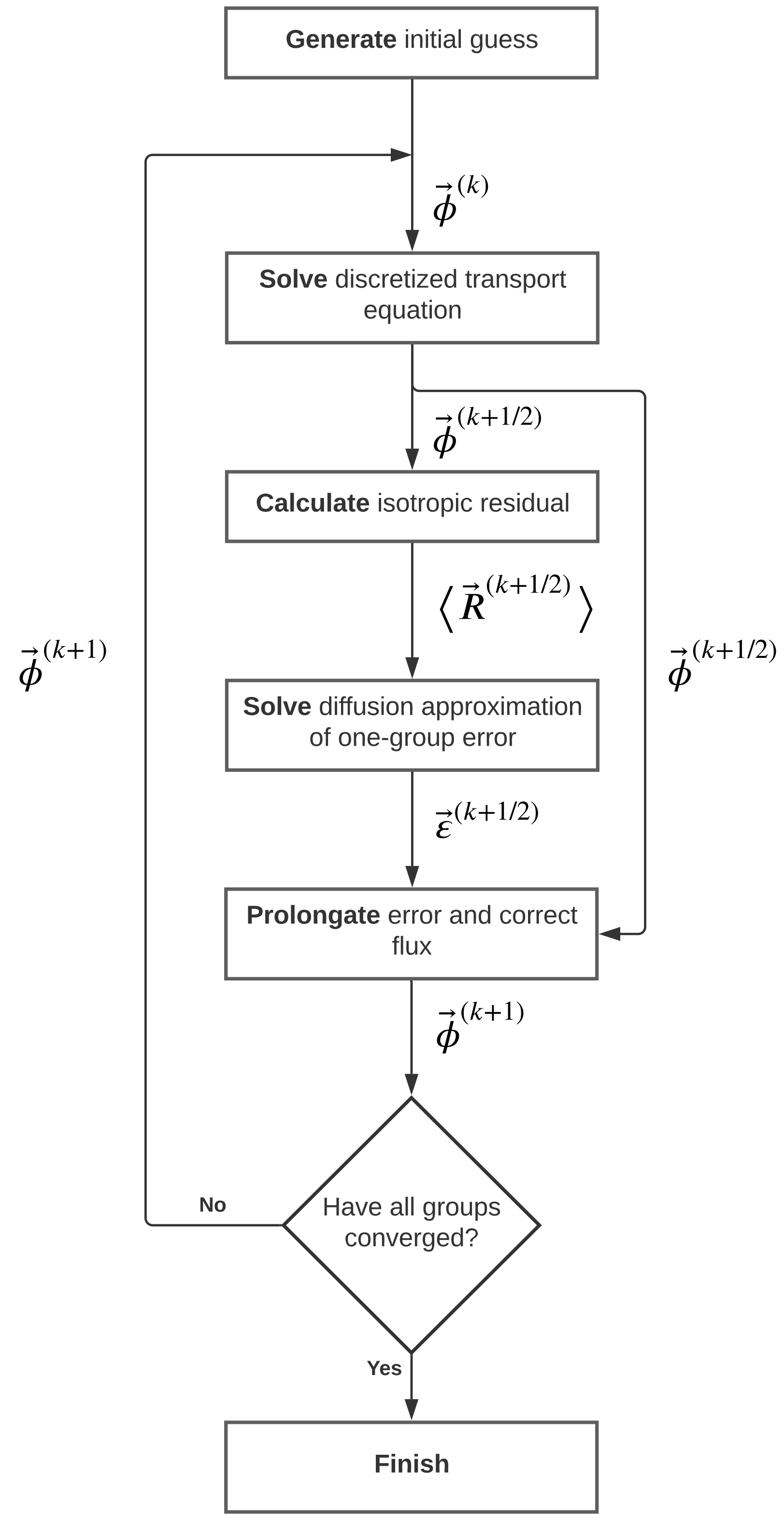

Generate initial guess
$\vec{\phi}^{(k)}$
Solve discretized transport equation
$\vec{\phi}^{(k+1/2)}$
Calculate isotropic residual
$\left\langle \vec{R}^{(k+1/2)} \right\rangle$
Solve diffusion approximation of one-group error
$\vec{\varepsilon}^{(k+1/2)}$
Prolongate error and correct flux
$\vec{\phi}^{(k+1)}$
Have all groups converged?
No
Yes
Finish
$\vec{\phi}^{(k+1)}$
$\vec{\phi}^{(k+1/2)}$

Figure 3.1: Two-grid algorithm.

4. Prolong the error as a correction using the energy-shape function and Eqs. (3.10) and (3.12)
$$\vec{\phi}_g^{(k+1)} = \vec{\phi}_g^{(k+1/2)} + \xi_g \vec{\varepsilon}^{\,(k+1/2)} \, .$$

The TG method has been show to speed up GS iteration in the presence of upscattering [17]. The acceleration depends on a very careful discretization of the restricted problem. This is a consequence of inconsistencies between the diffusion operator and the transport operator. To combat this, a version of two-grid that uses the transport operator, transport two-grid, was developed by Evans *et al.* [18]. For the purposes of this work, we will be using the TG algorithm to accelerate a diffusion-based formulation, so the transport two-grid method will not be required.

The weak formulation of the TG equation is nearly identical to the diffusion weak form, with the cross-sections weighed by the energy-shape function, and the right-hand-side replaced by the isotropic residual.

$$\left(\nabla\vec{v}, \left\langle D\right\rangle \nabla\vec{\varepsilon}^{(k+1/2)}\right)_{\mathbb{D}} + \left\langle \vec{v}, \frac{1}{2}\vec{\varepsilon}^{(k+1/2)}\right\rangle_{\partial\mathbb{D},\text{vacuum}} + \left(\vec{v}, \left\langle \Sigma_a\right\rangle \vec{\varepsilon}^{(k+1/2)}\right)_{\mathbb{D}} = \left(\vec{v}, \left\langle \vec{R}^{(k+1/2)}\right\rangle\right)_{\mathbb{D}} \, .$$

3.5 Eigenvalue Solver

The GS scheme will converge the scattering term of the discretized transport equation, but treats the fission term as a constant. To converge the fission eigenvalue, we will use the process of power iteration. This iterative method will converge an eigenvalue problem of the form,
$$\mathbf{A}\vec{x} = \lambda\vec{x} \, .$$
We assume that the matrix $\mathbf{A} \in \mathbb{R}^{m\times m}$ has m real eigenvalues $\lambda_0, \lambda_1, \ldots, \lambda_m$ that correspond to m eigenvectors $\vec{q}_0, \vec{q}_1, \ldots \vec{q}_m$. The eigenvalues are distinct and ordered such that $|\lambda_0| > |\lambda_1| > \ldots, |\lambda_m|$. The power iteration algorithm is as follows:

1. Begin with an initial guess $\vec{x}^{(0)}$,

2. Apply the matrix $\mathbf{A}$, $\vec{y}^{(k)} = \mathbf{A}\vec{x}^{(k-1)}$,

3. Normalize the vector to get the next vector $\vec{x}^{(k)} = \vec{y}^{(k)} / \left\|\vec{y}^{(k)}\right\|$,

4. Calculate the eigenvalue using the Rayleigh quotient,
$$\lambda^{(k)} = \left[\vec{x}^{(k)}\right]^T \mathbf{A}\vec{x}^{(k)} \, ,$$
and then,

5. Repeat the process of steps 2–4 until a convergence criterion is met, usually

$$|\lambda^{(k)} - \lambda^{(k-1)}|/\lambda^{(k-1)} < \epsilon \ ,$$

where ϵ is some convergence criterion.

The Rayleigh quotient in step 4 of the algorithm is an efficient means of calculating the eigenvalue. For a given guess $\vec{x}^{(k)}$, this finds $\lambda^{(k)}$ that minimizes

$$\left\|\mathbf{A}\vec{x}^{(k)} - \lambda^{(k)}\vec{x}^{(k)}\right\|_2 \ .$$

3.5.1 Convergence

To see the convergence properties of the power iteration algorithm, we will examine how the process converges and which eigenvalues and eigenvectors are produced. The eigenvectors of the matrix $\mathbf{A} \in \mathbb{R}^{m\times m}$ form a basis for $\mathbb{R}^m$, so we will expand our initial guess in this basis,

$$\vec{x}^{(0)} = x_0^{(0)}q_0 + x_1^{(0)}q_1 + \ldots + x_{m-1}^{(0)}q_{m-1} \ . \tag{3.14}$$

We will assume that, given $x_0 \neq 0$, our initial guess must have some component in the direction of the largest eigenvalue λ_0. If we follow our power iteration algorithm through k steps, we have applied the matrix $\mathbf{A}$ that many times to this initial guess,

$$\vec{x}^{\,(k)} = \mathbf{A}^k\vec{x}^{\,(0)} \ .$$

Substituting in our expanded form of the initial guess from Eq. (3.14),

$$\begin{aligned}
\vec{x}^{\,(k)} &= \mathbf{A}^k\left(x_0^{(0)}q_0 + x_1^{(0)}q_1 + \ldots + x_{m-1}^{(0)}q_{m-1}\right) \\
&= x_0^{(0)}\lambda_0^k q_0 + x_1^{(0)}\lambda_1^k q_1 + \ldots + x_{m-1}^{(0)}\lambda_{m-1}^k q_{m-1} \\
&= \lambda_0^k\left[x_0^{(0)}q_0 + x_1^{(0)}\left(\frac{\lambda_1}{\lambda_0}\right)^k q_1 + \ldots + x_{m-1}^{(0)}\left(\frac{\lambda_{m-1}}{\lambda_0}\right)^k q_{m-1}\right] \ .
\end{aligned}$$

Since λ_0 is the largest eigenvalue, as the number of iterations k increases, our guess approaches its eigenvector. The speed of convergence depends on how quickly the other components are suppressed. The convergence rate is based on the dominance ratio,

$$\left\|\frac{\lambda_1}{\lambda_0}\right\| \ . \tag{3.15}$$

If the problem of interest has a second dominant eigenvalue, this value may be close to unity and convergence can take arbitrarily long.

3.5.2 Application to the Transport Equation

We will now apply this power iteration method discussed in to the transport equation. We will begin with the operator form of the transport equation Eq. (3.2)

$$\mathbf{H}\vec{\psi} = \widetilde{\mathbf{S}}\vec{\psi} + \frac{1}{k}\widetilde{\mathbf{F}}\vec{\psi} \ . \tag{3.2, revisted}$$

This is an eigenvalue problem with the form,

$$\left(\mathbf{H} - \widetilde{\mathbf{S}}\right)^{-1}\widetilde{\mathbf{F}}\vec{\psi} = k\vec{\psi} \ .$$

If we substitute the matrix $\mathbf{A} = (\mathbf{H} - \widetilde{\mathbf{S}})^{-1}\mathbf{F}$ into our scheme for applying the matrix in power iteration, we see

$$\begin{aligned} \vec{\psi}^{(k+1)} &= \left(\mathbf{H} - \widetilde{\mathbf{S}}\right)^{-1}\mathbf{F}\vec{\psi}^{(k)} \\ \left(\mathbf{H} - \widetilde{\mathbf{S}}\right)\vec{\psi}^{(k+1)} &= \mathbf{F}\vec{\psi}^{(k)} \ . \end{aligned}$$

To solve the k-eigenvalue problem We will repeatedly apply the matrix $(\mathbf{H} - \widetilde{\mathbf{S}})^{-1}\mathbf{F}$ as described in the power-iteration algorithm:

1. Begin with an initial guess $\vec{\psi}^{(0)}$,
2. Apply the fission operator to the angular flux from the previous step (or the initial guess) $\mathbf{F}\vec{\psi}^{(k)}$,
3. Use this as the source term $\mathbf{Q}$ for the GS formulation of Eq. (3.8), and solve for $\psi^{(k+1)}$ using that scheme,
4. Calculate a new k using the Rayleigh quotient,
5. Repeat steps 2–4 until the k-eigenvalue has converged within some tolerance.

In this scheme, the power iterations are the eigenvalue iterations, which update the fission source through the application of the fission operator. Some method is still required to converge the scattering source, such as the previously GS iterative scheme. Some combination of power iteration and a scattering source iteration scheme are found in all deterministic codes that solve the k-eigenvalue problem. As this combination relies on the power iteration scheme, slow convergence will occur for problems with dominance ratios close to unity. Convergence is also slowed when the scattering source iteration scheme is inefficient, such as problems with a large amount of scattering that use the SI scheme. To alleviate some of these issues, acceleration methods have been developed, such as the NDA method described in the following section.

3.6 Nonlinear Diffusion Acceleration

The nonlinear diffusion acceleration (NDA) method [16, 8] is similar to the TG method in that it uses a higher-order angular solve that is accelerated by a lower-order scalar diffusion solve. But, unlike the TG method, which performs an additive correction to each step of the GS outer iteration, the NDA method modifies the inner iterative scheme. This method uses a low-order diffusion formulation to solve for the scalar flux, using an angular formulation as a closure. While the TG method is designed to speed up outer-iteration convergence of all groups in the presence of upscattering, NDA has been shown to improve the speed of within-group convergence. This section will describe the derivation and application of the NDA method. A full and comprehensive derivation can be found in Hammer *et al.* [8].

3.6.1 Derivation

We begin with the diffusion equation, Eq. (2.22), but with the current form of the first term and the fission source contained in a constant source term,

$$\left[\nabla \cdot \vec{J}_g(\vec{r}) + \Sigma_t^g(\vec{r})\right] \phi_g(\vec{r}) = \sum_{g'=0}^{G-1} \Sigma_{s0}^{g'\to g} \phi_{g'}(\vec{r}) + q_g(\vec{r}) \ . \tag{3.16}$$

To arrive at the diffusion equation, we would use Fick's law,

$$J_g(\vec{r}) = -D_g \nabla \phi_g(\vec{r}) \ ,$$

as a closure to this problem. Use of this closure would be exactly correct if the flux was perfectly diffusive, and is a very good approximation in equations with high scattering. But in reality, the flux will not act perfectly diffusively so using the angular flux, if known, should provide a better value for the current than Fick's law.

The NDA method uses a combination of the two, both the diffusive formulation of Fick's law, and a known angular flux. To do this, we will add the angular formulation of the current to Fick's law as a correction:

$$\begin{aligned}
J_g(\vec{r}) &= -D_g \nabla \phi_g(\vec{r}) + J_g(\vec{r}) - J_g(\vec{r}) \\
&= -D_g \nabla \phi_g(\vec{r}) + \int \hat{\Omega} \psi_g(\vec{r}, \hat{\Omega}) \, d\hat{\Omega} + D_g \nabla \phi_g(\vec{r}) \\
&= -D_g \nabla \phi_g(\vec{r}) + \left[\frac{\int \hat{\Omega} \psi_g(\vec{r}, \hat{\Omega}) \, d\hat{\Omega} + D_g \nabla \phi_g(\vec{r})}{\phi_g(\vec{r})} \right] \phi_g(\vec{r}) \\
&= -D_g \nabla \phi_g(\vec{r}) + \hat{D}(\vec{r}) \phi_g(\vec{r}) \ ,
\end{aligned}$$

where we define the drift-diffusion vector $\hat{D}(\vec{r})$. This additive correction should improve the value of our current when informed by the angular flux. Plugging this into the integrated transport equation gives the low-order nonlinear diffusion acceleration (LO-NDA) equation,

$$\nabla \cdot \left[-D_g \nabla + \hat{D}_g(\vec{r})\right] \phi_g(\vec{r}) + \left(\Sigma_t^g - \Sigma_s^{g\to g}\right) \phi_g(\vec{r}) = \sum_{g'\neq g} \Sigma_s^{g'\to g} \phi_{g'}(\vec{r}) + q_g(\vec{r})\,. \tag{3.17}$$

We will solve an angular form of the transport equation, the high-order nonlinear diffusion acceleration (HO-NDA), and use the result to calculate the drift diffusion term. Then we solve Eq. 3.17 holding the scattering source constant and updating the fission source until convergence. The solved scalar flux is then used to update the scattering source for the HO-NDA and the angular flux equation is solved and the process repeated.

3.6.2 Weak Form Derivation

To solve the LO-NDA equation using the finite-element method, we must derive the weak formulation. The LO-NDA is nearly identical to the diffusion equation, with the extra drift-diffusion term. The weak form is therefore,

$$\begin{aligned}\Big(\nabla v(\vec{r}), D_g(\vec{r})\nabla\phi_g(\vec{r})\Big)_{\mathbb{D}} + \Big(\nabla v(\vec{r})\hat{D}_g(\vec{r})\phi_g(\vec{r})\Big)_{\mathbb{D}} + \Big\langle v(r), \vec{J}_{\text{net}}(\vec{r})\Big\rangle_{\partial\mathbb{D}} \\ + \Big(v(\vec{r}), \Sigma_r^g(\vec{r})\phi_g(\vec{r})\Big)_{\mathbb{D}} = \Big(v(\vec{r}), \sum_{g'\neq g}\Sigma_{s0}^{g'\to g}\phi_{g'}(\vec{r})\Big)_{\mathbb{D}} + \Big(v(\vec{r}), q_g(\vec{r})\Big)_{\mathbb{D}}\,.\end{aligned}$$

As with the standard diffusion equation in the reflective boundary case, the incoming and outgoing fluxes are equal, so the net current is simply zero. For the vacuum boundary case, the net current is simply the outgoing current and we can calculate the outward current using the higher-order solve in a similar process to the drift-diffusion vector [8].

$$J_{\text{out}} = \frac{1}{4}\kappa\phi_g(\vec{r})$$

where the correction κ is calculated using the scalar flux from the higher-order solve, $\phi_g^{\text{HO}}(\vec{r})$:

$$\kappa = \frac{4}{\phi_g^{\text{HO}}(\vec{r})}\int_{\hat{n}\cdot\hat{\Omega}>0} |\hat{n}\cdot\hat{\Omega}|\psi_g(\vec{r},\hat{\Omega})d\hat{\Omega}\,.$$

Substituting this in gives the final weak form of the LO-NDA equation in the vacuum boundary case,

$$\begin{aligned}\Big(\nabla v(\vec{r}), D_g(\vec{r})\nabla\phi_g(\vec{r})\Big)_{\mathbb{D}} + \Big(\nabla v(\vec{r})\hat{D}_g(\vec{r})\phi_g(\vec{r})\Big)_{\mathbb{D}} + \Big\langle v(r), \frac{1}{4}\kappa\phi_g(\vec{r})\Big\rangle_{\partial\mathbb{D},\text{vacuum}} \\ + \Big(v(\vec{r}), \Sigma_r^g(\vec{r})\phi_g(\vec{r})\Big)_{\mathbb{D}} = \Big(v(\vec{r}), \sum_{g'\neq g}\Sigma_{s0}^{g'\to g}\phi_{g'}(\vec{r})\Big)_{\mathbb{D}} + \Big(v(\vec{r}), q_g(\vec{r})\Big)_{\mathbb{D}}\,.\end{aligned}$$

3.6.3 Algorithm

The steps of the NDA process are therefore,

1. Solve an angular form of the transport equation such as Eq. (3.2), the HO-NDA equation,

2. Calculate the drift-diffusion vector, $\hat{D}(\vec{r})$ using the higher-order solution,

$$\hat{D}(\vec{r}) = \frac{\int \hat{\Omega}\psi_g(\vec{r},\hat{\Omega})\, d\hat{\Omega} + D_g \nabla \phi_g^{\mathrm{HO}}(\vec{r})}{\phi_g^{\mathrm{HO}}(\vec{r})} \,,$$

   and calculate κ if required.

3. Solve the LO-NDA equation, Eq. (3.17), by holding the scattering source constant and updating the fission source,

4. Use the solved low-order scalar flux to update the scattering source term of the HO-NDA.

The NDA algorithm has been shown to improve the convergence of the k-eigenvalue problem in the presence of large amounts of scattering [8, 16].

3.7 Conclusion

In this chapter, we presented the methods by which we will solve the discretized transport equation. We discussed the source iteration and GS methods, which together solve fixed source transport problems. We also presented the two-grid acceleration scheme, which improves convergence of GS in the presence of a large amount of scattering and, in particular, upscattering. Power iteration provides a method to converge the k-eigenvalue form of the transport equation. Convergence challenges due to high within-group scattering with this method can be helped with the use of the nonlinear diffusion acceleration method. In the next chapter, we will discuss the mathematical basis for these methods, and the challenges of implementing and assessing their effectiveness.

Chapter 4

Assessment of Acceleration Methods

In the previous chapter we discussed the iterative methods used to solve the discretized transport equation. As described, these iterative methods have properties that can make convergence challenging or impossible, motivating the development of acceleration methods. These methods were developed through a mathematical understanding of the iterative solving process and are designed to combat the well understood convergence issues. For researchers interested in developing acceleration methods, this is only the first step. Once developed, there are three major challenges that must be overcome before the method can be published: implementation, assessing effectiveness, and validation.

The first of these challenges is practical in nature; once a method is developed, we must implement this method in some way to show that it works. The mathematical derivation of the method has limitations, often requiring assumptions about the solving domain and dimension that are almost certainly invalidated in more complex problems of interest. Therefore, researchers are expected to create or update a code that solves the transport equation to show that the method is effective when put into practice. Each of these options includes unique challenges and benefits, which will be discussed in this chapter. Once implemented, researchers can move on to assessing the effectiveness of the method.

Assessing the effectiveness of acceleration methods is often measured in iterations or clock time. These measurements of efficiency are motivated by the end-users of acceleration methods: researchers executing large production-scale codes to solve complex problems of interest. Ultimately, acceleration methods are efficient if they solve problems using less computational work, which is usually reflected in iterations or wall clock time. In this chapter, we will discuss these heuristics and associated assumptions.

The final challenge in assessing acceleration methods is validation. While users are often interested in a method merely being efficient, this may not be enough for researchers developing methods. The mathematical basis for a method often dictates how we expect the method to improve convergence. To validate whether the expected reason for improvement is happening correctly, specific data needs to be collected that is often not available in transport codes.

In this chapter, we will explore these challenges. We will discuss the practical challenges of implementation of acceleration methods and the barriers that exist. We will also discuss the assessment of acceleration methods, including the use of iterations as a heuristic for work. Finally, we will discuss the challenge of validation.

4.1 Implementation

Collecting good experimental data on the operation of novel acceleration methods can be a major practical challenge. Before even discussing how to measure effectiveness, we need to overcome the practical matter of implementing the method in some form. In general, there are two possible paths developers can take: they can develop their own code, or implement the novel method in an existing code base. Both of these approaches have benefits and challenges in an academic environment and we shall examine each.

4.1.1 Developing New Codes

One approach is to fully develop a new code to implement and experiment with a novel acceleration method. The first major barrier to creating a new code is proficiency in the programming language of choice. Although most graduate students and professors now possess some skill in one or more programming languages, a deep enough understanding of a language to build and maintain a new code of any real quality may be beyond their reach. A 2013 survey of institutions of higher learning in the United States [19] shows that of students taking a first programming course, 44% learned Java, 19% learned C++ and 17% learned Python. The survey indicates that in academic environments, some compiled languages like Java are slowly becoming less popular than interpreted languages like Python. For a first computing course, which the authors indicate is more geared towards general academic programming, the top languages taught are MATLAB and Python.

The impact of what languages graduate or undergraduate students know is important when developing a new code to test an acceleration method. Many essential computing libraries such as Portable, Extensible Toolkit for Scientific Computation (PETSc) [20] and Trilinos [21] are designed for use in C or C++ applications, and many have been adapted to work in Python. Adapting these libraries for use in a popular programming language such as Java may be difficult if a library does not already exist. The use of an interpreted language such as Python may be non-ideal for complex problems that require a large amount of computation. In many cases they are less efficient than a compiled language such as C or C++ that has the benefit of compile-time optimization.

The final challenge when developing a new code to test an acceleration method is code quality. Writing tested and well-written code is a time consuming process, which may be cast to the wayside in the time and resource strapped environment of academia. The graduate students and academics writing the code may not have the background to write testable code, or the time to learn the skills necessary. Unfortunately, the publishing focus is on the results of the numerical experiments, not the methods by which they were collected. This situation may lead to researchers trusting code that may be acting in ways they do not expect, which is not ideal for generating good experimental data.

Despite these challenges, there are benefits to developing a new code. First, the researcher has full control over the code design and execution. Many large existing code bases may have optimizations or subroutines that are not obvious to a developer or user, and may affect the quality of experimental data produced. By fully designing and building a code, researchers know exactly what their code is doing so they can produce more trustworthy experimental data. Second, if researchers already possess a good understanding of a coding language and good software practices, it may be faster to develop a new code base than to learn how to modify an existing one. Large deterministic codes are, in general, designed to be as fast as possible and are therefore not necessarily easy to modify. Finally, designing and writing a full new code can make it easier for others to replicate the work. Instead of needing to download a large code and then apply modifications, researchers can download the presumably smaller experimental code and run their own or the same test cases.

4.1.2 Modifying Existing Codes

The second approach is to modify an existing code by adding the novel acceleration method. The major benefit to this approach is that a majority of the solving process has already been implemented and proven to work. Depending on the code used, inserting a novel method into the solving process can be significantly faster and easier than writing a completely new code. Similar issues surrounding general programming knowledge factor in when modifying codes. In this case, researchers have no choice for what programming language is required since that decision has already been made. Many "production scale" codes are written in highly optimized compiled languages such as C++ or C. These production codes are designed to be fast and efficient, and therefore may use esoteric and modern coding practices that may not be within the technical expertise of researchers and students.

The use of coding frameworks like Multiphysics Object Oriented Simulation Environment (MOOSE) [22] provides an excellent opportunity for this type of development. This framework provides a developer-centered approach to designing finite element applications for solving various types of problems. Although extremely optimized and well written, the framework may be too complex for the simple application of an acceleration method. Researchers may also want more control over what the code is doing, requiring in-depth study of how the MOOSE framework operates. It cannot be overlooked, however, that a major benefit of using existing codes is that in general these codes are well tested and understood, providing assurances that experimental data collected is good.

Regardless of the path taken to implement a novel acceleration method, this is only the first practical challenge in testing its effectiveness. Researchers must now collect and assess data to determine if the method is working and, more importantly, working for the reasons expected.

4.2 Assessment of Efficiency

A transport solve is a complex series of interlocking routines that will ultimately lead (hopefully) to a convergent solution. As we have described, acceleration methods are designed to combat inefficiencies in these routines to improve the converging behavior. Different schemes will modify various routines or subroutines, so determining if the method has worked can be challenging. To discuss evaluation of acceleration methods, we will use two inexact but useful terms: upstream and downstream measures. We will compare means of measuring acceleration by first describing a relative proximity in the solving process to the modified routine.

The most downstream measure of acceleration is binary: if the solve converges or not. As discussed, in media with a very high scattering ratio the Gauss-Seidel (GS) iteration processes can take arbitrarily long to converge. This means that some problems solved with GS may take an unreasonable amount of time to converge, or may not converge at all due to the limitations of the computer's accuracy. If an acceleration method is designed to address this inefficiency and implementation of the method results in a non-convergent problem converging, then we can reasonably make the assertion that the method is effective. We will refer to this as the most downstream possible measure of acceleration effectiveness.

From a practical standpoint, if convergence of a non-converging problem is our goal this may be the only measure that we need. From the standpoint of analyzing acceleration methods and their effectiveness, it is less ideal. Although a method causing convergence clearly had an impact on the problem, claiming the method was effective relies on a few assumptions. In the case of GS, the first assumptions is that the arbitrarily slow convergence was due to the high scattering ratio. Second, that implementation of the acceleration method addressed this particular inefficiency. And third, that fixing this particular inefficiency caused the problem to ultimately converge. In many cases these are valid assumptions to make; but these are still assumptions that could be analyzed for their validity. To analyze the method further, we need more and better data that is closer to the point of acceleration: further upstream.

The most common means of verifying the effectiveness of an acceleration method is a comparison of number of iterations. The solve is run at least twice, once accelerated and once unaccelerated.[1] The number of iterations is then compared to determine the effectiveness of the method. As with analyzing convergence as a binary measure, this relies on a number of assumptions. The most important of these assumptions is that the number of iterations is a good heuristic for indicating the amount of work required to converge the problem. No matter the methods involved, convergence of the problem relies on the removal of a finite amount of error. If this error removal is performed in N iterations, the total work is merely

[1]The use of *nonaccelerated* indicates that a solve algorithm has no acceleration methods implemented. *Unaccelerated* will indicate routines within a solve that have access to an implemented acceleration method but are not utilizing it.

the sum of the work of each iteration,

$$W_{\text{total}} = \sum_{k=0}^{N-1} w^{(k)} \ , \tag{4.1}$$

where $w^{(k)}$ is the work performed by iteration k.

The goal of our acceleration methods is to remove the same amount of error with less total work. Our acceleration method is efficient only if

$$\sum_{k'=0}^{N'-1} w'^{(k')} < \sum_{k=0}^{N-1} w^{(k)} \ ,$$

where N' is the total number of iterations in our accelerated solve and $w'^{(k')}$ is the error removed by the accelerated solve in iteration k'. It is important to note that in the accelerated solve, both the total number of iterations and the work done in each iteration can be different but what indicates an efficient acceleration is the total work. If we create a subroutine that performs a large amount of work in each iteration but reduces the total number iterations, we can significantly reduce the total iterations while not reducing the total work.

The development of acceleration methods is driven by a desire to solve larger and more complex transport problems more efficiency. This is driven by large production-scale codes that are designed to operate on super computers. For the users of these codes, wall clock time or computational time are the most important measures of work. Unfortunately, for those who wish to compare the effectiveness of different acceleration schemes, time required is not an easily controlled variable. The amount of computational or wall clock time can vary wildly depending on the type of computer used, the loading of the computer at the time, and a vast array of other variables that cannot be controlled. We will therefore abandon the use of the wall clock time and move on to a different heuristic often used to measure efficiency, inversions.

For most solves, each iteration indicates an inversion of the transport operator, which is assumed to be the most computationally expensive of the subroutines. This inversion may involve an explicit inversion of a formed matrix, or an implicit inversion through a transport sweep. Inversion of the matrix scales $\mathcal{O}(n^3)$ with matrix size n compared to matrix-vector multiplication which scales $\mathcal{O}(n^2)$. In most cases, this inversion process will therefore dominate the computational cost of the solve. Our equation for work in Eq. (4.1) can be approximated

$$W_{\text{total}} = \sum_{k=0}^{N-1} w^{(k)} \approx \sum_{k=0}^{N-1} w_{\text{inv}} = N w_{\text{inv}} \ ,$$

where w_{inv} is the computational work required for each inversion of the transport operator.

The addition of a subroutine in our acceleration method complicates our model of work. If the work of our acceleration subroutine in each iteration is w', the total work of our accelerated scheme is

$$W'_{\text{total}} = \sum_{k'=0}^{N'-1} w'^{(k')} \approx \sum_{k'=0}^{N'-1} (w_{\text{inv}} + w') = N' (w_{\text{inv}} + w') \ .$$

If we assume that our acceleration routine work is negligible compared to the work of inverting the transport operator $w' \ll w_{\text{inv}}$,

$$W'_{\text{total}} \approx N' (w_{\text{inv}} + w') \approx N' w_{\text{inv}} \ . \tag{4.2}$$

In this case, meeting our convergence criteria with fewer iterations should indicate that we have reduced the same amount of error with fewer inversions and therefore less computational work. As the value of w' increases, using the number of iterations becomes less useful when determining total work.

In the formulation just presented, each iteration corresponds to a single inversion of the transport operator and a single execution of the acceleration subroutine. In an extreme case, many inversions worth of work can be performed by a single execution of the acceleration subroutine or the subroutine might be executed many times per inversion. This would significantly reduce the value of using N' as a measure of performance by violating the assumption that $w' \ll w_{\text{inv}}$.

Thus, as our acceleration routines become more complex and do more work, we cannot perform the simplification seen in Eq. (4.2). In this case our acceleration method is efficient only if

$$\begin{aligned} W'_{\text{total}} &< W_{\text{total}} \\ N' (w_{\text{inv}} + w') &< N w_{\text{inv}} \\ N' w' &< (N - N') w_{\text{inv}} \\ N' w' &< \Delta N w_{\text{inv}} \ , \end{aligned} \tag{4.3}$$

where $\Delta N = (N - N')$. Our acceleration method is only worthwhile if the amount of additional work required to converge the problem, $N'w'$, is less than the work saved by not performing ΔN extra inversions. We also see that a method can never be efficient if $N < N'$. This is a consequence of every iteration requiring at least one inversion of the transport operator.

Measuring the values of w_{inv} and w' in most cases is impractical due to the routines used for the inversion and the complexity of some acceleration methods. This heuristic is further complicated if we combine acceleration methods, either to accelerate the base problem in two or more ways or accelerate an acceleration method. What we can do is compare the ratios of work using our understanding of the methods to help us decide if a method is efficient. Dividing Eq. (4.3) by the work to invert and the total iterations in our accelerated solve we see,

$$\frac{w'}{w_{\text{inv}}} < \left(\frac{N}{N'} - 1 \right) \ .$$

In Fig. 4.1 we see a plot of this inequality. The x-axis is the ratio of the acceleration method work to the inversion of the transport operator and the y-axis shows the ratio of the accelerated iterations to the unaccelerated iterations. The region under the curve indicates

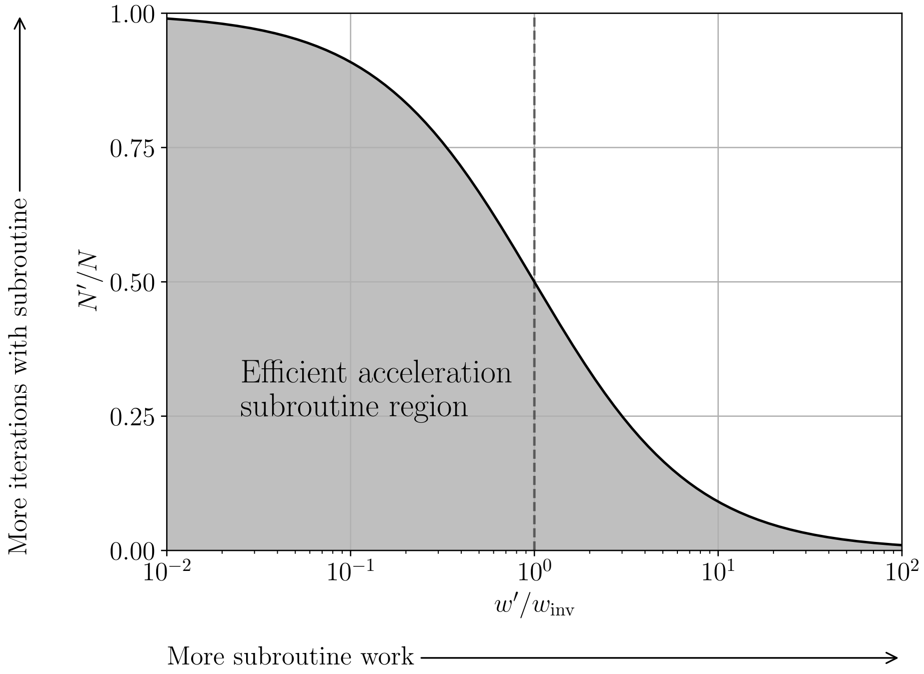


Figure 4.1: Ratio of accelerated to unaccelerated iterations vs ratio of acceleration method work to the work of inversion. The shaded region indicates the area where acceleration subroutines are efficient. The vertical dotted line indicates a subroutine that requires exactly as much work as an inversion of the transport operator.

the region where acceleration methods are efficient. As expected, if the acceleration method is doing the exact amount of work as a single inversion, we require convergence in exactly half the iterations and any fewer indicates acceleration. As our method becomes more efficient and w'/w_{inv} becomes less than unity, there is a larger region where our method is efficient, and as $w' \ll w_{\mathrm{inv}}$, any reduction in iterations indicates an efficient method. There are regions of efficiency where $w' > w_{\mathrm{inv}}$, assuming the number of iterations in the accelerated solve is low enough. Much like the use of inversions to represent work, this is merely a heuristic but it can add insight. When assessing, care must be taken to ensure that we are examining the correct number of inversions, which can be complicated by the use of both inner and outer iterations.

Without a large amount of analysis and many simplifying assumptions, we can't truly measure the total work performed by a method. We can use this model to make educated conclusions about the efficiency of an algorithm. A complicating factor is the required overhead of implementing a subroutine. The additional work done by an acceleration method is not just the work done by the derived mathematical acceleration scheme, but also any work done that comes from the code modifications to run the subroutine,

$$w' = w_{\text{subroutine}} + w_{\text{implementation}} \ . \tag{4.4}$$

To be clear, this implementation work is not work done by the subroutine to execute the acceleration method, but the surrounding logic required to perform the subroutine. To make good assessments about the effectiveness of our acceleration subroutine, we must minimize the computational work done by the implementation; this work is typically small but it must still be considered. Many subroutines perform a solve of some form of the transport equation, such as those that accelerate an angular solve using a diffusion solve. In these cases, it would be ideal for both solves to be as identical as possible in implementation except where changes are necessary.

4.3 Validation

As discussed earlier, determining if an acceleration method is efficient by measuring the number of iterations is only one part of assessment. Overall convergence is the most downstream possible measure, with the total iterations being only slightly more upstream. The benefit to using total iterations is that it gives us a standard candle to use when comparing different acceleration methods, and in many cases relates well to wall clock time required. This is of particular interest to researchers looking to implement methods in production level codes for faster convergence. For researchers interested in developing and modifying acceleration methods, this data is useful but there are much more interesting metrics to collect. When developing new acceleration methods, modifying existing methods, or combining them, we are interested in not just the overall acceleration but the reasons behind it.

To determine if our subroutine is working and working in the way we expect, we need to collect data from much closer to, or from within the subroutine itself. The type of data required and the collection location will vary greatly between different acceleration schemes. Collecting and analyzing this data is important for a number of reasons. First, we have designed acceleration methods to not just be more efficient, but to be more efficient for reasons identified mathematically. For complex problems, we expect these mathematical derivations to still hold, and so we would like to collect data to test this hypothesis. Second, we are comparing an accelerated solve to an unaccelerated solve that necessarily use different code. We need to make sure that our accelerated solve is more efficient because of the mathematical properties of the method, not due to coding practices or processes.

One tool often used by researchers to develop acceleration methods is Fourier analysis. This process can help reveal inefficiencies in the iterative process that can be targeted by acceleration methods. The ability to do fast discrete Fourier transforms then provides the ability to validate if the acceleration method is operating as we expect.

4.4 Fourier Analysis

A large body of research has focused on accelerating the convergence of source iteration (SI) by speeding up the convergence of the scattering portion of the formulation (generally referred to as the scattering source term or scattering source). A thorough discussion of the history of these methods can be found in Adams and Larson [12].

As we discussed in the previous chapter, development of preconditioning schemes began in the 1960s with the near concurrent work of Kopp [13] and Lebedev [14]. Here, we see the use of scalar (non-angular) formulations to accelerate angular formulations, using the understanding that the scalar form is much more efficient when dealing with more diffusive neutron behavior. Their work has been expanded on for many years, leading to the development of various acceleration schemes including diffusion synthetic acceleration (DSA) [15]. From the physical interpretation of the SI process, we know why these methods work. If the k-th iteration corresponds to calculating the flux for neutrons that have scattered at most $k-1$ times, using the diffusion method will help propagate those neutrons more efficiently in diffusive problems.

Mathematically, we can show this using Fourier analysis [12]. We begin our analysis with an infinite one-spatial dimension, single energy version of source iteration on convex domain $\mathcal{D} \subset \mathbb{R}^1$ with position and angle

$$\{x \in \mathbb{R} \mid x \in \mathcal{D}\}$$
$$\{\mu \in \mathbb{R} \mid \mu^2 < 1\} \; .$$

The transport equation in an infinite homogenous medium with isotropic scattering is given by,

$$\mu \frac{\partial}{\partial x}\psi(x,\mu) + \Sigma_t \psi(x,\mu) = \frac{\Sigma_s}{2}\int_{-1}^{1} \psi(x,\mu')d\mu' + \frac{Q}{2} \; .$$

This equation in the SI form is,

$$\mu \frac{\partial}{\partial x}\psi^{(k+1)}(x,\mu) + \Sigma_t \psi^{(k+1)}(x,\mu) = \frac{\Sigma_s}{2}\int_{-1}^{1} \psi^{(k)}(x,\mu')d\mu' + \frac{Q}{2} \; .$$

With error,

$$e^{(k)}(x,\mu) = \psi^*(x,\mu) - \psi^{(k)}(x,\mu) \; ,$$

where ψ^* is the exact solution. We can subtract the iterative scheme from the exact solution to get an equation for the error in iteration k,

$$\mu \frac{\partial}{\partial x}e^{(k+1)}(x,\mu) + \Sigma_t e^{(k+1)}(x,\mu) = \frac{\Sigma_s}{2}\int_{-1}^{1} e^{(k)}(x,\mu')d\mu' \; . \qquad (4.5)$$

To examine the modes of spatial variation in our error, it is useful to use the inverse Fourier transform. This mapping $f : \mathbb{C} \mapsto \mathbb{R}$ will express the error in terms of a spatial frequency. First, we must decide on a measure of spatial variation, a characteristic distance. One way that provides some physical intuition is to examine how error varies as measured in mean-free paths ℓ, giving a wavelength,

$$\lambda = \frac{\ell}{n}, \forall n \in \mathbb{R} \ . \tag{4.6}$$

The larger the value of n, the higher the frequency of the error over our characteristic length. These error modes are also assigned a linear wave number $\tilde{\nu}$, defined using Eq. (4.6).

$$\tilde{\nu} = \frac{1}{\lambda} = \frac{n}{\ell} = n \cdot \Sigma_t, \forall n \in \mathbb{R} \ . \tag{4.7}$$

Using the wave number, we can perform an inverse Fourier transform in space of our error, using our parameter n,

$$e^{(k)}(x,\mu) = \int_{-\infty}^{\infty} \hat{e}^{(k)}(n,\mu) e^{i\Sigma_t n x} dn \ ,$$

We can substitute this into Eq.(4.5) to get a recursion relationship for the error in each iteration,

$$\int_{-\infty}^{\infty} \left[\Sigma_t \left(i\mu n + 1\right) \hat{e}^{(k+1)}(n,\mu) - \frac{\Sigma_s}{2} \int_{-1}^{1} \hat{e}^{(k)}(n,\mu')d\mu' \right] e^{i\Sigma_t n x} dn = 0 \ .$$

The exponential basis functions of the Fourier transform e^{ikx} are all linearly independent. Therefore, the only way for this integral to be zero is by setting

$$\begin{aligned} \Sigma_t \left(i\mu n + 1\right) \hat{e}^{(k+1)}(n,\mu) &= \frac{\Sigma_s}{2} \int_{-1}^{1} \hat{e}^{(k)}(n,\mu')d\mu' \\ \hat{e}^{(k+1)}(n,\mu) &= \frac{\Sigma_s}{2\Sigma_t \left(i\mu n + 1\right)} \int_{-1}^{1} \hat{e}^{(k)}(n,\mu')d\mu' \ . \end{aligned}$$

We can also write the transformed error in terms of the scattering ratio, $c = \Sigma_s/\Sigma_t$.

$$\hat{e}^{(k+1)}(n,\mu) = \frac{c}{2} \frac{1}{\left(i\mu n + 1\right)} \int_{-1}^{1} \hat{e}^{(k)}(n,\mu')d\mu', \forall n \in \mathbb{R} \ .$$

If we integrate both sides over μ,

$$\begin{aligned} \int_{-1}^{1} \hat{e}^{(k+1)}(n,\mu)d\mu &= \int_{-1}^{1} \left(\frac{c}{2} \frac{1}{\left(i\mu n + 1\right)} \int_{-1}^{1} \hat{e}^{(k)}(n,\mu')d\mu' \right) d\mu \\ &= \frac{c}{2} \int_{-1}^{1} \hat{e}^{(k)}(n,\mu')d\mu' \int_{-1}^{1} \frac{1}{1 + i\mu n} d\mu \\ &= \frac{c}{2} \int_{-1}^{1} \hat{e}^{(k)}(n,\mu')d\mu' \cdot 2 \int_{0}^{1} \left| \frac{1}{1 + i\mu n} \right| d\mu \\ &= c \int_{-1}^{1} \hat{e}^{(k)}(n,\mu')d\mu' \int_{0}^{1} \frac{1}{1 + \mu^2 n^2} d\mu \ . \end{aligned}$$

We see we have a function that provides recursive generation of error expressions, which is a function of the parameter n,

$$\begin{aligned}\Lambda(n) &= c \cdot \int_0^1 \frac{d\mu}{1+\mu^2 n^2} \\ &= \frac{c}{n} \tan^{-1}(n\mu)\Big|_0^1 \\ &= \frac{c}{n} \tan^{-1}(n) \ .\end{aligned}$$

Now we have a function that describes how the angle-iterated error in spatial frequency space evolves from iteration to iteration as a function of our parameter n and the scattering ratio c

$$\int_{-1}^{1} \hat{e}^{(k+1)}(n,\mu)d\mu = \Lambda(n) \int_{-1}^{1} \hat{e}^{k}(n,\mu')d\mu' \ .$$

This allows us to express error in iteration k in relationship to the initial error,

$$\int_{-1}^{1} \hat{e}^{k}(n,\mu)d\mu = \Lambda^{(k)}(n) \int_{-1}^{1} \hat{e}^{0}(n,\mu')d\mu' \ .$$

We now see that the error is an eigenfunction of our iteration scheme, with corresponding eigenvalue $\Lambda(n)$. The spectral radius of the iteration scheme is therefore,

$$\rho = \sup_n |\Lambda(n)| = c \ .$$

As a problem introduces more scattering as a fraction of the total cross-section, the value of c approaches unity. In each step of our iteration, the error is reduced by a factor equal to the spectral radius of the iteration matrix, which is also c. Therefore, we see the mathematical basis for the arbitrarily slow convergence of the SI scheme.

It is important to note that this spectral radius corresponds to the maximum value of $\Lambda(n)$, which occurs at a value of $n = 0$ as we see in Figure 4.2. This is the most diffusive mode of scattering and corresponds to an infinite wavelength as shown by Eq. (4.6) and this provides a mathematical basis for methods like DSA that use a diffusion approximation to improve convergence. Fourier analysis of acceleration methods can show the theoretical improvement expected. For DSA, the spectral radius of the iterative scheme has been shown to be less than or equal to $0.23c$ [15], a vast improvement. Similar analysis of the spectral radius has been performed for other iterative schemes, including nonlinear diffusion acceleration (NDA) [8], and two-grid (TG) [17].

As shown, Fourier analysis is a powerful tool to analyze and develop acceleration methods. Unfortunately, the method is not without its limitations. In many cases we are limited to analysis of a one-dimensional, infinite-media case and are limited in our ability to introduce multiple energy groups or other changes to the iterative scheme.

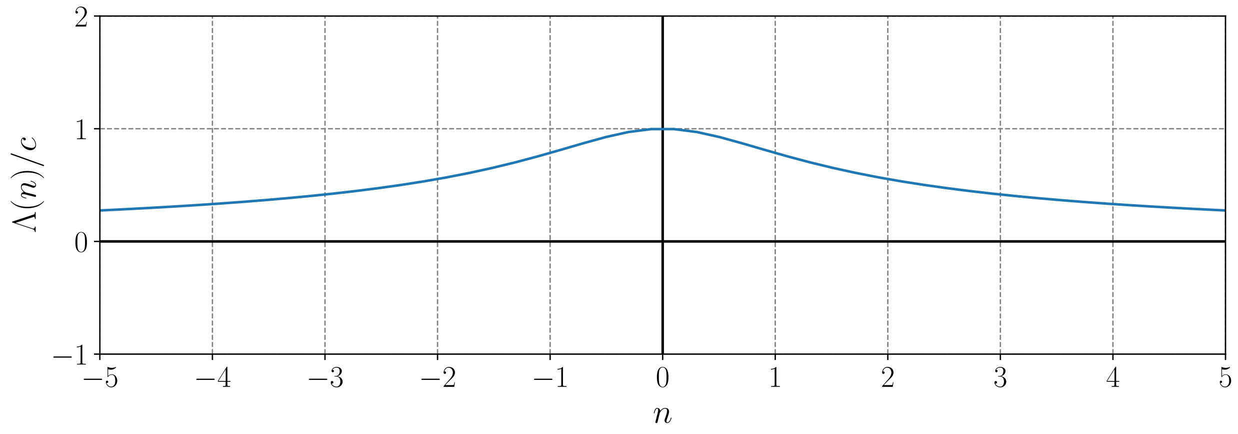


Figure 4.2: Value of $\Lambda(n)/c$ as a function of n.

Ideally, we are developing novel acceleration methods for use in improving the convergence of large, complex systems with complicated geometries and many energy groups. Therefore, experimental data is required to validate the effectiveness of the acceleration schemes using more complex models. We cannot know *a priori* what data researchers will need to validate the effectiveness of their acceleration schemes. Not only must we collect this data, but we must ensure that it is good data. The quality of the data will depend on many factors, not least of all if it separates the computer science of implementation from the mathematics of the method. A very efficient acceleration method can be undermined by inefficient computer science in implementation. In the reverse case, a very efficient acceleration method could bypass inefficient sections of the original code, showing improvement due to better computer science.

Separating the computer science from the mathematics of the subroutine can be difficult. What makes the process harder is that many codes are not designed to make this clear, precise, and easy to do. The act of modifying codes necessarily adds some amount of uncertainty to this distinction. Ideally, we could implement an acceleration method into a code while modifying as little of the rest of the solving process as possible. Luckily, this is possible using modern object oriented programming techniques, but relies on developing the code with the developer end-user in mind. The data collection itself also relies on the design of the code. It may be difficult or impossible to modify existing codes to output the data researchers are interested in when assessing their acceleration methods. The need for a code to meet these requirements motivates the development and design goals of the Bay Area Radiation Transport (BART) code, described in the next chapter.

4.5 Conclusion

In this chapter we discussed the three major challenges in assessing acceleration methods: implementation, assessment, and validation. Implementation is made more complex by the difficulty in creating or modifying codes. Once implemented, assessment and validation can require the collection of good experimental data. Many of these challenges motivate the design and development of a new code that would make implementation and analysis of the methods easier and faster. To this end, we will discuss the development of a novel code, BART in the following chapter.

Chapter 5

The BART Design and Code

Developing, implementing, and testing new acceleration methods is a difficult endeavor. The implementation is typically time consuming and resource heavy, especially if high software quality standards are to be met and maintained, which limits the speed with which we can try new methods and therefore the number of idea iterations. Further, as we will discuss in detail later, the transport community often does not have enough information to tell why a particular acceleration method works nor the measures to truly compare one method to another. Such comparisons are hindered by disparate implementations and code base differences. We developed the Bay Area Radiation Transport (BART) code to create an environment specifically for implementing and testing acceleration methods to speed development and improve assessment. The code is designed to meet four major practical challenges.

First, the difficulty of implementing new acceleration methods is a barrier to idea generation and testing. Development of a new method often begins with an analytic derivation based on mathematics and an understanding of what need the method is trying to meet. Once the method is designed, there exists the very practical need to implement the method to measure its effectiveness. Often researchers are left with few options: modifying an existing code or writing their own code. Both of these approaches have benefits and drawbacks, but both are typically resource and time intensive. Regardless of the method chosen, implementation is a major barrier between the derivation of an acceleration method and publishing the method and data about its effectiveness. Ideally, the BART code should ease some of the burden of implementing a novel acceleration method.

Once implemented, measuring the effectiveness of an acceleration method is the second major practical challenge. To do so, we must collect good data. The quality of the data we collect is greatly dependent on how we implemented the acceleration method and is dependent on two major factors. First, we need a good base case for comparison, either an unaccelerated problem or a problem using a different acceleration method. Second, once the base case is established, we need to change as little as possible between the base case and our accelerated solve. If a new code was developed to test our acceleration method, we must also ensure that this code can perform a nearly identical base case for valid comparison. If we modified an existing code, we must modify it such that as little as possible is changed, keeping in mind that a larger code may have optimizations that are unknown to the developer. Keeping this in mind, the BART code should provide an environment for developers to change as little as possible to implement an acceleration method and do so in an isolated manner such that the rest of the solving process is identical to a base case.

If we have implemented our novel method in a good environment that enables a good comparison to a base case, we must still collect good data. This can be a major challenge for large codes designed for efficiency. The data we want to validate our method may be difficult to find and extract, if not impossible. As described in Chapter 4, we have various heuristic methods to assess the efficiency of an acceleration method. Ideally, we can capture data upstream of convergence and total iterations to validate not only that the method is efficient, but the reasons for its efficiency. If we can easily collect good data, we can more quickly publish the results of novel methods and give others tools to determine if it will work in their own codes and for their particular problems of interest. It is also a way to check if the method is really correct: if it performs the way we expect given the mathematical basis we have much more confidence the method is right and is implemented correctly. We aim to create a paradigm in the BART code that makes it easy to extract data from the solving process in a minimally invasive way. In addition, the data extraction method is extensible in data type, allowing future developers to capture whatever kinds of data they may need.

Finally, assuming we have created a code for testing a novel method that provides a good environment for assessment and enables us to collect that data, we want our results to be portable and reproducible. In an ideal case, others will be able to download and compile our implementation to validate our results, or test cases that are similar to their own problems of interest. We would also like this code to be tested as thoroughly as possible to show reliability and that our results can be trusted. Not only does this improve the transparency of our research but enables others to expand and improve upon it.

We seek to overcome these major challenges with the design of the BART code. The main purpose of the BART code is to provide a controlled environment for comparing different solution and acceleration methods. The end-user for the code is not someone interested in solving the transport equation but a user interested in changing *how* the solve happens. New methods can be implemented easily and with minimal code change. In addition, BART is highly instrumented, so detailed information about solves is accessible and we can extract useful information and metrics about the solve process. Ultimately, the BART code should serve as a testing ground for proving the concept of novel acceleration methods to justify the time and resources required to implement them in production-level codes.

To summarize, the design goals of BART are to

1. Relieve the burden of implementing novel acceleration methods,

2. Provide a good environment for measuring the effectiveness of those methods,

3. Provide tools for measuring the effectiveness of methods, and,

4. Provide a tested, portable, and reliable environment.

In the following sections, we will discuss how BART meets each of these goals, following an overview of the code itself. Please note that the following sections contain code segments that are intended to illustrate the design of BART in a representative fashion. These code segments are not intended to be complete code or exactly reflective of the current state of the source code. Where possible, things such as virtual destructors and argument names have been omitted.

5.1 An Overview of the BART Code

In this section, we will provide an overview of the structure and implementation of the BART code. Some of the characteristics have been covered in the previous section where we described how they meet the design goals of the code. Therefore, these specific parts will not be repeated.

The BART source code is written in the C++ programming language and uses functionality implemented up to the C++20 [23] standard. This enables the use of many modern and time-saving C++ features that make the code more expressive and easy to understand. Many of the less common C++ features that researcher developers may not be familiar with are in classes that we do not expect them to modify. If a modern C++ feature would improve code we expect to be often modified, but at the expense of making it harder to understand, in many cases the feature was not used. Building the BART code can be accomplished using a provided build file for the CMake system.

5.1.1 Deal.II

The largest dependency that BART relies on is the deal.II finite element library [24, 25]. The BART code uses this library to provide several functions including continuous finite element basis functions and meshes for Cartesian grids. The linear solver implemented in BART is generalized minimal residual (GMRES)[26] and BART uses the deal.II implementation of the Portable, Extensible Toolkit for Scientific Computation (PETSc)[20] GMRES solver and vectors. BART implements a Gaussian cell quadrature provided by deal.II. Finally, the deal.II library supports Message Passing Interface (MPI) for parallel computing and handles domain decomposition and parallel vectors for efficient solving. BART leverages all of these library features to solve in 1, 2, or 3D and across multiple processors if desired.

As discussed in the previous chapter, the goal of BART is to provide an environment for developers that minimizes the learning curve to implementing new methods. To support this, all the deal.II methods used are wrapped by BART classes, minimizing interactions with the underlying dependency. There are, however, a few common objects used throughout BART provided by deal.II that are not wrapped and these are the deal.II vector and matrix formats and their MPI counterparts.

5.1.2 Implemented Methods and Formulations

BART supports both angular and scalar formulations of the transport equation. Three formulations are implemented for scalar solves: diffusion, drift-diffusion, and two-grid diffusion. One angular formulation is implemented at this time, the self-adjoint angular-flux (SAAF) formulation. All formulations support both vacuum and reflective boundary conditions. These formulations use the system matrix assembly procedure described in Section 2.3.3. A system "stamper" iterates over all the cells in the domain, calculates the local cell contribution to the system matrix and uses a local-to-global mapping to construct the system matrix. For one-dimension, a Gauss-Legendre quadrature is provided and for three-dimensions, a level-symmetric-like Guassian Qaudrature is provided, described in the next section.

As mentioned, power-iteration (PI) is implemented for eigenvalue iterations, Gauss-Seidel (GS) is implemented for outer iterations, and source iteration (SI) is implemented for inner iterations. Default implementations of each iteration type is provided; users merely need to identify how an iterative scheme updates the system and how it checks for convergence. For example, a PI iteration updates the fission source and checks for convergence of the k-eigenvalue. A Rayleigh Quotient calculator is implemented for calculating the k-eigenvalue. Convergence checkers for different types of data types are provided, using C++ templating to make future implementations straightforward.

Each instance of a transport equation formulation is contained in a "framework", a data structure that contains all objects and data needed to execute the solve. For describing this object, a framework "parameters" data structure specifies all the values needed to fully build a solvable framework. BART can use the deal.II parameters handler to ingest and validate input files and easily convert this to framework parameters for building problems. Builders and factories are provided for instantiating all the classes in BART and performing all required dependency injection.

5.1.3 Angular Quadrature

As described in Sec. 2.2.3, to solve the transport equation we need an implemented angular quadrature set. Our aim is to use a quadrature set that will exactly integrate the spherical harmonics which are of the form,

$$Y_{\ell,m}(\theta,\phi) = C_\ell^m P_\ell^m(\cos\theta)e^{im\phi}, \tag{5.1}$$

where $\ell \geq 0$ is the degree, $m \in [-\ell, \ell]$ is the order, C_ℓ^m is a normalization constant and P_ℓ^m are the associated Legendre polynomials. We base this decision on our method of expanding the flux and scattering operator in the spherical harmonic basis. Once solved, we can then calculate the scalar flux by combining the angular flux moments using this numerical integration. This set should contain N quadrature points in either Cartesian or polar forms, where we define the relationship between these two coordinate systems:

$$\begin{aligned} x &= \cos\phi \sin\theta \\ y &= \sin\phi \sin\theta \\ z &= \cos\theta \ , \end{aligned}$$

where ϕ is the azimuthal angle and θ is the polar angle. For one-dimension, it is easy enough to use a Gauss-Legendre quadrature set for the values of z, as this will integrate the Legendre polynomials exactly.

Historically, for higher dimensions a common angular quadrature set is the level-symmetric set. A description can be found in many sources [2, 1], and it has historically shown to be successful and efficient for solving the transport equation. One benefit of the level-symmetric set is that it is invariant in 90-degree rotations, such that no axis or direction receives preferential treatment [1]. A down side of the level-symmetric set is that it can not be procedurally generated. The ordinate choice is constrained such that there exists only one degree of freedom, choosing the first point x_0 determines all points x_i for $i \in (0, N]$. The value of x_0, the quadrature point weights, and layout of weights must all be provided in a lookup table. Therefore, we must have values provided for the specific values of N that users may choose. Ideally, we could instead generate a quadrature set that integrates the spherical harmonics with an arbitrary number of points N.

5.1.3.1 Product Gaussian Quadrature

One possible choice of procedurally generated angular quadrature is the product Guassian quadrature. A short description of the basis for this set is provided here, and a full derivation and proof of accuracy can be found in Atkinson and Han [27].

This quadrature formula performs the integral by using the product of two one-dimensional quadrature rules, one for θ and one for ϕ:

$$I(f(\theta_i, \phi_j)) = \sum_{i=0}^{N-1} w_i \sum_{j=0}^{M-1} w'_j f(\theta_i, \phi_j) \ ,$$

where N is the number of points in the quadrature rule for the θ-direction with associated weights w_i, and M is the number of points in the quadrature rule for the ϕ-direction with associated weights w'_j. Examining the form of the spherical harmonics in Eq. (5.1), we see that the portion that depends on ϕ is an exponential, $e^{im\theta}$. This exponential is periodic on the interval $[0, 2\pi]$, and can be integrated using the trapezoidal rule with uniform spacing [27],

$$\int_0^{2\pi} g(\phi) d\phi \approx I_N(g(\phi)) = h \sum_{j=0}^{M-1} g(j \cdot h) \ ,$$

where $h = \frac{2\pi}{M}$ is the spacing of the points. Note that the periodicity of the function removes the halving of the first and last terms seen in the trapezoidal rule.

In the θ-direction, we seek a quadrature set to integrate exactly the Legendre polynomials, as this is the portion of the spherical harmonics that is dependent on θ. These are polynomials that depend on $z = \cos\theta$, so we apply a Gauss-Legendre quadrature with N points over the interval $z \in [-1, 1]$. The values of θ can therefore be calculated,

$$\cos\theta_i = z_i, \quad i = 0, \dots, N-1 \ ,$$

where z_i is the Gauss-Legendre point and θ_i is the corresponding value for the product set. Our quadrature rule for the θ-direction is therefore,

$$\int_0^\pi g(\cos\theta)\sin\theta d\theta = \int_{-1}^1 g(z)dz \approx I_M(g(z)) = \sum_{i=0}^{N-1} w_i g(z_i) \ . \tag{5.2}$$

Our product Gaussian quadrature set is therefore defined as follows: choose $N > 1$ and

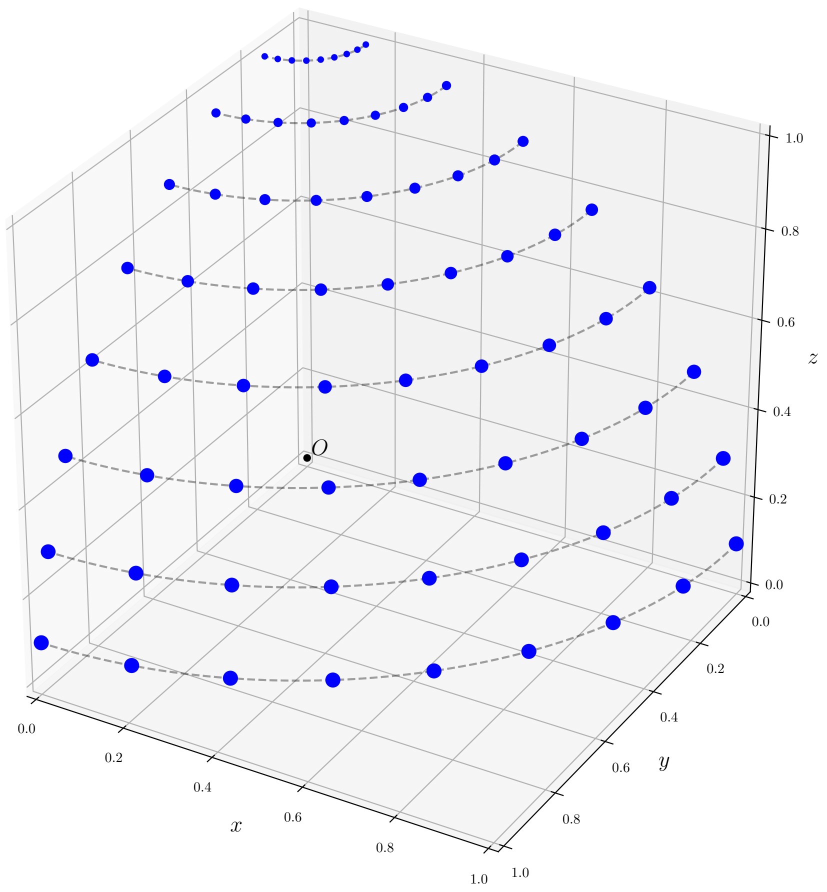


Figure 5.1: Product Gaussian quadrature set for $N = 16$. See App. B.1 for the code to generate this set and graph.

apply the trapezoidal rule with $M = 2N$ to the ϕ direction with spacing

$$h = \frac{\pi}{N},$$

and quadrature points

$$(\phi_j, w_j) = (jh, 1), \quad j = 0, \dots N-1 \ .$$

In the θ-direction, apply a Gauss-Legendre quadrature with N points (z_i, w_i) where $i = 0, \dots N-1$. This results in a product quadrature set that will exactly integrate (with appropriate normalization) a function $f(\theta, \phi)$ of the spherical harmonics using the approximation,

$$I(f) = \int_0^{2\pi} d\phi \int_0^{\pi} \sin\theta d\theta f(\theta, \phi) \approx \frac{\pi}{N} \sum_{i=0}^{N-1} w_i \sum_{j=0}^{2N-1} f(\theta_i, \phi_j) \ .$$

This product quadrature can be procedurally generated for any value $N > 1$, unlike the level-symmetric set. Unfortunately, it is not invariant under rotation; many points are clustered near the pole, as we see in Fig. 5.1. We will modify this product Gaussian quadrature to have the symmetry properties that we desire.

5.1.3.2 Level-Symmetric-Like Product Gaussian Quadrature

As stated before, the level-symmetric set has invariance under 90-degree rotation, making it symmetrical, so no axis is preferred [1]. Our product Gaussian set does not have this property, as we are using two different quadrature methods in ϕ and θ, precluding any chance for that kind of rotational symmetry. We can, though, modify the quadrature set to make it closer to the level-symmetric set. This will reduce the clustering of quadrature points near the pole, more equally space the points out through the angular phase space, and reduce the overall number of points. We will use the nomenclature of "levels", which refers to sets of quadrature points with the same θ value as shown in Fig. 5.1 by dotted lines.

We will use the product Gaussian as a basis for this set. Observing Fig. 5.1, we see that with the same number of points in each level, the points become more and more clustered with increasing z-value. To counteract this effect, we will change the number of points in each level. Again, we start with the generalized product quadrature form,

$$I(f(\theta_i, \phi_j)) = \sum_{i=0}^{N-1} w_i \sum_{j=0}^{M-1} w'_j f(\theta_i, \phi_j) \ .$$

Our interest is in modifying the distribution of points in the ϕ direction, so we will apply the same quadrature rule for θ. We will apply a Gauss-Legendre quadrature with N points in the same way, using Eq. (5.2). For the ϕ-direction, we will again apply the trapezoidal rule, but will adapt the spacing h for each level. To make the layout more symmetrical, we will place a single point per octant in the top level to mimic the ends of the distributed points on the "equatorial" $z = 0$ level.

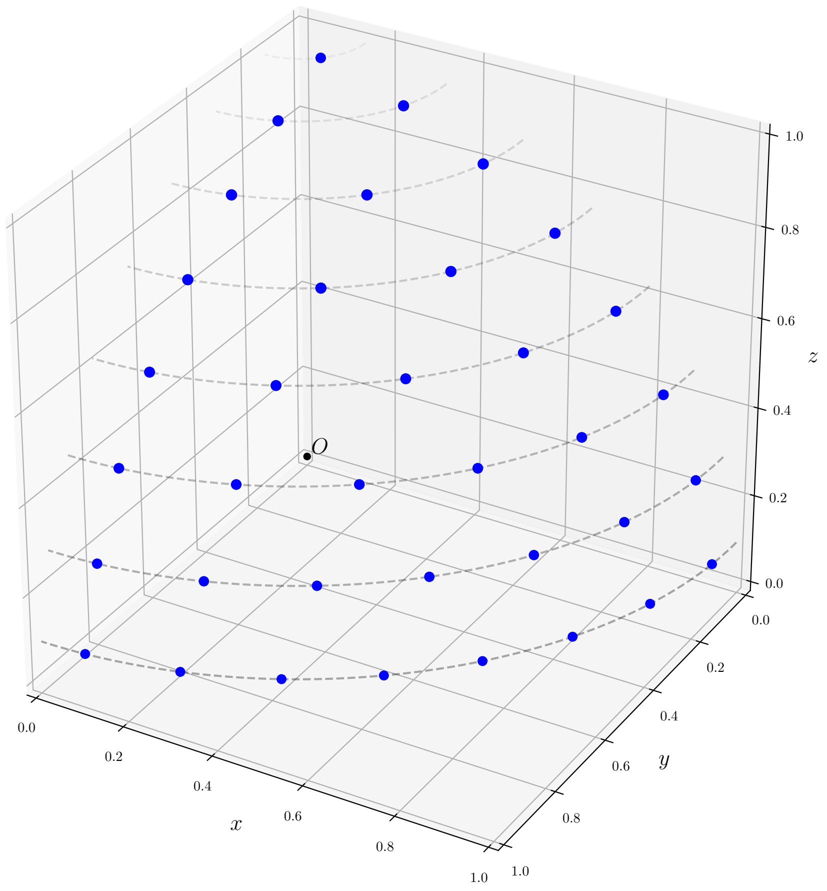


Figure 5.2: Level-symmetric-like Gaussian quadrature set for $N = 16$. See App. B.2 for the code to generate this set and graph.

With each subsequent level we will increase the number of points resulting in exactly $2N$ points on the equatorial level, the same as the product Gaussian set. We will assume that the values of θ_i are ordered such that $\theta_0 < \theta_1 < \dots \theta_{N-1}$. The value of θ_0 corresponds to the smallest value (the "highest" level) where we will place exactly four points, and increase this value by four for each subsequent value of θ. The spacing for the level corresponding to θ_i is therefore

$$h_i = \frac{2\pi}{4(i+1)}, \quad i = 0, \dots N-1 \; .$$

For this level, the quadrature points are therefore,

$$(\phi_j, w_j) = (jh_i, 1), \quad j = 0, \dots, 4(i+1) \; .$$

Combining the two quadrature rules results in the following product quadrature rule,

$$I(f) = \int_0^{2\pi} d\phi \int_0^{\pi} \sin\theta d\theta f(\theta,\phi) \approx 2\pi \sum_{i=0}^{N-1} \frac{w_i}{4(i+1)} \sum_{j=0}^{4(i+1)} f(\theta_i,\phi_j)\ .$$

A plot of this quadrature is shown in Fig. 5.2. Although not rotationally symmetrical like the level-symmetric quadrature set, this set has a more even distribution of points within the octant. As we will be using these points as collocation points for solving the transport equation, we will more efficiently solve for angular fluxes that span the space. Importantly, this set also accurately integrates the spherical harmonics. The spherical harmonics are orthogonal and normalized such that,

$$\int_0^{2\pi} d\phi \int_0^{\pi} \sin\theta d\theta Y_{\ell,m}(\theta,\phi) Y_{\ell',m'}(\theta,\phi) = \delta_{\ell\ell'}\delta_{mm'}\ .$$

We can therefore calculate an error in our numerical integration,

$$e = \left|1 - I(Y_{\ell,m}^2)\right|\ . \tag{5.3}$$

The error for the level-symmetric-like Gaussian quadrature set with $N = 16$ are shown in Table 5.1 for values of $\ell \in [0,4]$.

5.1.4 Cross-Sections

BART uses a novel cross-section format developed with Google Protocol Buffers [28]. This format enables us to describe the format of a structured data file and have the code for parsing that data file generated automatically. We have created a file structure for cross-sections, and used the protocol buffers executable to generate C++ code that is used by BART to parse the files. The most significant benefit of the protocol buffers format is that it reduces the work required to create new codes by automatically generating the code to parse the cross-section data. There is no tool for automatically converting existing cross-section formats like evaluated nuclear data file (ENDF) [29] or XML to the new protocol buffer format. But, the ability of protocol buffers to automatically generate parsing code for many programming languages means that existing codes designed to ingest and process ENDF files can be modified to output to protocol buffers easily. For the test problems included with BART, we generated parsing code for Python and used this to manually create cross-section libraries based on existing benchmark problems such as those published by Sood *et. al.* [30], Ganapol [31], and a Korea Advanced Institute of Science and Technology (KAIST) benchmark problem [32].

5.2 Reducing the Burden of Method Implementation

Typically, the end user of a transport code has less interest in the solving process than in the solution to their particular problem of interest. As long as the deterministic codes arrives

Table 5.1: Error for the level-symmetric-like Gaussian quadrature set with $N = 16$. The code used to generate these values is shown in App. B.3

ℓ	m	e
0	0	2.220446049250313e-16
	-1	2.220446049250313e-16
1	0	1.2212453270876722e-15
	1	2.220446049250313e-16
	-2	4.440892098500626e-16
	-1	0.0
2	0	1.4432899320127035e-15
	1	2.220446049250313e-16
	2	4.440892098500626e-16
	-3	4.440892098500626e-16
	-2	8.881784197001252e-16
	-1	6.661338147750939e-16
3	0	2.220446049250313e-16
	1	5.551115123125783e-16
	2	2.220446049250313e-16
	3	2.220446049250313e-16
	-4	0.0
	-3	0.0
	-2	8.881784197001252e-16
	-1	1.1102230246251565e-16
4	0	1.1102230246251565e-16
	1	2.220446049250313e-16
	2	8.881784197001252e-16
	3	0.0
	4	3.3306690738754696e-16

at the correct answer, the exact process by which it gets there is of no importance. Users typically want to solve more complex problems with higher fidelity, which forms demand for codes that utilize more powerful modern hardware, better designed software, and more efficient methods. Ultimately, users are interested in the solution, and the solving process only in that it uses as few resources as possible. Our interest in developing BART is to explore an inverted paradigm: we are deeply interested in the solving process and only the solution in that it is correct. The BART code is designed with a different end user in mind: the developer.

The ultimate goal of this inverted design is to allow the developer to insert their own process or subroutine into the solve, validate that it works against provided test problems, and compare its performance against other methods in an information-rich way. Unlike major production codes, the speed and efficiency we want is in modifying the solve, not the solve process itself. Those who develop the acceleration methods that would be implemented in a code like this are researchers who study the mathematics of the neutron transport equation. Therefore, the code must be designed to make sense to this audience. Many decisions made in designing the code may not be optimal from a computer science perspective, but will make things more clear to an academic developer making modifications. The development of the BART code itself therefore uniquely benefits from being developed by academics and not pure computer scientists. In this section, we will discuss some of the qualities and characteristics of the BART code that emphasize its focus on the developer end-user.

5.2.1 Class Design

In performing a careful division of the solve process and using individual classes to perform many of the individual steps, we seek to make modifying the routine straightforward. Larger production codes may use more procedural processes that are difficult to modify because it is a more efficient routine for completing the solve. The downside of the more efficient procedural code is that it is harder to modify. Implementing new acceleration methods or schemes often requires multiple modifications to different parts of the solve process, inserting new subroutines into the procedural code, or inserting logic trees. A key design goal of BART is that developers can identify the part of the solve process where their routine fits, identify the interface that performs that function, and add their customized class or classes. Doing so limits the amount of code that needs to be understood and modified.

We will illustrate this with an example. Consider a researcher who has developed a new method of calculating the k-eigenvalue for power-iteration (PI). They will naturally need to find the PI routine and determine where modifications will need to be made. In BART, outer iterations are defined as generally as possible, but with steps clearly named. The researcher will see that the BART code identifies that an eigenvalue iteration has a few basic steps,

1. Update the system in some way,

2. Perform outer iterations,

3. Check for convergence, and
4. Repeat if necessary.

The provided power iteration scheme is a specialization of this class that updates the fission source in step 1, and calculates the k-eigenvalue in step 3. The benefit to this careful consolidation of iteration procedures in a generalized base class means that the power-iteration scheme implementation requires less than ten lines of code. This greatly reduces the amount of existing code the researcher will need to read and understand to implement their new calculation.

In looking at the few lines of code that specialize the PI scheme,the researcher can easily identify the location where the k-eigenvalue is calculated,

```
// src/iteration/eigenvalue/power_iteration.cpp
...
convergence::Status EigenPowerIteration::CheckConvergence(
    system::System &system) {

  double k_effective_last = system.k_effective.value_or(0.0);
  system.k_effective =
      k_effective_updater_ptr_->CalculateK_Eigenvalue(system);

  return convergence_checker_ptr_->ConvergenceStatus(
      system.k_effective.value(), k_effective_last);
}
...
```

The power iteration class owns a class that has a single job, calculating the k-eigenvalue given a system. Therefore, the researcher merely needs to create a new class that calculates the k-eigenvalue using their new method. No further modification to the solving routine is required. Once the new method is implemented in this new class, the routine that constructs the classes responsible for the solving process must be updated to use this new method. To ease the modification of the code and make it clear to developers where and how formulations of the transport equation are solved, we use a framework model.

5.2.2 Framework Model

A major design feature of BART is "frameworks." A framework is a container that holds the constellation of objects required to solve the transport equation. A framework is not responsible for connecting all the classes, this is done by a framework builder, it merely executes the solve. For an unaccelerated solve, only one framework would be constructed to contain all the required classes to solve some specific formulation of the transport equation. A non-exhaustive list includes: cross-section data, meshes, finite-element domains, quadrature, and specific formulations of the transport equation. More complex solves may involve multiple frameworks, each able to independently solve a different formulation of the transport equation as needed.

The power of this framework model is realized when implementing acceleration methods that require solving a different formulation of the transport equation. For example, the nonlinear diffusion acceleration (NDA) method requires solving a drift-diffusion formulation with each iteration of the angular solve. Accomplishing this nested solve is easy with the framework model: we only need to create a post-iteration subroutine that contains a framework with the appropriate formulation, in this case drift-diffusion. We will refer to the framework solving the problem of interest as the main framework, and any other frameworks solving auxiliary problems as subroutine frameworks.

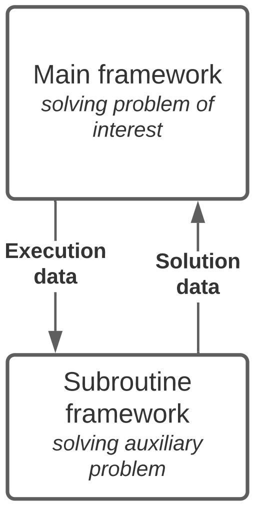


Figure 5.3: Main and subroutine frameworks.

As we see in Fig. 5.3, there are, in general, two data flows between frameworks. The main framework often provides execution data to the subroutine frameworks, which inform the solve of the auxiliary problem. An example of this in NDA is the angular solution data provided to the drift-diffusion auxiliary problem for calculation of the boundary term and drift-diffusion vector. This data path is established when the frameworks are constructed at the start of problem execution and is transparent to the solve process. The second data flow is solution data provided from a subroutine framework to the main solving process. This is handled by a subroutine located at a specific location in the solving process of the main framework. It will call on the subroutine framework to execute a solve, and then extract the solution to be used. In the example of two-grid (TG), the solution to the auxiliary problem provides an error update to the main framework solution in each solver step.

The process to define and construct frameworks is designed to specifically aid developer end-users. The construction of frameworks is controlled by a “framework builder” class that takes a recipe for a transport solve and instantiates and wires together all the needed classes to execute it. This recipe takes the form of a simple but comprehensive data structure that can either be generated from an input file or created at runtime by the code itself. To implement an acceleration method, users only need to tell the builder where to find and how to build their new class and specify using it in the recipe.

There is no limit to the number of frameworks that can be present, except for practical memory considerations. The framework construction process allows for the sharing of solver components, such as meshes and finite element domains, reducing the overall memory consideration.

5.2.3 Term Storage Consideration

Another design consideration that is explicitly included in BART that is not ideal from an optimization point of view is the storage of terms in transport equation formulations. Solving the transport equation in general requires either the explicit or implicit formulation of the left-hand side and right-hand side terms and then a linear solve. In the BART code, terms on the right-hand side and left-hand side can be stored separately. By default, the code provides the option of storing "fixed" terms that will not change iteration-to-iteration and "variable" terms that will be updated. The benefit to storing these terms separately is that their values can be easily collected and output. The process that would be used for this will be discussed in the following section.

5.2.4 Documentation

The final, but important, part of designing a code for a developer end-user is good documentation. BART uses Doxygen, an automated system for generating easily-navigable documentation from C++ files. The documentation is included in the source files themselves, and this program will extract and format the documentation into various formats, including HTML. A large benefit to using this system for a research code is built-in support for $\LaTeX$-like equations using MathJax. This is particularly important for describing the formulations for the transport equation. Overall, good documentation is important because it will reduce the time that developers spend determining how the code works. In addition, good documentation provides information about other parts of the code without needing to read the source files themselves.

5.3 Providing a Good Environment for Measuring Effectiveness

The next design goal of the BART code is to provide a good environment for measuring the effectiveness of acceleration methods. Ideally, we can easily collect good information about the efficiency of our acceleration method. An important step in generating good data is defining a base case and minimizing the changes between it and the accelerated case. As researchers driven by the scientific method, we seek to change only one thing: the implementation of the acceleration method. To do so, we need to be able to insert the novel method without disrupting the remainder of the solving process. This is a practical challenge with evaluating acceleration methods in existing codes, because codes often require logical cutouts to run subroutines and modifications that span multiple sections of the code. We seek to make BART a code that makes this easy and effective, providing a good environment for measuring the effectiveness of the method compared to a base case, or other acceleration methods.

One of the key design features in BART that achieves these goals is the use of pure abstract interfaces. An interface is a design pattern in which we define a class that doesn't actually execute any code, but acts as a template for what classes of its type *should* do. The rest of the code only interacts with the interface, and the actual internal workings of the code are a black box. To put it a different way, interfaces define *what* a class does, not *how* it should do it and the rest of the code only cares about the former. The implication of this design pattern is that we can swap out one part of the solving process with a modified version without changing how the rest of the code operates. This not only minimizes the changes needed to implement a new method, but isolates the changes from the rest of the solving process. This maximizes the quality of our comparisons to a base case.

As an example, in C++, a pure abstract function is designated as follows:

```cpp
class PureAbstractI {
  public:
    virtual auto AbstractFunction() -> void = 0;
}
```

The `virtual` identifier indicates that this function may be overridden by other classes. A virtual method table (VMT) will be instantiated with this class that will point to the address of the dynamically bound implementation of this function. The use of `= 0` indicates that this class provides no default implementation but that it *must* be overridden by an inherited class. All classes in BART are derived from a pure abstract class identified by a trailing `I` in their name.

As much as possible, classes interact with each other through the virtual methods in the interface, not implementation-specific methods. When classes are passed as dependencies or as arguments, they are passed as either references or pointers and then bound to a reference or pointer to their base interface. Strict adherence to this design principle means the code can implement a single interface many different ways, without any impact on the way the rest of the code operates. There are two major benefits to this design paradigm.

The first comes from good encapsulation of each step or part of the solve process. We have designed the BART code so that researchers in the transport field will recognize many of the classes in the solve routing and what they are expected to do. This will make it clear to researchers what part of the routine they should modify to implement their novel acceleration method. In most cases, we have tried to limit each class to performing a single calculation or mediating the interaction between multiple classes. This also helps with creating testable code, as will be discussed in a later section. In some cases, the division of classes is not ideal from a computer science perspective, but may improve the understanding for academic developers.

By using this interface design model, developers can be sure that their changes are encapsulated and minimal. This design paradigm is focused on accomplishing the first two major design goals. By minimizing the amount of code that needs to be modified to implement new method, we have created an environment that seeks to reduce the burden of implementing new methods. In addition, the interface paradigm creates a good environment for measuring the effectiveness of these new methods. By encapsulating the changes and eliminating modifications to other parts of the solve process, we seek to create a controlled environment where researchers can be sure that they can effectively measure the effectiveness of their acceleration routine as they will not be changing lots of pieces of code that could affect performance in ways that are unrelated to the method itself. The data collection to make these assessments is described in Section 5.4.

Finally, the use of interfaces improves the portability of code modifications. In most cases, implementing acceleration methods will not require modifying a majority of the solving process, merely creating a new version of a class. Therefore, if one developer would like to share their modified code with another, the potential for conflicts is greatly reduced. This streamlines sharing and validating of novel acceleration methods and will allow others to see if they are as efficient on the types of test problems in which they are interested.

5.4 Providing Instrumentation for Measuring Effectiveness

The third design goal of the BART code is to provide tools for measuring the effectiveness of acceleration methods. As discussed in Chp 4, total iterations until convergence is a good heuristic, but more data unlocks more insight. This improved data will help researchers determine if and how their acceleration methods are working in detail, driving future improvements and development. Capturing this data must be extensible; we cannot *a priori* know all the of the data users will want or where in the code it will come from. Finally, capturing different kinds of data must be easy, as this is a key feature in the code and should be as minimal a burden on the end-users as possible.

The initial question to be asked when dealing with instrumentation is what data needs to be extracted from the solving process. Acceleration methods are mathematically derived and often seek to improve a particular inefficiency in existing algorithms. Fixing these inefficiencies will, in most cases, cause the problem to converge in fewer iterations. In most cases, better data about how well the solve is working exists further upstream than iteration count and is within the solve itself; the form this data takes will be determined by the type of acceleration method used and how it is expected to operate.

As an example, the TG method described in Sec. 3.4 is specifically designed to converge the scattering-source term faster than normal GS iterations. Collecting data on the convergence of the source term would therefore provide a better understanding of how the method operates than observing the total iteration count. Finer data could potentially be collected as well, such as looking at the magnitude of the isotropic residual. What is important is that this data is very specific to the design and mathematical basis of the TG method. Some of the data that can speak to the efficiency of the method may help users analyze other methods, but some is specific to this scheme.

Therefore, we cannot know ahead of time what kind of data future users will need. This will be motivated by method development and needs that we cannot and should not anticipate. To this end, the ability to collect different types of data must be straightforward and easy to add. In addition, the code should not limit the types of data that can be collected. Finally, the data collection must be as transparent as possible, reducing the overhead for the solving process when that data is not being collected.

The instrumentation system is also designed to meet the first goal of BART, being a developer-centered code that makes implementing new methods easier. We recognize that part of developing and assesing a new method is the data collection process to determine effectiveness. Therefore, new types of instruments may be required, and developing them must be straightforward. As we will see, many common instruments and components have already been implemented in BART, reducing the coding burden. In addition, instrument components are designed to be simple and straightforward: each has a single responsibility that is contained in a single function call. The aim of this is to make their operation and use transparent and easy to develop.

We will use a motivating example to help understand the two main parts of the data collection system: ports and instruments. Consider a hypothetical acceleration method that uses a diffusion solve to accelerate our main angular solve. Let this method have a variable diffusion coefficient $\tilde{D}$ that is dependent on a parameter from the angular solve that changes iteration-to-iteration. As we assess this method, we may be interested in the value of $\tilde{D}$ as the solve progresses. This may give us insight into if the method is efficient and if it is working in the way we expect it to. To extract this value from the solve, we will first need to install a port that will make the data available, and then install an appropriate instrument to output the data. We will begin with a description of the port system.

We note that the port and instrument system rely heavily on the C++ concept of templating, which allows us to create generic code that is not specified for a particular data type. While this gives us and future developers the flexibility to create ports and instruments for any data types desired, it does require some more in depth knowledge of C++.

5.4.1 Ports

Most production codes are not designed with the intention of data collection. This is not a failing of the design, but a feature. These codes are designed to solve the transport equation as rapidly and efficiently as possible, not to collect data about the solve process. Therefore, much of the data method developers may be interested in may reside in difficult-to-find locations, and may not even be accessible without major modifications to the code. The easiest and most rudimentary way to collect data is to output the data to the standard output and collect it manually. This is clearly not an ideal circumstance, but data transfers to files that work properly in every part of a code can be both frustrating and difficult. In both cases, changes must be made to the source code near where the data is collected that may alter program flow, not to mention slowing the code down, and creates a much larger change than is necessary. An ideal system for data collection would not only be able to collect data from anywhere in the solve process, but also provide as minimal a change as possible to that process.

The first part of the data collection system in BART is the port system. The port system is designed to collect any type of data with minimal changes to the surrounding code. Ports can collect any type of data through use of the C++ template system. The port class is defined using a data type and a unique data name. Multiple ports can have the same data type, so names are used to differentiate them if needed. The `Port` class is designed to be inherited by any class that will provide data collection and a portion of the code is as follows:

```
// src/instrumentation/port.hpp
namespace bart::instrumentation {

template <typename DataType, typename DataName>
class Port {
 public:
  using InstrumentType = InstrumentI<DataType>;

  auto AddInstrument(std::shared_ptr<InstrumentType>) -> void;
  auto Expose(const DataType&) -> void;
  ...
}

} // namespace bart::instrumentation
```

This shows the two major functions provided by the `Port` class. The first is `AddInstrument`, which will “install” an instrument in the port. This instrument must be specified for the type of data that the port is designed to read, and the use of a smart pointer ensures that it can be shared among different ports if desired. The second major functionality provided by a port is `Expose`, which will provide or “expose” data to an instrument if installed. The use of a constant reference to the data ensures that the data is not changed or copied when exposing. The `Expose` function consists of a single check to see if the instrument pointer is valid, and then a call to its `Read` function. Determining if there is a valid instrument installed in the port must be done at runtime, so exposing data always carries with it the cost of checking if a pointer is valid. The code is kept as simple as possible to avoid unnecessary overhead and increase the chance that the compiler can optimize the call by inserting the code inline at the calling location. This meets our design goals of minimal overhead and disruption to read data, as well as making it extensible to any data type that the user may need.

The next design goal is to make it easy for a user to install and expose data using these ports. For our example, we would like to read out our floating point value of the variable diffusion coefficient $\tilde{D}$ from some arbitrary iteration class. To do so, we will create a port to extract it. We will give this port a unique name `DiffusionCoefPort`, and the data the name `DiffusionCoef`; names that can be as specific or general as we need them to be. To extract the data we will first define a template specialization of the port class and inherit it. This would be done in the header for the class.

```
// iteration.hpp
#include <src/instrumentation/port.hpp>

namespace data_ports {
  using Port = bart::instrumentation::Port;
  struct DiffusionCoef;
  using DiffusionCoefPort = Port<double, DiffusionCoef>;
}

class Iteration : public data_ports::DiffusionCoefPort { ... };
```

Now that the class `Iteration` inherits from the port class, we can expose the temperature data simply by calling `Expose`:

```
auto Iteration::Solve() -> void {
  ...
  data_ports::TemperaturePort::Expose(diffusion_coef);
  ...
}
```

The specialization using a unique name, `DiffusionCoef`, allows us to add multiple ports to the class that all read the same thing.

Adding a port to our `Iteration` class required a minimal number of lines of code, less than ten. This meets our design goal of making adding new data extraction methods cause minimal changes to the code. It is also very clear within the implementation of the `Solve` function that data is being exposed, and when. The user also does not need to worry about *not* outputting data when it is not desired. They merely do not install an instrument into the port.

To install an instrument, there is a provided free function.

```
// src/instrumentation/port.hpp
template <typename PortType, typename T>
auto GetPort(T& to_expose) -> PortType&;
```

Use of this function requires the name of the port to be retrieved, and the underlying port will be returned, allowing us to call `AddInstrument`.

```
Iteration iteration;
auto instrument_ptr = std::make_shared<DoubleInstrument>();
bart::instrumentation::GetPort<data_ports::DiffusionCoefPort>(iteration)
    .AddInstrument(instrument_ptr);
```

This is a compact way of installing the instrument that minimizes the total code required, and hides much of the implementation and template specialization that isn't necessary for the user to understand.

A class can have as many ports as it needs or the user desires. The use of unique identifiers for each port means that multiple ports can read the same types of values, or even the same values themselves. The generality of the port class means that it can be used by any part of the BART code. Now that we have the ability to expose data, we need instruments that are designed to read that data.

5.4.2 Instruments

With ports installed to access the data and information we need to assess the solving process, we need instruments to process that data. These instruments must be able to read any kind of data, and may need to convert that data into other formats depending on where that information needs to go. The simplest instruments may read data, convert it into a string and place it into the standard output or a file. More complex instruments may perform various calculations on the data before outputting. Our instrument framework must therefore be versatile enough to handle both situations. Unlike ports, the inner workings of which are not of much use to a user, the inner workings of instruments will need to be understood by the developer end user. Therefore, constructing complex instruments must be as easy as possible.

The interface for instruments is straightforward, specifying an input data type, and the `Read` function that is accessed by the ports.

```
// src/instrumentation/instrument_i.h
namespace bart::instrumentation {

template <typename InputType>
class InstrumentI {
 public:
  virtual auto Read(const InputType&) -> void = 0;
};

} // namespace bart::instrumentation
```

We see that the `InputType` specification must match the `DataType` of the port where this instrument is installed. The most basic type of instrument will take this data and send it elsewhere. We call the class of objects that send the data out of the instrument outstreams.

5.4.2.1 Outstreams

The basic outstream interface also only has a single class function,

```
// src/instrumentation/outstream/outstream_i.h
namespace bart::instrumentation::outstream {

template <typename DataType>
class OutstreamI {
 public:
  virtual auto Output(const DataType&) -> OutstreamI& = 0;
};

} // namespace bart::instrumentation::outstream
```

The outstream also has a template parameter `DataType` to identify the type of data that it is expected to output. Again, the data is passed as a constant reference, so no data is copied or changed in the process. There are many possible outstreams that can be implemented or conceived of, so only relying on the interface enables users to define their own if needed. One of the most common data types that an outstream takes is strings. Therefore, BART has available implemented outstreams for sending strings to a standard `std::ostream` that can support either the standard output using `std::cout` or to a file using a `std::fstream` object. To support MPI, there is also an outstream that sends strings to a conditional output that can be configured to only display for the rank 0 processor. Another useful task is sending `dealii::Vector` objects to a file to be read by Paraview, so BART has outstreams implemented for that task as well.

We now have enough building blocks to describe one of the most simple instruments in BART. This instrument will read a string from a port configured to expose a string, and send it to an oustream. This is used in various parts of BART to provide status updates to the standard output. In this case, the instrument used is of type `BasicInstrument`, an instrument with the same input and output type. For this class, the `Read` method is straightforward:

```
// src/instrumentation/basic_instrument.h
namespace bart::instrumentation {

template <typename InputType
class BasicInstrument : public InstrumentI<InputType> {
 public:
  using OutstreamType = typename outstream::OutstreamI<InputType>;
  explicit BasicInstrument(std::unique_ptr<OutstreamType>);
  auto Read(const InputType &input) -> void override {
    outstream_ptr_->Output(input); }
  ...
}

} // namespace bart::instrumentation
```

It passes the data from input to the outstream operator. We see that the implementation requires that the outstream accepts the same data type as the instrument reads.

There are a number of benefits to using the port-instrument pattern with classes instead of just having the classes output directly to the standard output. First, we can easily control if a class outputs status by leaving the port uninstrumented, instead of requiring a boolean that needs to be checked every time the class would output status. Second, we can have multiple status ports that may expose different information, and choose which we instrument at any given time. Last, if we change how we output to the standard output, for example by using a different formatting library, we only need to change the instrument and all classes that use it will be affected.

5.4.2.2 Converters

Exposing, reading, and outputting strings is the most straightforward means of reading data within BART. Much of the time, however, the data users are interested in when assessing acceleration methods will not be in the form of strings. Although we could have classes convert these values and expose them as strings, in many cases the user may need to alter or modify the data before outputting to a file or the standard output. In fact, the user may never convert the values to a string at all. Changing data read by an instrument into data to be output by an outstream is handled by converters. Like the other instrumentation classes, converters have one main member function, the aptly named `Convert`.

```
// src/instrumentation/converter/converter_i.h
namespace bart::instrumentation::converter {

template <typename InputType, typename OutputType>
class ConverterI {
 public:
  virtual auto Convert(const InputType& input) const -> OutputType = 0;
};


} // namespace bart::instrumentation::converter
```

Unlike the other parts of the instrument, we see that a converter is necessarily specified by two types, an input and an output type. The input will match the data read by the instrument (and exposed by the port) and the output will match the data output by the outstream. Most commonly, we'd like to process or read the data from various parts of the solve, so many converters will convert various data types into strings to be output into files or the standard output. Naturally, BART has implementations for many data types to be converted to strings.

We also anticipate users will require more complex converters, including multistage converters. An example of a multistage converter could be an instrument that reads a vector, calculates the norm, and then converts the norm to a string to be read in the standard output. This would involve two converters, one to convert a vector into a numerical value, and a second to convert the numerical value into a string. Accomplishing this must be easy for users, who we want to be able to focus on method development, and not the intricacies of instrumenting their solve.

To accomplish this, BART has a multi-converter class that acts as a mediator between two converters.

```
// src/instrumentation/converter/multi_converter.hpp
namespace bart::instrumentation::converter {

template <typename InputType, typename IntermediateType,
          typename OutputType>
class MultiConverter : public ConverterI<InputType, OutputType> {
  public:
   using FirstStageConverter = ConverterI<InputType, IntermediateType>;
   using SecondStageConverter = ConverterI<IntermediateType, OutputType>;
   ...
   MultiConverter(std::unique_ptr<FirstStageConverter>
                  std::unique_ptr<SecondStageConverter>);

   OutputType Convert(const InputType &input) const override {
     return second_stage_converter_ptr_->Convert(
         first_stage_converter_ptr_->Convert(input));
  }
}

} // namespace bart::instrumentation::converter
```

As we see, our multi-converter owns two different converters, linked by an `IntermediateType`. The multi-converter itself is a converter, whose `Convert` method nests the two internal converters. By this logic, it should be possible to nest many multi-converters to create an endless chain of converters as needed. Accomplishing this nesting can be tricky and require a large amount of code.

To meet our design goal of making easy implementation of instrumentation, we need an easy way to string converters together in a way that users will not need to worry about the workings of the complex `MultiConverter` class. To accomplish this, we overload the binary plus sign operator.

```
// src/instrumentation/converter/multi_converter.hpp
template <typename InputType, typename IntermediateType,
          typename OutputType>
inline std::unique_ptr<ConverterI<InputType, OutputType>>
operator+(
    std::unique_ptr<ConverterI<InputType, IntermediateType>> lhs,
    std::unique_ptr<ConverterI<IntermediateType, OutputType>> rhs) {
  return std::make_unique<
      MultiConverter<InputType, IntermediateType, OutputType>>(
          std::move(lhs), std::move(rhs));
}
```

While this code looks complex, the C++17[33] template deduction actually makes the code quite easy to use.

If the user has two converters that they would like to convert into a single multi-converter, they use the binary plus sign. For example, if the user is combining a converter that changes a vector into the value of its L_1 norm, and one that will convert this value into a string, the code could look like:

```
auto vector_to_double = std::make_unique<VectorToL1NormConverter>();
auto double_to_string = std::make_unique<DoubleToStringConverter>();
auto vector_l1_to_string = std::move(vector_to_double) +
                           std::move(double_to_string);
```

The template deduction will not only return the correct multi-converter, but will cause a compilation error if the user tries to add two incompatible converters. This not only makes it easier to create complex instruments, but favors compilation errors over runtime errors, which helps with development and avoid hard-to-find bugs. Assuming all the converters are compatible, users can string as many together as they'd like using the binary plus operator, without ever needing to know that the `MultiConverter` class even exists.

5.4.2.3 Full Instruments

The basic instrument described earlier in this section read a specified input type and passed it to an outstream specified with the same type. A full instrument has two parts, a converter and an outstream, and two types, an input type and an output type. As can be deduced from the pattern, a full instrument's `Read` method nests the conversion and output calls of the converter and outstream it owns.

```
// src/instrumentation/instrument.h
namespace bart::instrumentation {

template <typename InputType, typename OutputType>
class Instrument : public InstrumentI<InputType> {
 public:
  using ConverterType = converter::ConverterI<InputType, OutputType>;
  using OutstreamType = outstream::OutstreamI<OutputType>;
  ...
  auto Read(const InputType &input) -> void override {
     outstream_ptr_->Output(converter_ptr_->Convert(input));
  }

} // namespace bart::instrumentation
```

Finally, BART also provides an `InstrumentArray` class that acts as a storage for multiple instruments that read a single type. The array is an instrument itself, and when installed will call read on all the instruments that it owns, allowing multiple instruments to process the same data.

5.4.2.4 Complex Instrument Systems

Instruments can be expanded to include functionality that is not merely conversion between data types and output. Converters are designed as merely black boxes between one data type and another, and can therefore do as many internal complex operations as desired. One example of this that is included in the BART code is a discrete Fourier transform (DFT) instrument. This multistage instrument will perform the DFT of a provided vector, "converting" an input vector into a complex output. The implementation in BART is designed to calculate the DFT of the error and is therefore made up of multiple stages. Each time the scalar flux is exposed to the instrument it,

- subtracts the scalar flux from a known solution provided *a priori*[1] to calculate the error,
- performs the DFT of the error,
- converts the complex output of the DFT into a string, and
- outputs the string, usually to a file.

This process requires a converter for each step, and uses the multiconverter structure described earlier.

The instrumentation and port system is complex due to the use of the C++ templating system, but gains the flexibility to read any data type desired by future users. Many classes specialized for common data types are already implemented in BART, enabling users to extract a large amount of data without ever designing a new instrument. In this system, we meet the goal of providing the tools needed to assess the effectiveness of acceleration methods. Our instrumentation and port system is internally complex but represents minimal intrusion into the solve process and the minimal amount of work needed to instrument for new data.

5.5 Testing and Reproducibility

The fourth design goal of BART is to provide a tested, portable, and reliable environment. Much of the design of BART approaches this by using modern coding techniques and tools. When developing a code in an academic environment, these are qualities that can often be overlooked. This is not malicious, but a practical consideration when resources and time are limited. In this section we will describe how the design of BART makes unit testing possible and discuss some of the other tools used to meet this goal.

[1]BART provides an option for performing the DFT of the error and executes this by automatically running the simulation twice and using the solution from the first run as the known solution.

5.5.1 Testing

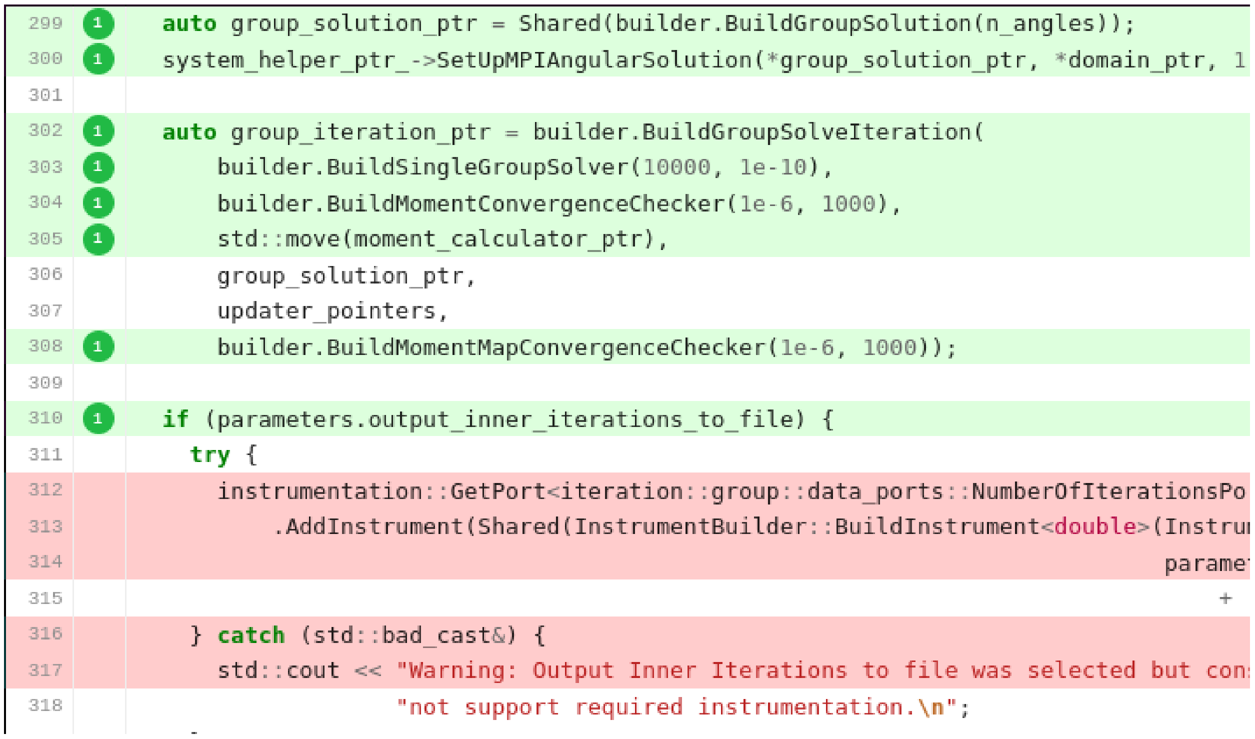
Testing is an extremely important part of code development. In general, we will focus on two types of testing: unit testing and integration testing. Unit testing is code written to verify that a specific portion of our source code operates properly in isolation. In contrast, integration testing verifies that different portions of the code work together properly. When combined, a fully developed test suite should verify the operation of each small segment of code as well as overall operation. There are two major hurdles that we will discuss to writing tested code: it can be difficult and time consuming to cover all parts of a code with tests, and the code must be written so that it is testable. Despite these hurdles, the benefits of testing cannot be overstated. Testing can immediately catch bugs and errors that would cost many hours to identify and rectify. Even more importantly, testing can catch issues with the code that do not result in errors at runtime, but result in an incorrect solution. Finally, good testing liberates developers to refactor and change sections of the code, confident that the code still works properly.

5.5.1.1 Writing Tests

For most developers is it not feasible or desirable for researchers to write their own testing environment. One could design their own framework for testing entirely from scratch, but much of the work has already been done by computer scientists. There are many testing libraries out there that provide frameworks for testing code, the one that BART uses is the GoogleTest framework [34]. The GoogleTest framework provides a large number of macros that make testing and reporting the results of tests straightforward, making it ideal for developing an academic code with time and resource constraints.

We want to meet the goal of providing tested code for end-users, but also understand that academic code development has constraints. To this end, we employed a philosophy of testing critical functions first, and then if time allows, testing other functions. As much as possible, the BART code attempts to limit each class to having a single responsibility. When developing tests, we focus on testing that responsibility first, before testing other supporting parts of the class. A fully tested class may require hundreds of lines of code to test every member function and all edge case. The less critical parts to be tested may include functions to get or set member variables or access dependencies. When time allows, it is imperative that we implement those tests, as the time spent writing the tests may prevent seeking hard-to-find bugs in the future.

Identifying portions of the code that are not tested can be simplified by using test coverage. Various coverage systems provide easy-to-use outputs that indicate what portions of the code were run during testing. At the time of writing the BART code uses a website called Codecov (`codecov.io`) to display the coverage; an example can be seen in Fig. 5.4. Code coverage is an excellent tool for finding sections of code that are not executed by tests, but does not verify that the tests are actually testing the code. The use of coverage needs to be coupled with an understanding of how classes need to be tested to ensure testing is



```
auto group_solution_ptr = Shared(builder.BuildGroupSolution(n_angles));
system_helper_ptr_->SetUpMPIAngularSolution(*group_solution_ptr, *domain_ptr, 1

auto group_iteration_ptr = builder.BuildGroupSolveIteration(
    builder.BuildSingleGroupSolver(10000, 1e-10),
    builder.BuildMomentConvergenceChecker(1e-6, 1000),
    std::move(moment_calculator_ptr),
    group_solution_ptr,
    updater_pointers,
    builder.BuildMomentMapConvergenceChecker(1e-6, 1000));

if (parameters.output_inner_iterations_to_file) {
  try {
    instrumentation::GetPort<iteration::group::data_ports::NumberOfIterationsPo
        .AddInstrument(Shared(InstrumentBuilder::BuildInstrument<double>(Instru
                                                                         parame
                                                                             +
  } catch (std::bad_cast&) {
    std::cout << "Warning: Output Inner Iterations to file was selected but con
                 "not support required instrumentation.\n";
```

Figure 5.4: Coverage output on the Codecov website; the green indicates lines that are accessed by tests, and the red lines that are not.

comprehensive and useful. Where coverage is instrumental is identifying decision trees or other parts of code that have not been accessed, especially if the tester thinks that they are covered.

Providing a fully tested code for developing and assessing acceleration methods is an important goal. Users can feel confident that they have built the code correctly and that it is operating the way we expect it to. Importantly, users may not have the time to write fully tested modifications to BART. In these cases, the vast majority of the solve, all the parts not modified by the user will still be covered by tests. Not only does this provide a level of security that their code is operating properly, even if not tested, but that they haven't broken other parts of the solve with their modifications. To this end, the BART code has set a standard of 95% code coverage, to ensure as much is covered by tests as possible. Much of the non-covered portions of the code are not critical parts of the solving process, but the parts that assemble the code for execution. It is vital that we only accept parts of the code to be uncovered by tests if their incorrect operation will be caught by the user at runtime.

5.5.1.2 Writing Testable Code

A part of writing tests that is often overlooked is that code needs to be written so that it can be tested. The BART code follows a few guidelines to maximize the testability of code:

1. every class should have one responsibility,
2. that responsibility should be either doing logic or instantiating other classes, and
3. dependencies should be provided by classes, not instantiated by them.

We will discuss each of these and the roll they play in making the BART code base testable.

The first guideline is known as the single-responsibility principle, which focuses development of a class around performing a single task. This principle has a number of benefits and downsides when developing academic code. The first issue is identifying the difference between what a responsibility means to a computational scientist when compared to a computer scientist. When approaching the idea of responsibilities from computational science, we often divide discrete steps in the execution process as responsibilities. For example, to a computational scientist, computing the discrete Fourier transform of a vector may seem like a single responsibility. To a computer scientist this step has many responsibilities that should be executed by independent classes.

When developing a code like BART with the computational scientist end-user in mind, we need to find a balance between these two positions. From a computer science perspective, breaking up a complex operation into smaller classes with smaller responsibilities makes testing much more powerful: if there is an issue with any single step, testing will identify it. Testing the complex operation overall will only reveal that the overall process is broken. The downside to breaking complex operations down into small classes with single jobs is that it causes the code base to grow dramatically. The issue is then that it may take longer for the computational scientist to grasp how the pieces fit together. If our goal is to focus on a developer end-user, we need the code base to not just be well designed and testable, but comprehensible without a large amount of work. With BART we have attempted to find a balance between the single-responsibility principle and our responsibility to have code that is not overly complex.

The second guideline divides the responsibilities of classes into two: those that perform logic and those that instantiate other classes. This division is very important because it separates out two types of fundamental code: code that is unit tested and code that is integration tested. If a class is prevented from instantiating any other classes, it can be tested alone and in isolation without interacting with any other real classes. Testing with dependencies or mediating interactions between two classes can be accomplished using mocks, discussed in the next section. If a class instantiates other classes, it can no longer be unit tested and requires integration testing. By keeping this divide, we maximize the amount of code that can be unit tested, which is both easier to accomplish and makes isolating bugs easier. In contrast, an issue identified by integration testing may indicate an issue with either the class being tested, the interaction it has with another class, or the operation of the other class.

The third guideline dictates how a very large portion of the BART code base is designed: dependencies should be provided to classes, not instantiated by them. This is a design paradigm called dependency injection. When instantiating classes, we do not allow that class to create its own dependencies, but rather "inject" the dependencies into the constructor. In BART, when dependencies are injected, the constructor only specifies an interface that it requires. The interface is a purely virtual class that only identifies how the rest of the code interacts with classes that derive from it, not how they should do so.

For example, if we design a `Foo` class that has a dependency of type `Bar`, we will first design the interface for classes that can act as a `Bar`.

```
class BarI {
  public:
    ~BarI() = default;
    virtual auto GetValue() const -> double = 0;
  }
```

Here, we define that any class that can act as a `Bar` must implement a constant function `GetValue()` that returns a double. Our `Foo` class now takes a pointer in its constructor.

```
class Foo {
  public:
    Foo(std::shared_ptr<BarI> bar_dependency_ptr);
    ...
}
```

The injection of the dependency must either be a pointer or a reference because the code does not provide a class of type `BarI` but some class that derives from it. In the BART code, dependencies are always passed using smart pointers that are shared pointers (`std::shared_ptr`) when the dependency is shared by more than one class or unique pointers (`std::unique_ptr`) otherwise.

We now have a very powerful tool at our disposal when trying to test the `Foo` class. It requires a dependency, but doesn't specify any particular concrete class that can be instantiated, only the abstract interface. We can therefore create a fake or "mock" class that derives from `BarI` that we control within our tests. While testing, we can make this mock class return whatever values we'd like from `GetValue()` without needing to define any underlying code. With this in mind, we can fully test the `Foo` class without ever writing a concrete `Bar` class and can test edge cases that a concrete class may never generate. A mocking framework is provided by GoogleTest called GoogleMock. This framework is used extensively in testing BART to unit test a majority of the classes where possible.

The only part of the BART code where classes often instantiate their own dependencies are those that wrap portions of `Deal.ii`. For these classes, we will assume that the underlying `Deal.ii` classes operate properly. This is no different than classes instantiating objects from the standard C++library, which we assume will operate as expected.

5.5.2 Reproducibility

We consider reproducibility to be the ability for others to reproduce the results of our work. Not only should people be able to find, compile, and operate the BART code, but it should be easy to do so. To achieve this goal, we have used many modern tools when developing the BART code. These tools provide automated systems to ensure BART compiles without error, that the code meets our code coverage requirements, and that we provide an environment for others to use the code with a low barrier to entry.

First, to verify that the BART code will build, we use a continuous integration system. Every time a new version is pushed to the code repository, this system downloads and builds BART. This is key to verifying that we are not introducing changes into the code that will break the compilation process. Second, our continuous integration runs all the tests and uploads the coverage metrics to our code coverage tool, as discussed in the previous section. Finally, we provide a virtual environment where BART can be built with all dependencies already installed. Our continuous integration system also uses this container, verifying that it works as expected. We also maintain a repository containing the steps used to build this virtual environment, which acts as a step-by-step guide to setting up a system to build and use BART.

These tools are vital to others using and validating our code. Through them we not only verify that the BART code is operating and testing properly, but provide transparency and accessibility.

5.6 Conclusion

In this chapter we presented the major goals of the BART code. These goals support a vision of the code as a tool for researcher developers to easily implement and assess novel acceleration methods in a controlled, tested environment. The design of the BART code itself is done with the researcher developer end-user in mind, making implementation and sharing of code modifications easy. Once modified, the ports and instrumentation systems provide the ability to collect and ultimately analyze the data we need to accurately assess the effectiveness of acceleration methods. In the next chapter, we will discuss how we leveraged these design features to implement and assess a novel combination of acceleration methods.

Chapter 6

Using BART to Assess Acceleration Methods

In the previous chapters we discussed the discretized transport equation, the iterative methods used to solve it, and the acceleration schemes that improve the convergence of these methods. We also covered the practical challenges with implementing and assessing acceleration methods in existing codes that motivated the development of the Bay Area Radiation Transport (BART) code. In this chapter, we will discuss the implementation details of the two presented acceleration methods, two-grid (TG) and nonlinear diffusion acceleration (NDA), in the BART code. For the TG method we will test effectiveness in all dimensions using a test problem with a large amount of upscattering to leverage the benefits of the method. The NDA method will be used to accelerate an analytic benchmark in one- and three-dimensions with a large amount of within-group scattering.

We will then present the data collected in those tests using the tools included in BART to assess and validate these methods. We will observe that the TG method improves the convergence of the diffusion formulation in all dimensions, greatly reducing the total Gauss-Seidel (GS) outer iterations required. The NDA method will accelerate the convergence of the self-adjoint angular-flux (SAAF) formulation in analytic benchmarks with a large amount of within group scattering.

6.1 Two-grid Acceleration in BART

The TG method, as described in Sec. 3.4, uses a diffusion approximation to create an additive correction to the scalar flux following each GS outer iteration. This additive correction should improve convergence properties of the iteration when there is a large amount of upscattering. We will first discuss the implementation of TG in the BART code. Following this, we will describe the test problem used to stress the GS iterative process to assess the effectiveness of the TG method.

6.1.1 Implementation Details

Implementation of the method uses two frameworks, one for solving a scalar form of the transport equation and the second for solving the approximated error diffusion equation to calculate the correction. These frameworks and data flows are shown in Fig. 6.1. The two-grid framework is contained in a subroutine that is installed in the outer-iteration and is run after all groups have been solved, but before checking for all-group convergence. This subroutine then uses the scalar fluxes to calculate the isotropic residual. The isotropic residual is material dependent, and consequently is discontinuous on material boundaries. Therefore, the subroutine first iterates over all cells in the domain and calculates the local cell contribution to the isotropic residual for each local degree of freedom. This local contribution is then inserted this into a domain-wide vector using a local-to-global degree of freedom mapping. The contribution from each group is summed for use as the right-hand-side in the auxiliary framework's solve of the two-grid diffusion equation.

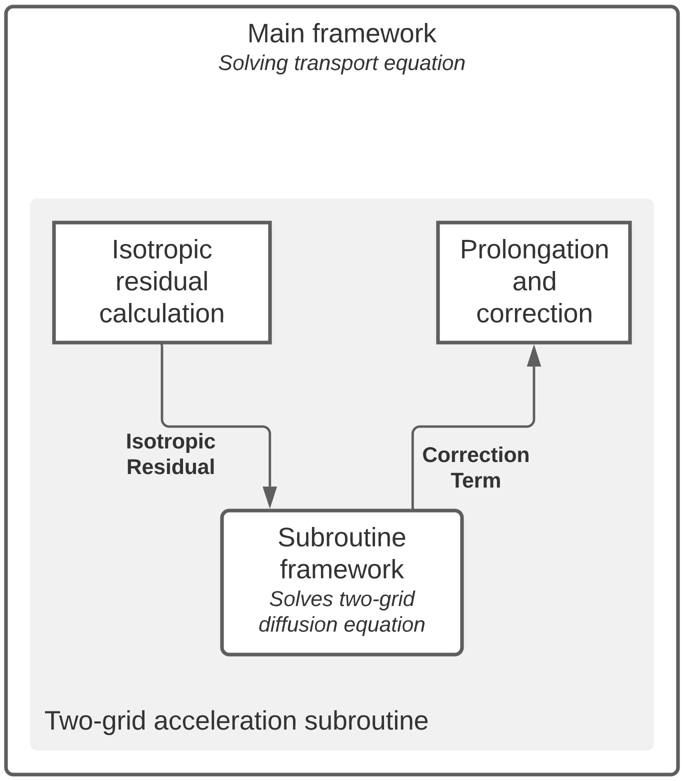

Main framework
Solving transport equation
Isotropic residual calculation
Prolongation and correction
Isotropic Residual
Correction Term
Subroutine framework
Solves two-grid diffusion equation
Two-grid acceleration subroutine

Figure 6.1: Two-grid acceleration method frameworks and data flows.

The two-grid diffusion equation is a specialization of the diffusion equation that overrides the source terms on the right-hand-side with zero and instead applies the isotropic residual as the source. The solution vector is then the error in the isotropic component of the solution instead of flux. The solving process, however, is identical. Both frameworks share a solving domain to be sure that the error solved for in the subroutine framework can be more easily applied to the main framework solution. This is not strictly necessary, but prevents the need for determining a degree-of-freedom index mapping from the auxiliary framework domain to the main framework domain. It also reduces the overall memory burden by only needing to store the domain and mesh a single time.

All the required steps of the two-grid acceleration are internal to the subroutine or completed before the solving process begins. Therefore, the difference between an unaccelerated base case and a two-grid accelerated case is minimized and isolated to the subroutine.

The energy-shape function is determined by solving the eigenvalue problem

$$\left[\mathbf{\Sigma}_t - \mathbf{\Sigma}_{s0}^L - \mathbf{\Sigma}_{s0}^D\right]^{-1} \mathbf{\Sigma}_{s0}^U \vec{\xi} = \lambda \vec{\xi}\,,$$

where $\mathbf{\Sigma}_t \in \mathbb{R}^{G\times G}$ is a diagonal matrix with the total cross-sections and $\mathbf{\Sigma}_{s0} \in \mathbb{R}^{G\times G}$ is the zeroth-moments of the scattering cross-sections divided into strictly upper (U), diagonal (D), and strictly lower (L) matrices. These energy-shape functions are material dependent, and therefore need to be calculated for all materials in the problem. As the cross-sections will not change during the solve, the energy-shape function is calculated prior to the solve and then stored in the prolongation portion of the two-grid subroutine.

The energy-shape function is discontinuous at material boundaries, which causes the isotropic residual to be discontinuous at the boundaries and complicates application of the correction in these locations. On material boundaries, the BART code uses an average of the energy-shape function for surrounding materials when applying the error. Consider a degree of freedom shared by two cells that are of different materials with energy-shape functions ξ_1 and ξ_2. When applying the correction to the group flux at this degree of freedom, we would use the following formulation,

$$\phi_g^{(k+1)} = \left(\frac{\xi_1 + \xi_2}{2}\right) \varepsilon^{(k+1/2)} + \phi_g^{(k+1/2)}\,.$$

Identification of material boundaries is not required; when a degree of freedom is interior to a single material the average reduces to the material's energy-shape function. This average value only needs to be calculated once prior to solving for each degree of freedom and is then stored and retrieved when applying the error.

6.1.2 Instrumentation for Assessment

To assess the effectiveness and implementation of the TG scheme, we need to collect good data. We expect that the TG scheme will cause the scattering source term to converge faster in the outer GS iterations for a diffusion-based formulation. We will therefore choose the unaccelerated diffusion equation for the base case and use the following ports to collect the data to make our assessment,

- `iteration::group::data_ports::NumberOfIterationsPort`: outputs the number of iterations required for the GS iteration to converge,
- `iteration::group::data_ports::ScatteringSourcePort`: outputs the scattering source term for the domain after each GS iteration and two-grid subroutine execution.

The number of GS iterations needed to converge is a good initial downstream indication of acceleration. We expect the inversion of the diffusion operator in the error equation to require less work than the standard diffusion equation due to reducing the problem to a single energy group. Examining the scattering source term will give us information about the means of acceleration; we can check if the method is truly speeding up convergence of the scattering source term as expected. Note that we do not expect the number of inner iterations per GS iteration nor the number of eigenvalue iterations to change since the TG method is not affecting either solve.

6.1.3 Test Problems

To test the TG method, we will use a two material test problem in one-, two-, and three-dimensions with reflective boundary conditions. In each case we will use fictional seven-group cross-sections described below that are designed to stress the GS iterative scheme by including a large amount of upscattering among the lower energy groups. The group scalar flux convergence criteria is

$$\frac{\left|\phi_g^{(k+1)} - \phi_g^{(k)}\right|_\infty}{\left|\phi_g^{(k+1)}\right|_\infty} < 1 \times 10^{-6} \ ,$$

We use the same the convergence threshold in source iteration (SI) within each group. The convergence criterion for the k-eigenvalue is

$$\frac{\left|k^{(k+1)} - k^{(k)}\right|}{k^{(k+1)}} < 1 \times 10^{-6} \ .$$

In each test we will check that the accelerated solution is consistent with the unaccelerated solution. We will do this by calculating an error relative to the base case for each group where the relative error is,

$$\text{Rel. Error} = \frac{|\vec{\phi} - \vec{\phi}_{\text{tg}}|}{\vec{\phi}}, \tag{6.1}$$

where $\vec{\phi}$ and $\vec{\phi}_{\text{tg}}$ are the scalar fluxes at each spatial degree of freedom in the base case and accelerated cases, respectively. The values of k are also compared. We will now describe the fictional cross-sections developed for these test cases.

Cross-sections for assessment We will have the chance to see the most benefit from the TG method if we provide a problem that has a large amount of upscattering. To this end, we have created two fictional materials with a very large amount of upscattering to stress the base case and the TG method. These materials have seven energy groups and neutrons are only generated in the top three energy groups as seen in Tab. 6.1. The total-

Table 6.1: χ_g for the fake material used to assess the two-grid acceleration scheme.

Group	0	1	2	3	4	5	6	7
χ_g	0.5	0.25	0.25	0.0	0.0	0.0	0.0	0.0

cross section Σ_t^g for all energy groups is set to unity, and the scattering cross-sections are set to maximize upscattering among the thermal groups. The scattering matrix and total scatter cross sections are given by the following, where empty entries are zero,

$$\Sigma_s^{g'\to g} = \begin{bmatrix} 0.499 & & & & & & \\ 0.250 & 0.499 & & & & & \\ 0.250 & 0.499 & 0.459 & & & & \\ & & 0.240 & 0.300 & 0.290 & 0.250 & 0.200 \\ & & 0.100 & 0.230 & 0.300 & 0.240 & 0.200 \\ & & 0.100 & 0.230 & 0.200 & 0.300 & 0.200 \\ & & 0.100 & 0.230 & 0.200 & 0.200 & 0.300 \end{bmatrix}, \qquad \Sigma_s = \begin{bmatrix} 0.999 \\ 0.998 \\ 0.999 \\ 0.990 \\ 0.990 \\ 0.990 \\ 0.900 \end{bmatrix},$$

and the entry at row i, column j gives the scattering contribution into group i from group j. The sum of each column therefore gives the total scattering cross-section for that group. Due to setting the total cross-section to unity, this value also represents the scattering ratio for this group. The remaining portion of the cross-section is the absorption cross-section, of which we set 95% to fission, multiplied by a chosen value of ν of 2,

$$\nu\Sigma_f = 2 \times 0.95 \times \left(\vec{1} - \Sigma_s\right) .$$

This leads to a system dominated by down-scattering in the upper three energy groups where all neutrons are born from fission. Scattering is coupled through the third energy groups to the lower four groups. These groups are dominated by scattering, with a small amount of fission. We will also use a highly scattering reflector that has an identical scattering and total cross-section, but no fission. Both of these materials will be used in the following test problems.

6.1.4 One-Dimensional Test Problem

The one-dimensional test problem includes a central fissionable material region that is surrounded by reflector, with reflective boundary conditions. The domain is triangulated into

100 segments of uniform size, resulting in 101 spatial degrees of freedom. The solution and problem layout for the highest energy group and lowest energy group are shown in Figs. 6.2 and 6.3.

Box plots showing the relative error as defined in Eq. (6.1) in each energy group are shown in Fig. 6.4. These boxes show a statistical analysis using the data set of relative error across all spatial degrees of freedom. The box extends from the median of the upper half of the data set values to the media of the lower half of values, with the central green line indicating the median of the set. Whiskers indicate the minimum and maximum values, and outlier points show values that fall outside 1.5 times the size of the box. We observe that the relative error in the higher energy groups is on the order of 1×10^{-6} and the lower energy groups is on the order of 1×10^{-5}. This low relative error shows that the TG method is returning a solution consistent with the unaccelerated case. The higher error in the lower energy groups – those with upscattering – is expected, as this is the region impacted by the TG acceleration method. The k-eigenvalue calculated by the accelerated solve is consistent with the unaccelerated case, with a relative error of 1.59×10^{-5}.

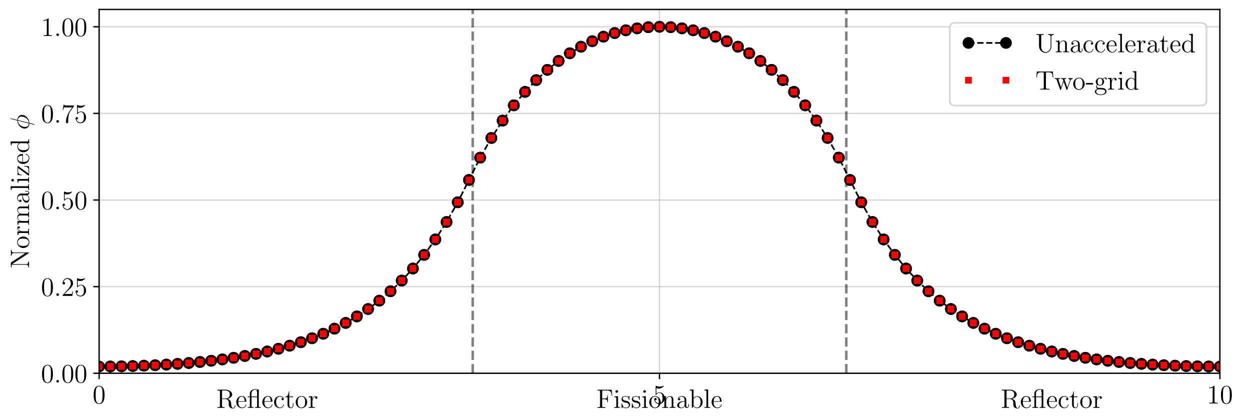


Figure 6.2: Normalized highest energy group scalar flux for the high scattering one dimensional two-grid test problem.

6.1.4.1 Iterations Assessment

Having established from the low relative error that the solution of the two-grid acceleration is consistent with the base case, we will now examine the efficiency of the method. Table 6.2 summarizes the iterations required for the unaccelerated and TG cases. With both methods, there were a total of five eigenvalue iterations required, but the two-grid method required significantly fewer GS outer iterations. Consequently, the TG method requires significantly fewer inner iterations, each of which requires a linear solve and therefore an inversion of the transport operator. Note that the number of inner iterations per outer iteration remains constant, as expected, and the reduction in iteration count comes from accelerating GS–the goal of the TG method.

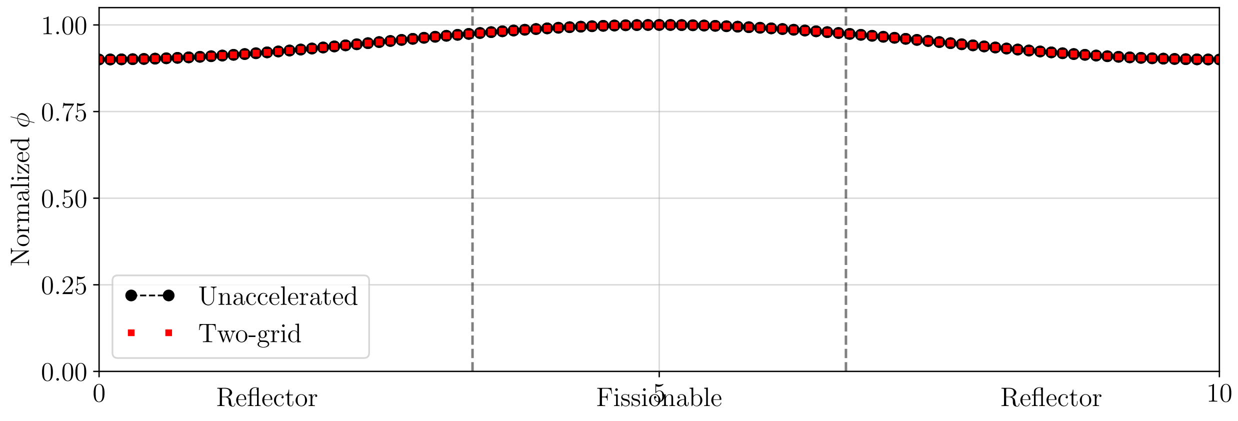


Figure 6.3: Normalized lowest energy group scalar flux for the high scattering one dimensional two-grid test problem.

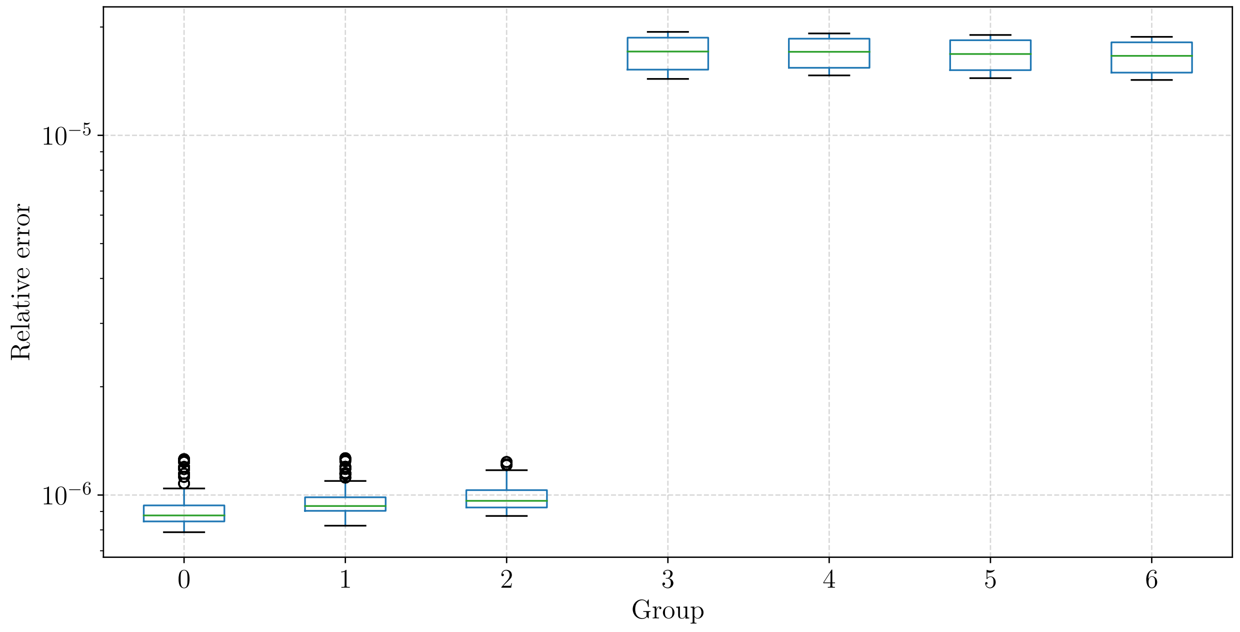


Figure 6.4: Relative error vs energy group for all groups for the 1D two-grid test problem; box plot showing the distribution of values across all degrees of freedom.

Table 6.2: Iteration summary for the unaccelerated 1D base case and two-grid.

	Iterations			
	Eigenvalue	Outer GS	Inner (Within-group)	Two-grid
Unaccelerated	5	441	6174	
Two-grid	5	21	294	21

We thus have a good indication that the two-grid method is efficiently accelerating the diffusion solve, requiring a twentieth of the GS and linear solves as the unaccelerated case. We can feel confident that this is a good comparison, given that differences between the two-grid case and the base case are isolated to the subroutine. Each completed GS outer iteration requires running the TG routine. The right-hand-side of the TG single-group diffusion equation is a constant, only requiring a single linear solve of the one-group transport operator.

Using our discussion of work in Chp. 4, we can calculate the required relative work of the subroutine for this method to be efficient. We know that for the method to be efficient, the following inequality must hold,

$$\frac{w'}{w_{\text{inv}}} < \left(\frac{N}{N'} - 1\right) .$$

For the values of these iterations, the method is efficient if and only if

$$w' < \left(\frac{6174}{294 + 21} - 1\right) w_{\text{inv}} = 19.6 w_{\text{inv}} .$$

We can now confidently state that this method is very efficient for our problem. While the subroutine has the same spatial discretization as the main diffusion problem, the collapse of the problem into a single energy group greatly decreases the total degrees of freedom. This reduction in problem size means that the subroutine linear solve requires less work than the inversion of the main problem. For sparse matrices, the generalized minimal residual (GMRES) conservatively requires floating point operations on the order of $\mathcal{O}(n)$ where n is the number of degrees of freedom. By this metric, by collapsing the seven energy groups into a single group, the TG method would require a reduction in iterations of

$$\frac{1}{7} < \left(\frac{N}{N'} - 1\right) \implies N' = 0.85 \times N ,$$

to be considered efficient and we achieved $N' \approx 0.05N$. This implementation of the routine is far more efficient than this conservative estimate. The overhead of prolongation and calculation of the isotropic residual may add further inefficiency, but far less than would be required for the routine to not be considered efficient.

6.1.4.2 Further Assessment Using BART Tools

The unique features and tooling of the BART code enable us to dig deeper into assessing why and how the TG method works. Unlike other codes, BART stores each right-hand-side term individually before assembling to solve. This enables us to instrument these terms themselves and examine how each of them converges. Using this, we can examine the convergence of the scattering source term itself. Taking the final scattering source term at the end of the GS iterations, we calculate the relative error between the same term at each step of the iteration. A plot of this convergence, summed over all the equally-sized groups, is shown in Fig. 6.5. We can observe the slow convergence of the scattering source in the unaccelerated case and the rapid convergence when TG is used. It is clear that the TG method is better at

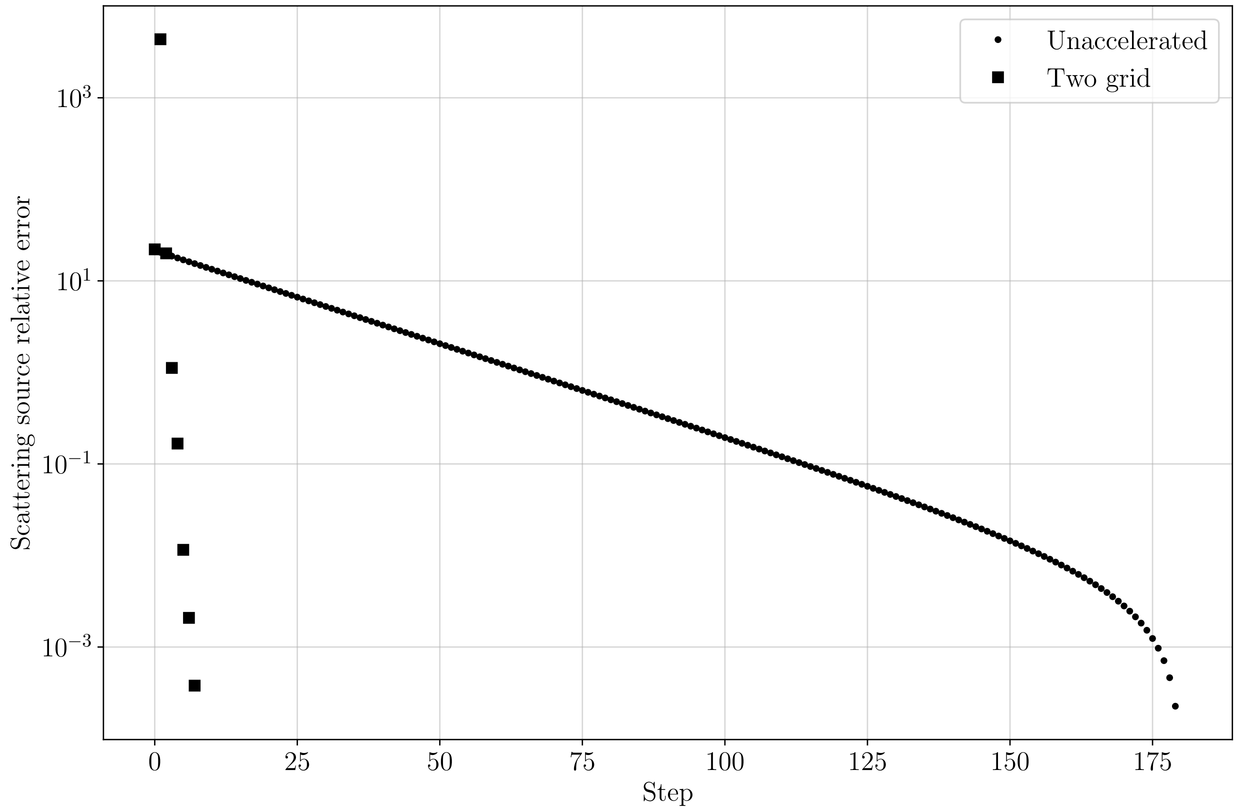


Figure 6.5: Convergence of the scattering source for all groups in the GS iterations for the first eigenvalue iteration.

converging the scattering source faster. Importantly, BART provided this information easily while other transport codes do not provide it at all, giving unique insight to confirm this method works the way we think it does.

6.1.5 Two-Dimensional Test Problem

The two dimensional test problem uses a central region made of the fissionable high scattering material surrounded by the reflective high scattering material. The layout is shown in Fig. 6.6, where the dark region is the fissionable material and white areas contain reflective materials and all boundary conditions are reflective. The domain is discretized into 100 cells in both dimensions resulting in 10,201 spatial degrees of freedom. The cross-sections and convergence criteria are the same as described in Sec. 6.1.4. The solutions for the high and low energy groups are shown in Fig. 6.7.

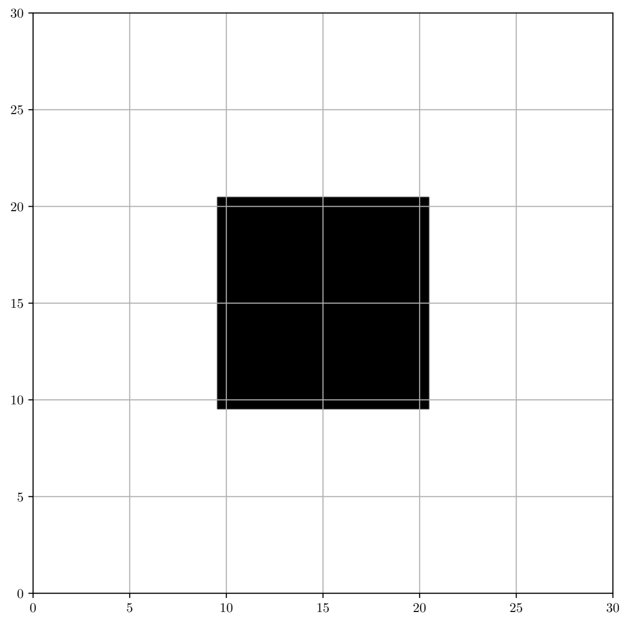


Figure 6.6: Two-grid 2D test problem layout, the dark region is the fissionable material and white areas contain reflective materials and all boundary conditions are reflective.

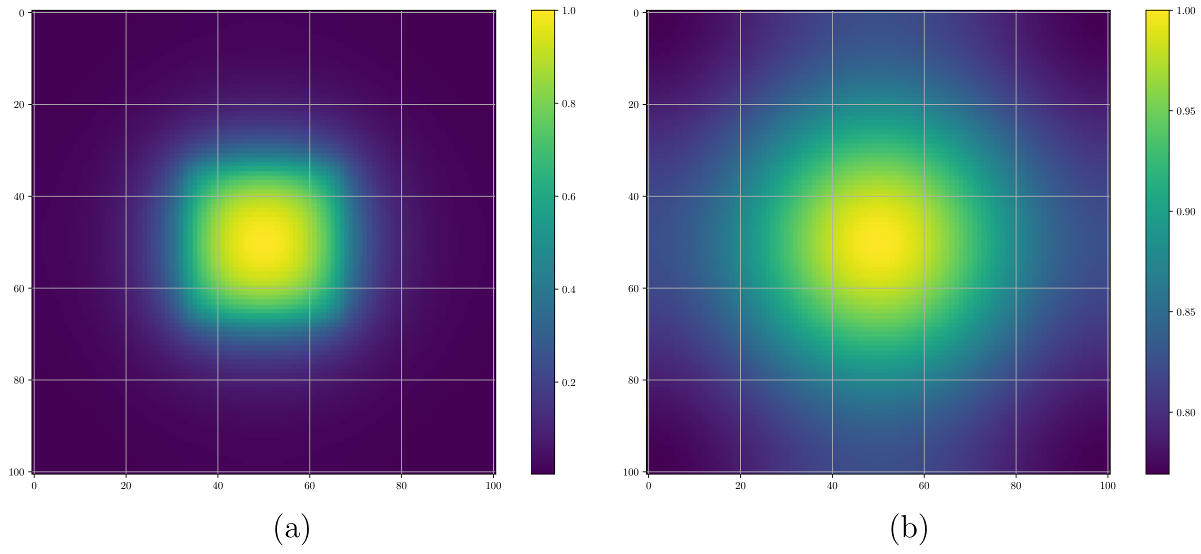


Figure 6.7: Two-dimensional test problem (a) highest and (b) lowest energy group solutions

Performing the same relative error analysis as the one-dimensional case, we see box plots of the error for all groups in Figs. 6.8. As with the one-dimensional case, we observe the highest errors in the groups with upscattering where the TG method results in the largest corrections. All relative errors are still very small, consistent with each method separately being converged to 1×10^{-6}, indicating a consistent result with the TG method. The k-eigenvalue calculated by the accelerated solve is consistent with the unaccelerated case, with a relative error of 1.14×10^{-5}.

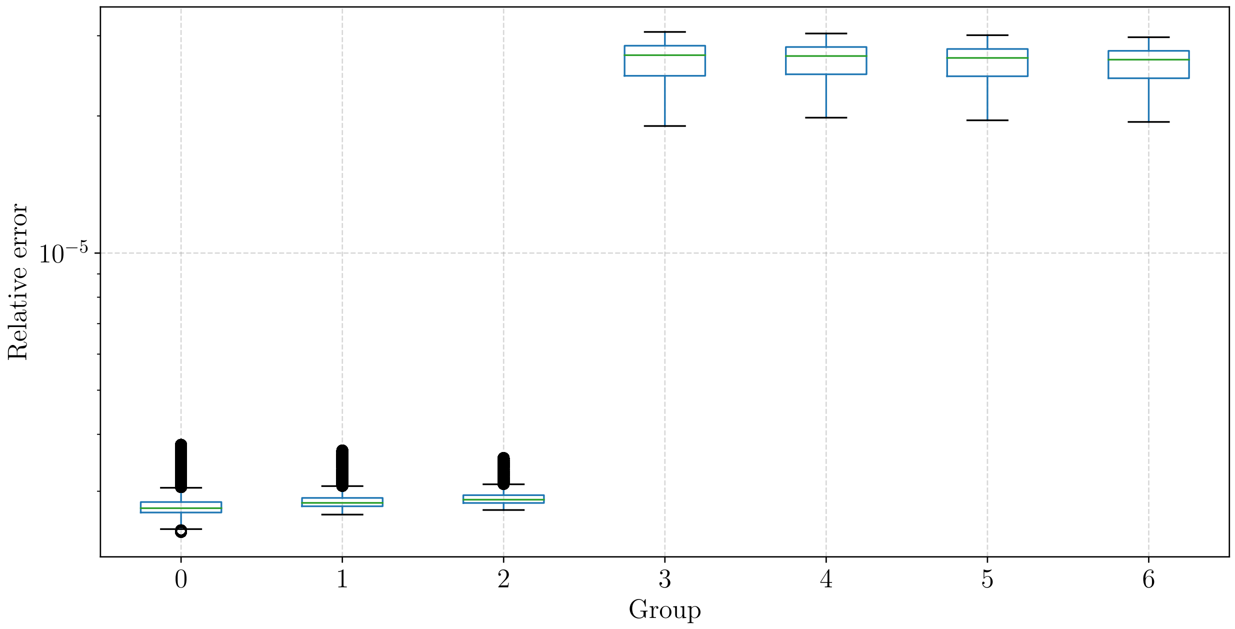


Figure 6.8: Relative error vs energy group for the two-grid 2D test problem, box plot showing the distribution of values across all degrees of freedom.

Table 6.3: Iteration summary for the unaccelerated base case and two-grid 2D test problem.

	Iterations			
	Eigenvalue	Outer GS	Inner (Within-group)	Two-grid
Unaccelerated	6	674	9436	
Two-grid	5	24	336	24

A summary of the iterations for the unaccelerated and TG two-dimensional test problem are shown in Table 6.3. As we observed with the one-dimensional case, the TG method results in a large reduction in the GS iterations and, as a result, the linear solves required. We again obtain the same number of inner iterations per outer iteration, as expected. Here there is a reduction in power iterations, likely because the eigenvector converged more quickly given better GS performance.

Using the same analysis as before, we find that in two-dimensions the TG method is efficient if

$$w' < \left(\frac{9436}{336 + 24} - 1\right) w_{\text{inv}} = 25.2 \times w_{\text{inv}} \ .$$

As with the one-dimensional case we can say with some certainty that the work of the TG subroutine is much less than 26 times the work of a linear solve of the main routine. We can therefore assess that the TG method is efficient in two-dimensions as well. We show the relative error in the scattering source as defined in the previous section in Fig. 6.9. Again, we see that the unaccelerated case suffers from slow convergence of the scattering source in GS iteration steps and that the TG method converges the source much more rapidly.

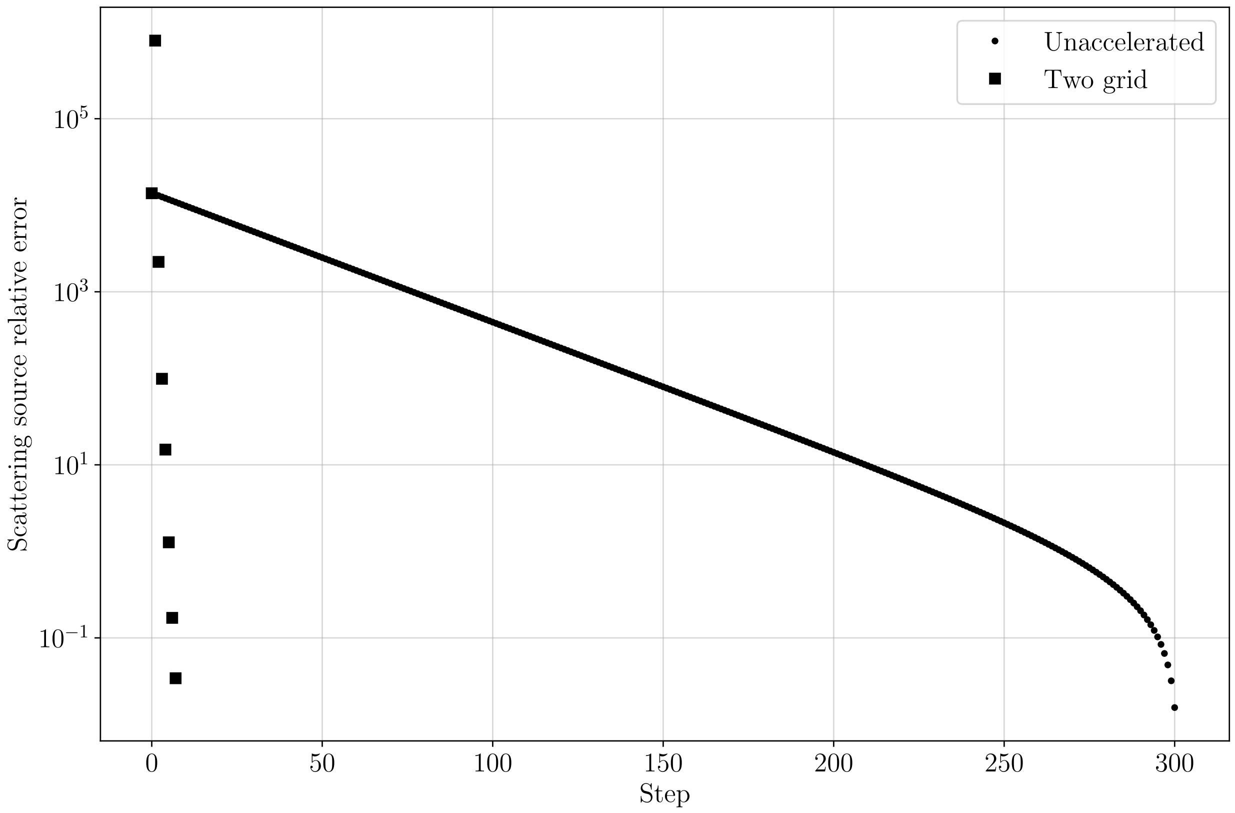


Figure 6.9: Convergence of the scattering source for all groups in the GS iterations for the first eigenvalue iteration for the two-dimensional test problem.

6.1.6 Three-Dimensional Test Problem

The three dimensional test problem uses a central region made of fissionable high scattering material surrounded by scattering material with reflective boundary conditions. The layout uses the same xy layout as the two-dimensional test problem shown in Fig. 6.6 and this layout is constant in the z-direction. The domain is discretized into 30 cells in the x- and y-directions with one cell in the z-direction. This results in 1922 spatial degrees of freedom. The cross-sections and convergence criteria are the same as the one- and two-dimensional test cases. Solution plots for the highest and lowest energy groups are shown in Figs. 6.10 and 6.11.

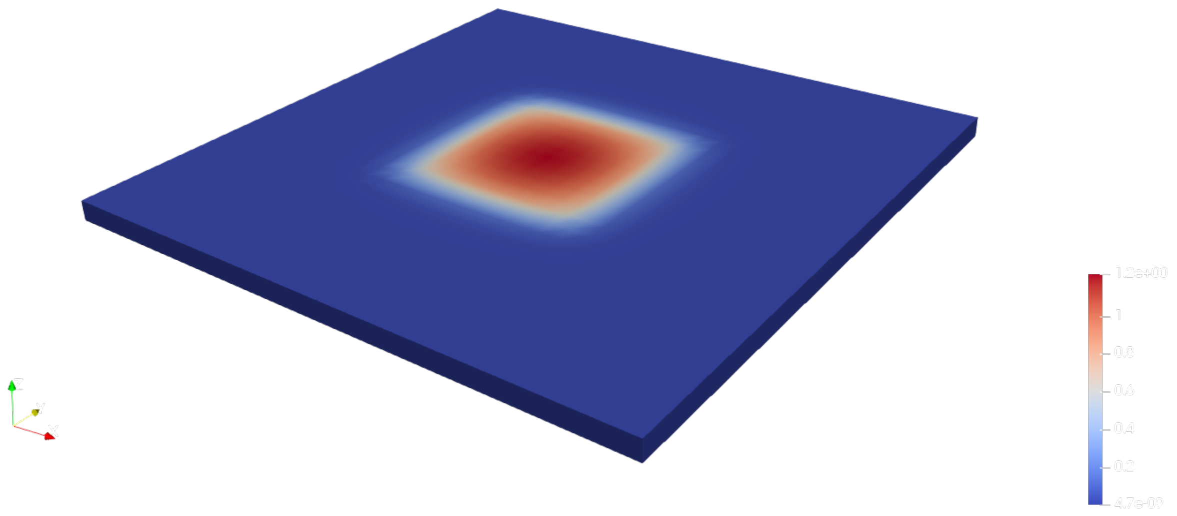


Figure 6.10: Three-dimensional test problem highest energy group solution.

The relative error for all energy groups is shown in Fig. 6.12. As with the other cases, we observe the highest errors in the groups with upscattering where the TG method results in the largest corrections. All relative errors are still small, indicating a consistent result with the TG method. We note that the errors are higher in this case than the lower-dimensional cases. The higher error comes from having all reflective boundary conditions in such a highly scattering system. The k-eigenvalue calculated by the accelerated solve is consistent with the unaccelerated case, with a relative error of 1.14×10^{-5}.

A summary of the iterations required for both cases are shown in Tab. 6.4. We find that

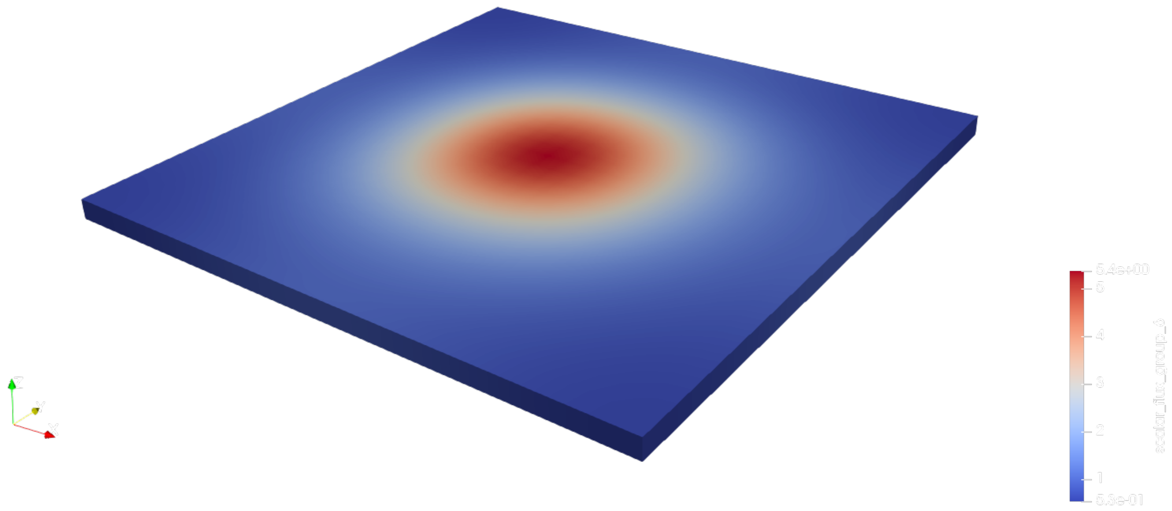


Figure 6.11: Three-dimensional test problem lowest energy group solution.

in three-dimensions, the TG method is efficient if

$$w' < \left(\frac{12600}{546+39} - 1\right) w_{\text{inv}} = 20.5 \times w_{\text{inv}} \ .$$

As expected, the TG method is efficient in three-dimensions, and based on the reduction in linear solves allows for the subroutine to be 20.5 times more costly than a linear solve of the base system. We see a nearly identical trend in the aggregated scattering source, shown in Fig. 6.13.

Table 6.4: Iteration summary for the unaccelerated base case and two-grid 3D test problem.

	Iterations			
	Eigenvalue	Outer GS	Inner (Within-group)	Two-grid
Unaccelerated	9	900	12600	
Two-grid	9	39	546	39

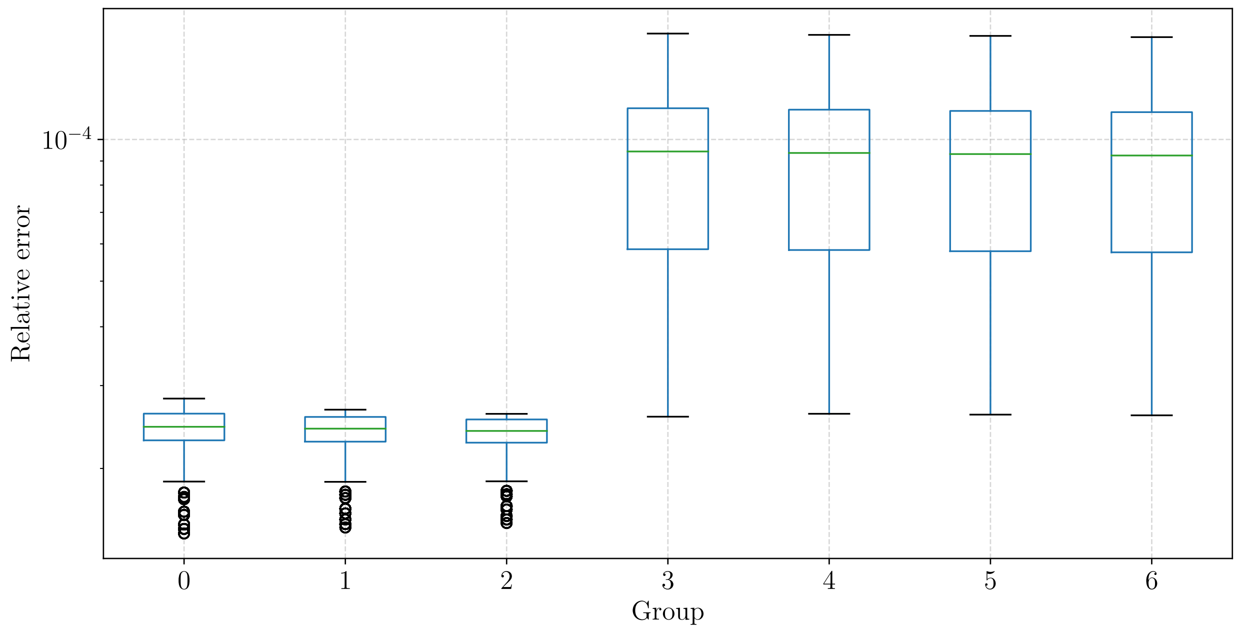


Figure 6.12: Relative error vs energy group for the two-grid 3D test problem, box plot showing the distribution of values across all degrees of freedom.

6.1.7 Summary

We have described a test problem in one-, two- and three-dimensions that uses fictional cross-sections designed to stress the convergence of GS iterations. We were able to use BART's capabilities to get detailed information from the method to assess how TG performed for this problem in a way that is not easy or tractable in other codes. We see consistently in each dimension that the scattering source suffers from slow convergence rates, resulting in a high number of GS iterations and linear solves. We have shown that the TG method accelerates this convergence. The accelerated solution is consistent, but the correction terms added by TG increases the error in groups with upscattering. In all dimensions, the number of required linear solves is reduced by a factor of more than 20. For the TG method to be inefficient, the subroutine would need to require 20 times more work than a linear solve of the main diffusion problem. We know that the subroutine does not require nearly that much work as the TG system contains a factor of seven fewer degrees of freedom by using a collapsed group structure. We can therefore assess that the TG method is efficient in all dimensions.

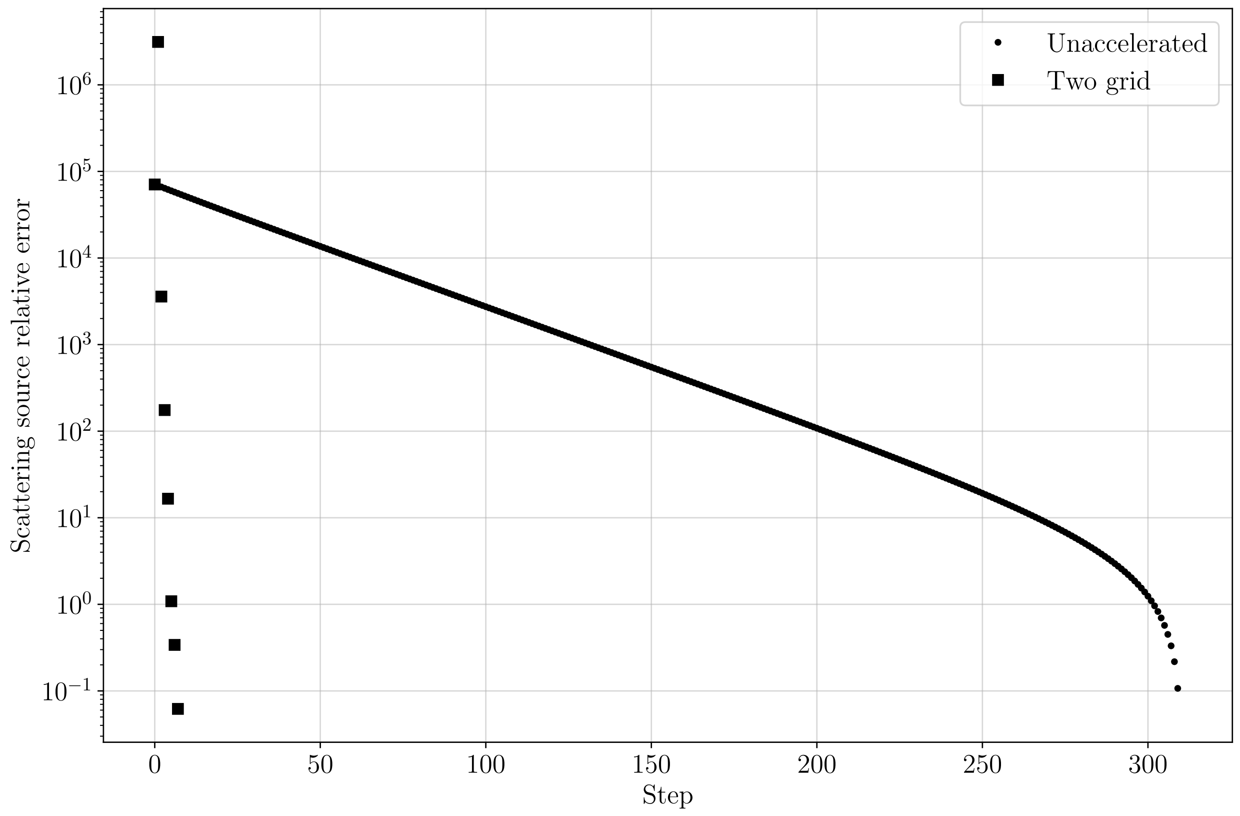


Figure 6.13: Convergence of the scattering source for all groups in the GS inner iterations for the first outer iteration for the three-dimensional test problem.

6.2 Nonlinear Diffusion Acceleration

In addition to the TG method, the NDA method described in Sec. 3.6 is implemented in the BART code. This method alters the GS iteration scheme to more efficiently converge inner iterations when the scattering ratio is high. The eigenvalue iteration process for the angular solve remains unchanged. To assess this method we will use an analytic benchmark from Sood *et al.*[30] in one- and three-dimensions. We will first present the implementation details of NDA in BART and then the results of the test cases.

6.2.1 Implementation Details

Similar to the TG method, the NDA method requires two frameworks. The main framework solves an angular form of the transport equation, such as the SAAF formulation. The auxiliary framework will solve the drift-diffusion formulation informed by the drift-diffusion vector calculated by the main framework. The frameworks and data flows are shown in Fig. 6.14. When solving for each group in the GS iteration, the solved group angular flux and scalar fluxes are passed to the NDA subroutine. The angular flux is used to calculate the drift-diffusion vector, and the scalar fluxes are used to calculate the scattering and fission sources. Following the convergence of the drift-diffusion equation, the group scalar flux is passed up to the main framework, and the next group is solved.

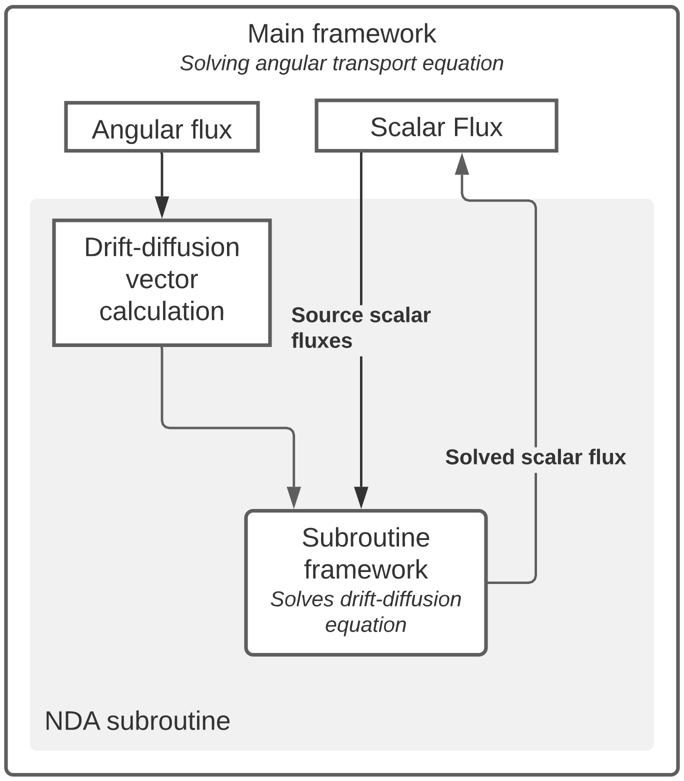


Figure 6.14: NDA method frameworks and data flows.

The drift-diffusion formulation adds the drift-diffusion term to the left-hand side of the diffusion formulation. The class in charge of updating the drift-diffusion formulation adds the scattering source as a fixed term, as defined by the NDA routine described by Park *et al.* [16]. Like with the TG method, the domain is shared by both frameworks to reduce overall memory consumption.

6.2.2 Instrumentation for Assessment

To assess and validate the effectiveness of NDA, we need to collect data to indicate if the within-group convergence is accelerated. We will compare this to a base case of SAAF without any acceleration using the following ports,

- `iteration::group::data_ports::NumberOfIterationsPort`: outputs the number of iterations required for the outer GS iteration to converge,
- `iteration::group::data_ports::NumberOfWithinGroupIterationsPort`: outputs the number of *within-group* inner iterations required by SI.

The NDA method is not expected to reduce the total number of outer GS iterations, as this will be mostly dependent on the amount of upscattering in the problem. We do expect fewer within group iterations to be required, indicating that it is more efficient at converging each individual group, if not all groups in total.

6.2.3 One-Dimensional Test Problem

To test the NDA method in one-dimension we will use a single material, six-group infinite medium problem. All convergence criteria are identical to the ones specified in Sec. 6.1.3. For the angular solve we used the SAAF formulation with Gauss-Legendre quadrature.

Cross-sections for test problem The analytic benchmark described by Sood *et al.* [30] uses a six-group cross-section structure that contains two coupled three-energy groups, similar to the cross-sections used for TG. The material properties are shown in Tab. 6.5. We observe the reflected pattern of the first three and last three energy groups. The scattering

Table 6.5: Material properties for the material used to assess the NDA acceleration scheme.

Group	0	1	2	3	4	5
χ_g	0.48	0.02	0.00	0.00	0.02	0.48
Σ_t	0.240	0.975	3.10	3.10	0.975	0.240
$\nu\Sigma_f$	0.018	0.15	1.80	1.80	0.15	0.018

cross-sections are,

$$\Sigma_s^{g'\to g} = \begin{bmatrix} 0.024 & & & & & \\ 0.171 & 0.600 & & & & \\ 0.033 & 0.275 & 2.000 & & & \\ & & & 2.000 & 0.275 & 0.033 \\ & & & & 0.600 & 0.171 \\ & & & & & 0.024 \end{bmatrix}$$

and the entry at row i, column j gives the scattering contribution into group i from group j. We see that the top and bottom energy groups are only coupled via fission. Using these cross-sections in a single medium with reflective boundary conditions, we expect constant

Table 6.6: Relative error between unaccelerated and NDA fluxes and expected ratios.

	Error	
	Unaccelerated	NDA
ϕ_1/ϕ_0	1.11×10^{-7}	4.47×10^{-9}
ϕ_4/ϕ_5	1.11×10^{-7}	4.47×10^{-9}
ϕ_3/ϕ_4	2.34×10^{-8}	2.23×10^{-8}
ϕ_2/ϕ_1	2.34×10^{-8}	2.23×10^{-8}
ϕ_2/ϕ_0	8.94×10^{-8}	1.79×10^{-8}
ϕ_3/ϕ_5	8.94×10^{-8}	1.79×10^{-8}

scalar fluxes with the following ratios,

$$\frac{\phi_1}{\phi_0} = \frac{\phi_4}{\phi_5} = 0.4800$$
$$\frac{\phi_3}{\phi_4} = \frac{\phi_2}{\phi_1} = 0.3125$$
$$\frac{\phi_2}{\phi_0} = \frac{\phi_3}{\phi_5} = 0.1500 \ .$$

In each case we will check that the solution is consistent with the expected solution by reporting the relative error of the flux to the expected ratio. We expect the value of the k-eigenvalue to be exactly 1.6.

The one-dimensional test problem was conducted with reflective boundary conditions and a domain triangulated into 10 segments of uniform size, resulting in 11 degrees of freedom. A Gauss Legendre angular quadrature was used with four angles. The scalar flux was found to be constant at all points as expected, with ratios described in Tab. 6.6 We observe that the relative errors between the simulated and expected ratios are very low for both the unaccelerated and NDA cases, indicating that the method returns a consistent solution. The value of the k-eigenvalue is also consistent, we have shown the absolute error in the calculated k-eigenvalues in Tab. 6.7.

Table 6.7: Error in the k-eigenvalue for the unaccelerated and accelerated cases for the NDA test problem. The expected value of $k_{\text{expected}} = 1.6$.

	$\lvert k - k_{\text{expected}} \rvert$
Unaccelerated	3.70×10^{-8}
NDA	1.31×10^{-13}

Table 6.8: Iteration summary for the unaccelerated 1D base case and NDA.

	Iterations		
	Eigenvalue	GS Outer	Inner (within-group)
Unaccelerated	3	33	1768
NDA	2	70	118

6.2.3.1 Iterations Assessment

Table 6.8 shows the error, GS iterations, and total within-group iterations for the unaccelerated case and the NDA accelerated case We see that the NDA modification to the inner iteration process requires more outer iterations, but greatly reduces the total number of within-group iterations required. Since we were changing the convergence of the inner iteration scheme, it is unsurprising that the number of outer iterations and therefore eigenvalue iterations changed. Each of these within-group iterations requires an inversion of the transport operator for solving the linear problem. We can therefore calculate the work ratio for the NDA scheme using,

$$w' < \left(\frac{1768}{118} - 1\right) w_{\text{inv}} = 14.0 \times w_{\text{inv}} \; .$$

We can therefore consider the NDA routine to be efficient if each execution is less than 14 times the cost of the standard GS iteration scheme. In fact, by solving a diffusion formulation of the transport equation, the NDA routine reduces the total degrees of freedom by the total number of angles in our quadrature set. We can therefore assess that the NDA method is efficient.

6.2.3.2 Further Assessment Using BART Tools

We can also examine the evolution of the Fourier modes of the error. It has been established by Hammer *et al.* [8] that the NDA method should reduce the spectral radius of the iteration matrix, and therefore also cause the Fourier modes with the largest wavelength to converge faster. As described in Sec. 4.4, the error mode that corresponds to $n = 0$ drives the slow convergence of the scattering source. When conducting an angular solve with a formulation such as SAAF, each GS iteration to converge the scattering source accounts for neutrons that scatter an extra time. For a problem with a large ratio of scattering, this can require many iterations and take arbitrarily long. This difficulty in capturing the diffusive behavior is characterized by these hard-to-suppress low frequency error modes. The NDA modified process uses a diffusion formulation to combat this behavior, capturing – in essence – the distribution of neutrons that have scattered an infinite number of times.

The NDA method's use of the diffusion formulation should improve this convergence compared to the standard GS method that relies on an angular formulation, SAAF. To examine this we will use the built-in one-dimensional discrete Fourier transform (DFT) tool in BART. This tool runs following each eigenvalue iteration and requires the code to be run twice, using the first run for the actual solution. In Fig. 6.15 we show the magnitude of the $n = 0$ Fourier mode of the error summed over all energy groups compared to the eigenvalue iteration that was just completed. After the first eigenvalue iteration, we see that the magnitude for this lowest-frequency error mode is more than 10^{12} larger for the unaccelerated case. This error is from the difficulty SAAF and other angular formulations have in regions where the diffusive nature of neutrons is important. The NDA method uses the drift-diffusion equation to better solve for this behavior. Therefore, just after a single eigenvalue iteration, this diffusive error mode has been significantly reduced by the NDA method, as we expected.

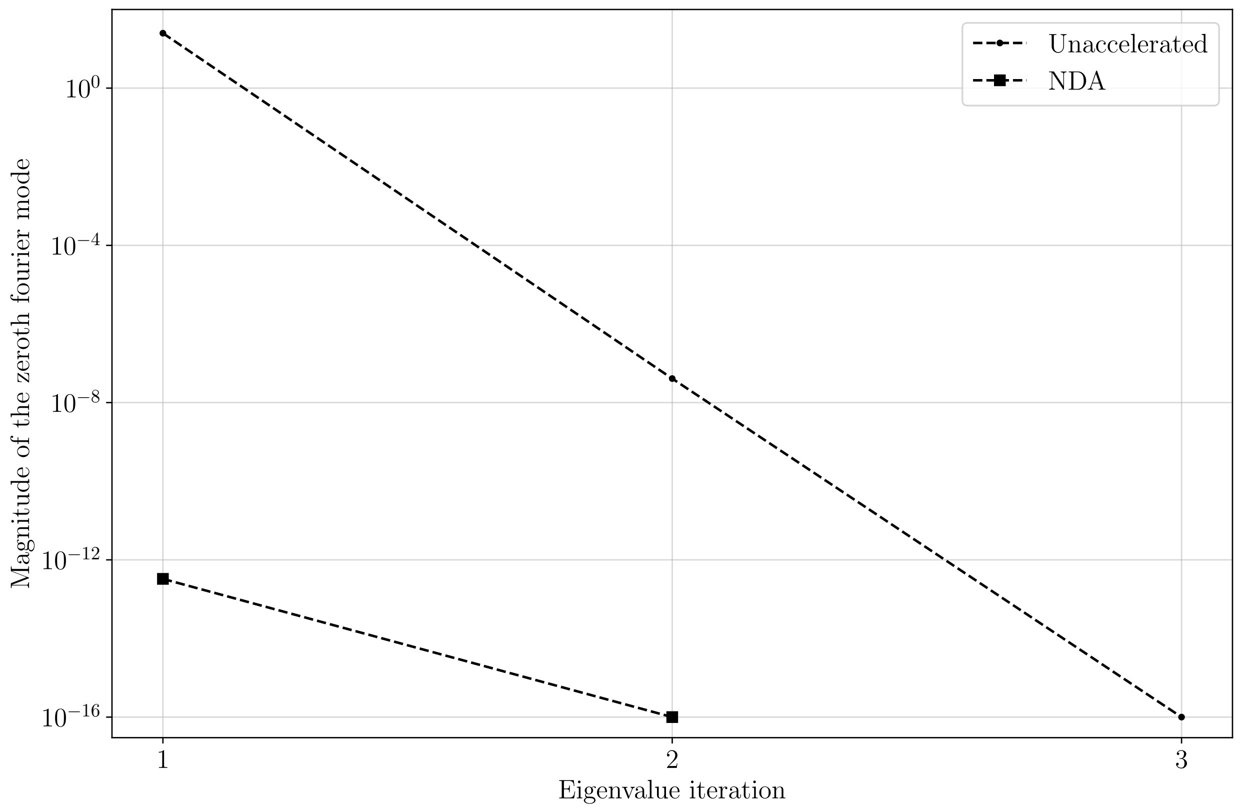


Figure 6.15: Zeroth moment of the error Fourier modes per eigenvalue iteration for the nonaccelerated and NDA cases.

We also expect that the high frequency error modes do not improve as much since NDA will not target these as well as the more diffusive modes. We See in Fig. 6.16 the magnitude of the Fourier mode summed over all groups for the highest frequency error mode we can observe in this problem, $n = 10$. We see that the NDA method has reduced the error mode more after the first iteration, but by only a factor of 10^4, eight orders of magnitude less than the improvement seen in the diffusive error mode. We see that the NDA method is not only efficiently converging the problem, but can validate that it is targeting the type of error we expected it to. This is exactly the kind of information we would like to be able to validate our methods and is not available in standard transport codes.

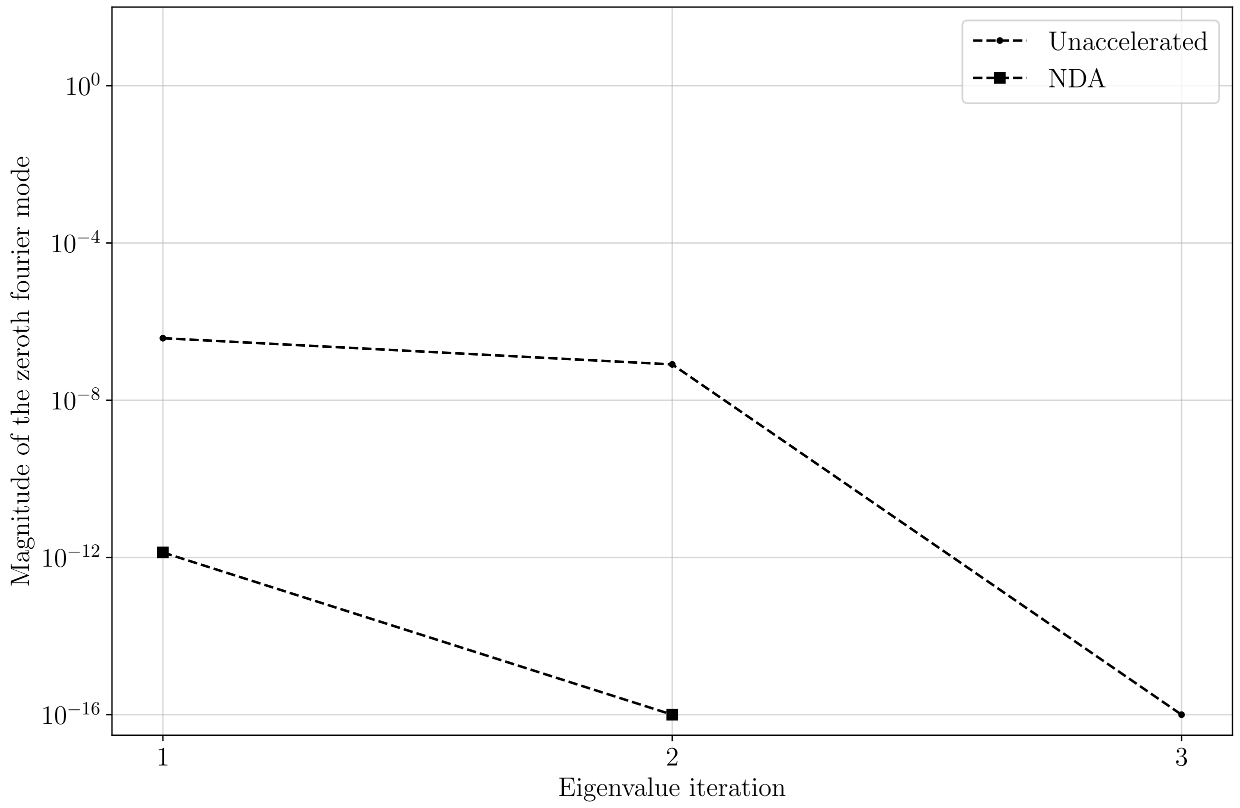


Figure 6.16: $n = 10$ moment of the error Fourier modes per eigenvalue iteration for the nonaccelerated and NDA cases.

6.2.4 Three-Dimensional Test Problem

To test the NDA method in three-dimensions we will use a single material, two-group infinite medium problem defined by Sood *et al.* [30]. All convergence criteria are identical to the ones specified in Sec. 6.1.3.

Cross-sections for test problem This analytic benchmark uses a two-group cross-section structure. The material properties are shown in Tab. 6.9. We observe the reflected pattern of the first three and last three energy groups. The scattering cross-sections are,

$$\Sigma_s^{g'\to g} = \begin{bmatrix} 0.0792 & 0.000 \\ 0.0432 & 0.23616 \end{bmatrix}$$

Table 6.9: Material properties for the material used to assess the NDA acceleration scheme in 3D.

Group	0	1
χ_g	0.575	0.425
Σ_t	0.2208	0.3360
$\nu\Sigma_f$	0.290	0.250

Table 6.10: Relative error for the unaccelerated and NDA flux ratios and k-eigenvalue for the 3D test problem.

	Relative Error	
	Unaccelerated	NDA
ϕ_0/ϕ_1	5.18×10^{-7}	5.40×10^{-7}
k	1.73×10^{-7}	1.83×10^{-7}

and the entry at row i, column j gives the scattering contribution into group i from group j. Using these cross-sections in a single medium with reflective boundary conditions, we expect constant scalar fluxes with the following ratio,

$$\frac{\phi_0}{\phi_1} = 0.675229 \ ,$$

and eigenvalue $k = 2.683767$. In each case we will check that the solution is consistent with the expected solution by reporting the relative error of the flux to the expected ratio.

The three-dimensional test problem was conducted with reflective boundary conditions and a domain triangulated into sixteen cells, with four cells in the y- and z-directions and a single cell in the x-direction. For the angular solve we used the SAAF formulation with a level-symmetric-like Gaussian quadrature of order $N = 4$, with 24 total angles. This results in 50 spatial degrees of freedom, with 2400 total degrees of freedom in the full phase space.

The scalar flux was found to be constant at all points as expected. The scalar flux ratio and the k-eigenvalue are consistent with the expected values, with relative errors shown in Tab. 6.10. We observe that the relative errors between the simulated and expected ratios and k-eigenvalue are very low for both the unaccelerated and NDA cases, indicating that the method returns a consistent solution.

Table 6.11 shows the error, GS iterations, and total within-group iterations for the unaccelerated case and the NDA accelerated case. We see that the NDA modification to the inner iteration process greatly reduces the total number of within-group iterations required. Since we were changing the convergence of the inner iteration scheme, it is unsurprising that the number of outer iterations and therefore eigenvalue iterations changed. Each of these within-group iterations requires an inversion of the transport operator for solving the linear

Table 6.11: Iteration summary for the unaccelerated 3D base case and NDA.

	Iterations		
	Eigenvalue	GS Outer	Inner (within-group)
Unaccelerated	3	44	953
NDA	2	28	56

problem. We can therefore calculate the work ratio for the NDA scheme using,

$$w' < \left(\frac{953}{56} - 1\right) w_{\text{inv}} = 16.0 \times w_{\text{inv}} \ .$$

We can therefore consider the NDA routine to be efficient if each execution is less than 16 times the cost of the standard GS iteration scheme. In fact, by solving a diffusion formulation of the transport equation, the NDA routine reduces the total degrees of freedom by the total number of angles in our quadrature set. We can therefore assess that the NDA method is efficient.

6.2.5 Summary

We have described test problems in one- and three-dimensions that uses fictional cross-sections based on an analytical benchmark by Sood *et al.* [30] to show the convergence properties of the NDA routine. We leveraged the tools in BART to show that the NDA scheme effectively accelerated convergence without introducing error into the solution. The NDA routine reduced the total iterations more than would be necessary to make the routine efficient, so we showed this method is useful for these kinds of problems. We also showed that the implemented BART DFT instrument was useful in identifying how the NDA routine improved this convergence in one-dimension. As expected, the use of the drift-diffusion equation more rapidly suppressed the lowest-frequency error mode than standard GS iterations by a significant margin. We can therefore assess that the NDA method is efficient in one-dimension and that the information easily available through BART lets us confirm why.

6.3 Conclusion

In this chapter, we have presented the implementation details of the TG and NDA methods. The BART code enabled us to implement them in a contained and controlled way, allowing for the minimal amount of code change to facilitate comparison. Further, the ease of adding instruments made the collection of useful data to assess and validate the methods straightforward. To assess the methods, we used cross-sections designed to stress the methods and the convergence of the GS iterative process. For the TG method, we see a large reduction in the required inner GS and within-group iterations and assess that the method is efficient in all dimensions. We also showed that the scattering source converges rapidly compared to the unaccelerated case, as expected with the method. For the NDA method, we found a significant reduction in the number of within-group solves required to converge the SAAF equation in one-, and three-dimensional analytic benchmarks. We also observe in the one-dimension case that the most diffusive error mode is suppressed much more rapidly, as expected with this method.

Importantly, we leveraged the ability of BART to instrument the solve and collect the data we needed to assess and validate these methods. Once collected, we can be sure that the data is useful for a good comparison because the structure of the BART code isolates the modifications from the rest of the solve process. It is important to note that BART works in one, two, and three dimensions. Many codes written as research codes do not due to complexity and computational time. We think the ability to test in 3D is an important addition as many methods behave differently depending one/two/three dimensions. Future work should focus on expanding the NDA test cases to more test problems, and implementing a similar DFT instrument in more dimensions for higher dimensional analysis.

Chapter 7

Conclusions and Future Work

In this dissertation, we discussed the design goals and implementation of the Bay Area Radiation Transport (BART) code and how it creates a new unique tool for analyzing acceleration methods. The BART code focuses on creating a developer end-user focused source code that minimizes the work required to implement new methods, and provides a unique environment for assessing and validating those methods. We then used this environment to assess and validate two acceleration methods, the two-grid (TG) and nonlinear diffusion acceleration (NDA) methods.

The neutron transport equation spans a large and complex phase space in space, angle and energy. This requires complex discretization when solving deterministically and, in general, a large number of degrees of freedom. Complicating the problem is that solving for the flux requires solving scattering and fission sources that are dependent on the flux itself. This and the size of the phase space necessitate the use of iterative methods that do not solve the problem directly, but rely on repeated iterations to converge to a solution. The source iteration (SI) scheme converges the scattering source term of the multigroup transport equation using a stationary iterative method. The convergence of the scheme is dependent on the ratio of scattering to total cross-sections and can take arbitrarily long as this ratio approaches unity. The power-iteration (PI) scheme similarly solves the k-eigenvalue criticality formulation of the transport equation. This is of particular interest for the modeling of reactors, as it gives a measure of departure from criticality. The convergence of the PI method is dependent on the ratio of the largest eigenvalue to the second largest, the dominance ratio.

The slow convergence of the SI has motivated the development of two acceleration methods we implemented here. These methods are designed to address inefficiencies in the iterative solving process, making it more efficient and ultimately reducing the time and iterations required for convergence. The first of these methods is TG, which uses a one-group collapsed diffusion formulation to accelerate the convergence of problems with a large amount of upscattering, the process by which neutrons at lower energies will scatter to higher energies. Upscattering requires repeatedly iterating over energy groups, as higher energy groups have a contribution from lower energy groups. This process can take arbitrarily long if there is enough upscattering. Within-group scattering can also cause issues with convergence, which motivated the development of the NDA routine. These methods, like other acceleration schemes, were developed using an in-depth understanding of the mathematical formulation of the iterative schemes. Expecting the methods will work based on mathematical properties is different than showing that the methods will work applied to practical problems. Such mathematical analysis is often done in very simplified conditions, such as one energy group or infinite media, and those simplifications may make the conclusions about method performance invalid in real problems. Assessing the effectiveness of acceleration methods requires three more steps: implementation, assessment, and validation.

We discussed the first challenge: we must actually implement the problem in a code that enables us to test it on real problems of interest. The two options, modifying an existing code and writing a new code both present unique challenges. Writing a new code requires a large investment of resources and time, especially if written to very high standards of coding. In general, it is not possible or not worth the time investment to make these codes comprehensively tested or reproducible. The second option requires modifying an existing code. This can also require an investment of time to learn the structure of the code and modify it appropriately. In both cases, the codes may not provide an ideal environment for assessing the effectiveness of the acceleration method once implemented.

The second major issue was also discussed: assessing the effectiveness of the method once implemented. Doing this requires an appropriate environment for assessing methods and good data collection. An appropriate environment provides the ability to compare an acceleration routine to a base case in a controlled manner. Creating an environment like this is difficult; modifications to code naturally creates cutouts and other logical branches that complicate program flow and make comparisons less than ideal. We discussed the benefit to a code designed to allow for insertion of acceleration methods with minimal change to the surrounding program flow.

Finally, we discussed the challenge of validating acceleration methods. Validation requires the collection of data to ensure that the method is efficient for the reasons we expect it to be. This can be difficult in existing codes as they typically do not provide access to the data required. Most codes the number of iterations and the runtime. While this is useful information, it is not enough to tell how methods are really working. Information like Fourier modes, convergence of different terms, etc. is more helpful for identifying how something is really working and adding such capabilities is difficult. These challenges motivated the development of a new code to provide new capabilities.

This work described the design goals of the BART project and code features implemented to meet these goals. One of the main goals of this project is to reduce the burden of implementing, assessing, and validating new and novel combinations of acceleration methods. To this end, we described how the code is designed to target and end-user who is a researcher interested in studying acceleration methods. The code is not designed to quickly solve problems of interest, but to provide a good environment for implementing new ways of solving the transport equation. A framework with this goal can eliminate the need for researchers to write new codes or modify existing codes to test acceleration methods.

Not only does the BART code make implementing these new methods easier, it provides a good environment and tools for assessing their effectiveness. As we described, the code is designed to leverage object oriented programming in such a way that parts of the solving routine can be changed without impacting the rest of the program flow or execution. This provides a controlled environment where unaccelerated solves can be easily compared to those accelerated by new methods. In addition, the BART code focuses on providing complex but easy to implement instrumentation to collect any kind of data required to assess these methods. The ability to extend this instrumentation framework to any kind of data needed was described, as well as those standard instruments already implemented to make assessment easy, including a built in discrete Fourier transform (DFT) tool for one-dimensional solves. There aren't any other codes that do this or have taken an approach like this. The ability to do this kind of detailed assessment and comparison is truly enabling to better methods development.

To showcase the ability of the BART code, we implemented two acceleration schemes: two-grid (TG) and nonlinear diffusion acceleration (NDA). We presented a test case using fictitious cross-sections to stress the convergence of the Gauss-Seidel (GS) SI process due to a large amount of upscattering. As expected, the scattering source took many iterations to slowly converge due to the interaction between groups. We showed that the TG method improved this issue, causing rapid convergence. The method performed well in one-, two-, and three-dimensions, accelerating the diffusion solve in a twentieth of the iterations. For the NDA method we showed improvement in fictitious infinite media analytic benchmark problems in one- and three-dimensions. The NDA method modified the GS inner iterations to more efficiently deal with diffusive neutron behavior. This was reflected in the more rapid convergence of the error modes linked to the most diffusive behavior of the neutrons, as shown by the DFT.

Future work includes improvement of the BART code to provide more tools to researcher developers. Other formulations of the transport equation should be implemented to provide angular forms other than the self-adjoint angular-flux (SAAF) formulation. Coupled with this, more quadrature sets should be implemented, including a 2D quadrature set, along with support for solving for flux modes other than the scalar flux. The code supports the use of Message Passing Interface (MPI) for solving the provided formulations generally and in all dimensions, but the implemented acceleration schemes are not implemented in a compatible fashion. Future version of BART should include support for MPI for all acceleration schemes already implemented. Importantly, inconsistencies in the NDA formulation occur when solving problems with more than one material, an issue that should be identified and resolved. Finally, the project should be expanded and modified to further the design goals, it should be updated to be more friendly to those less familiar with modern C++. The most important future work is identifying, innovating, and implementing new ways for the code to meet the original design goals that are at its heart.

Bibliography

[1] E.E. Lewis and W.F. Miller. *Computational Methods of Neutron Transport*. New York, NY: John Wiley & Sons, Inc., 1984.

[2] G.I. Bell and S. Glasstone. *Nuclear Reactor Theory*. New York, NY: Van Nostrand Reinhold Company, 1970.

[3] J.J. Duderstadt and L.J. Hamilton. *Nuclear Reactor Analysis*. New York, NY: John Wiley & Sons, Inc., 1976.

[4] D G Cacuci et al. "Eigenvalue-Dependent Neutron Energy Spectra : Definitions , Analyses , and Applications". In: 442 (1982), pp. 432–442.

[5] Katherine E. Royston et al. "Application of the Denovo Discrete Ordinates Radiation Transport Code to Large-Scale Fusion Neutronics". In: *Fusion Science and Technology* 74.4 (2018), pp. 303–314. ISSN: 19437641. DOI: 10.1080/15361055.2018.1504508. URL: https://doi.org/10.1080/15361055.2018.1504508.

[6] B. T. Rearden and M.A. Jessee. *SCALE Code System*. 6.2.3. 2016, p. 2747. ISBN: 1800553684.

[7] C. Johnson. *Numerical Solution of Partial Differential Equations by the Finite Element Method*. Mineola, NY: Dover Publications, Inc., 2009.

[8] Hans R Hammer, Jim E. Morel, and Yaqi Wang. "Nonlinear Diffusion Acceleration in Voids for the Weighted Least-Square Transport Equation". In: *Mathematics and Computation2*. 2017.

[9] J E Morel and J M Mcghee. "A Self-Adjoint Angular Flux Equation". In: *Nuclear Science and Engineering* 132 (1999), pp. 312–325. ISSN: 0003018X. DOI: 10.13182/NSE132-312.

[10] Jon Hansen et al. "A least-squares transport equation compatible with voids". In: *Journal of Computational and Theoretical Transport* 43.1-7 (2014), pp. 374–401. ISSN: 23324325. DOI: 10.1080/00411450.2014.927364.

[11] Anne Greenbaum. *Iterative Methods for Solving Linear Systems*. Society for Industrial and Applied Mathematics, 1997. ISBN: 978-0-89871-396-1.

[12] Marvin L. Adams and Edward W. Larsen. "Fast iterative methods for discrete-ordinates particle transport calculations". In: *Progress in Nuclear Energy* (2002). ISSN: 01491970. DOI: 10.1016/S0149-1970(01)00023-3.

[13] H. J. Kopp. "Synthetic Method Solution of the Transport Equation". In: *Nuclear Science and Engineering* 17.1 (1963), pp. 65–74. ISSN: 0029-5639. DOI: 10.13182/nse63-1.

[14] V. I. Lebedev. "An Iterative KP-Method". In: *USSR Computational Mathematics and Mathematical Physics* 7.6 (1967), pp. 53–76.

[15] Edward W. Larsen. "Unconditionally Stable Diffusion-Synthetic Acceleration Methods for the Slab Geometry Discrete Ordinates Equations. Part I: Theory". In: *Nuclear Science and Engineering* 82.1 (1982), pp. 47–63. ISSN: 0029-5639. DOI: 10.13182/NSE82-1. URL: https://www.tandfonline.com/doi/full/10.13182/NSE82-1.

[16] H. Park, D. A. Knoll, and C. K. Newman. "Nonlinear Acceleration of Transport Criticality Problems". In: *Nuclear Science and Engineering* 172.1 (Sept. 2012), pp. 52–65. ISSN: 0029-5639. DOI: 10.13182/NSE11-81. URL: https://www.tandfonline.com/doi/full/10.13182/NSE11-81.

[17] B. T. Adams and Jim E. Morel. "A Two-Grid Acceleration Scheme for the Multigroup S n Equations with Neutron Upscattering". In: *Nuclear Science and Engineering* 115.3 (1993), pp. 253–264. ISSN: 0029-5639. DOI: 10.13182/NSE115-253. URL: https://www.tandfonline.com/doi/full/10.13182/NSE115-253.

[18] Thomas M. Evans, Kevin T. Clarno, and Jim E. Morel. "A Transport Acceleration Scheme for Multigroup Discrete Ordinates with Upscattering". In: *Nuclear Science and Engineering* 165 (2010), pp. 292–304. ISSN: 00295639. DOI: 10.13182/NSE09-28.

[19] Latifa Ben Arfa Rabai, Barry Cohen, and Ali Mili. "Programming language use in US academia and industry". In: *Informatics in Education* 14.2 (2015), pp. 143–160. ISSN: 16485831. DOI: 10.15388/infedu.2015.09.

[20] Shrirang Abhyankar et al. "PETSc DMNetwork: A Library for Scalable Network PDE-Based Multiphysics Simulations". In: *ACM Trans. Math. Softw.* 46.1 (Apr. 2020). ISSN: 0098-3500. DOI: 10.1145/3344587. URL: https://doi.org/10.1145/3344587.

[21] Trilinos. *Primary repository for the Trilinos Project.* https://github.com/trilinos/Trilinos. Version 13.0.1. 2021.

[22] The Idaho National Lab. *Multiphysics Object Oriented Simulation Environment.* https://github.com/idaholab/moose. 2021.

[23] *Programming languages — C++.* Standard ISO/IEC 14882:2020. Geneva, CH, Dec. 2020. URL: https://www.iso.org/standard/79358.html.

[24] Daniel Arndt et al. “The `deal.II` Library, Version 9.2”. In: *Journal of Numerical Mathematics* 28.3 (2020), pp. 131–146. DOI: `10.1515/jnma-2020-0043`. URL: `https://dealii.org/deal92-preprint.pdf`.

[25] Daniel Arndt et al. “The deal.II finite element library: Design, features, and insights”. In: *Computers & Mathematics with Applications* 81 (2021), pp. 407–422. ISSN: 0898-1221. DOI: `10.1016/j.camwa.2020.02.022`. URL: `https://arxiv.org/abs/1910.13247`.

[26] Youcef Saad and Martin H. Schultz. “GMRES: A GENERALIZED MINIMAL RESIDUAL ALGORITHM FOR SOLVING NONSYMMETRIC LINEAR SYSTEMS”. In: *Journal of Scientific and Statistical Computing* 7.3 (1986), pp. 856–869.

[27] K. Atkinson and W. Han. *Spherical Harmonics and Approximations on the Unit Sphere: An Introduction.* 2012. ISBN: 978-3-642-25983-8. URL: `https://www.springer.com/gp/book/9783642259821`.

[28] *Google Protocol Buffers.* `https://developers.google.com/protocol-buffers`. Accessed: 2021.

[29] D. A. Brown et al. “ENDF/B-VIII.0: The 8th Major Release of the Nuclear Reaction Data Library with CIELO-project Cross Sections, New Standards and Thermal Scattering Data”. In: *Nuclear Data Sheets* 148 (2018), pp. 1–142. ISSN: 00903752. DOI: `10.1016/j.nds.2018.02.001`.

[30] Avneet Sood, R.A. Forster, and D.K. Parsons. *Analytical Benchmark Test Set for Criticality Code Verification.* Tech. rep. Los Alamos National Lab, 1999.

[31] Barry D Ganapol. *Analytical Benchmarks for Nuclear Engineering Applications.* 6292. 2009, p. 269. ISBN: 978-92-64-99056-2. URL: `http://www.worldcat.org/title/analytical-benchmarks-for-nuclear-engineering-applications-case-studies-in-neutron-transport-theory/oclc/430990895%7B%5C%%7D5Cnpapers3://publication/uuid/F1AEDE79-EFE7-4E77-A82C-8596463847D3`.

[32] Nam Zin Cho. *Benchmark Problem 3A : MOX Fuel-Loaded Small PWR Core (MOX Fuel with Zoning) (7 Group Heterogeneous Cells).* Tech. rep. KAIST, 2000.

[33] *Programming languages — C++.* Standard ISO/IEC 14882:2017. Geneva, CH, Dec. 2017. URL: `https://www.iso.org/standard/68564.html`.

[34] Google LLC. *GoogleTest - Google Testing and Mocking Framework.* `https://github.com/google/googletest`. Version 1.11.0. 2021.

Appendix A

Spherical Harmonics as Eigenfunctions of the Scattering Operator

The spherical-harmonic functions (in tesseral form) of degree ℓ and order m are given by,

$$Y_{\ell,m}(\vec{\Omega}) = \begin{cases} (-1)^m \sqrt{C_\ell^m} P_\ell^{|m|}(\mu) \sin(|m|\phi) & -\ell \le m < 0 \\ (-1)^m \sqrt{C_\ell^m} P_\ell^{m}(\mu) \cos(m\phi) & 0 \le m \le \ell \end{cases} ,$$

where $P_\ell^m(\mu)$ is the associated Legendre function, μ is the cosine of the polar angle θ, and the constant is defined by

$$C_\ell^m = (2 - \delta_{m0}) \frac{2\ell + 1}{4\pi} \frac{(\ell - |m|)!}{(\ell + |m|)!} .$$

The functions are orthogonal, with the constant ensuring that they are orthonormal,

$$\int_0^{2\pi} \int_{-1}^{1} Y_\ell^m(\vec{\Omega}) Y_k^n(\vec{\Omega}) = \delta_{\ell k} \delta_{mn} .$$

If we operate on the spherical harmonic with our scattering operator

$$S Y_\ell^m(\vec{\Omega}) = \int_{4\pi} \Sigma_s(\vec{\Omega}' \cdot \vec{\Omega}) Y_\ell^m(\vec{\Omega}') \, d\vec{\Omega}' . \tag{A.1}$$

We will then expand the cross-section using the spherical harmonic basis as well, we can replace the cross-section with a summation of cross-section moments using the associated Legendre functions,

$$\sigma_k = \int_{-1}^{1} \sigma_s(\mu_0) P_k^0(\mu_0) \, d\mu_0 , \tag{A.2}$$

where we have defined $\mu_0 = \vec{\Omega}' \cdot \vec{\Omega} = \cos(\theta)$, and allowing us to redefine the differential cross-section,

$$\sigma_s(\vec{\Omega}' \cdot \vec{\Omega}) = \sum_{k=0}^{\infty} \sigma_k P_k^0(\vec{\Omega}' \cdot \vec{\Omega}) \ . \tag{A.3}$$

Plugging Eq. (A.3) into Eq. (A.4),

$$SY_\ell^m(\vec{\Omega}) = \int_{4\pi} \sum_{k=0}^{\infty} \sigma_k P_k^0(\vec{\Omega}' \cdot \vec{\Omega}) Y_\ell^m(\vec{\Omega}') \, d\vec{\Omega}' \ . \tag{A.4}$$

We then apply the addition theorem for the associated Legendre polynomial,

$$P_k^0(\vec{\Omega}' \cdot \vec{\Omega}) = \sum_{n=-k}^{k} Y_k^n(\vec{\Omega}) Y_k^n(\vec{\Omega}')$$

and plug this into Eq. (A.4),

$$SY_\ell^m(\vec{\Omega}) = \int_{4\pi} \sum_{k=0}^{\infty} \sigma_k \sum_{n=-k}^{k} Y_k^n(\vec{\Omega}) Y_k^n(\vec{\Omega}') Y_\ell^m(\vec{\Omega}') \, d\vec{\Omega}' \ .$$

The orthonormality conditions of the spherical harmonics collapses the summations, requiring $m = n$ and $k = \ell$, reducing this integral to

$$SY_\ell^m(\vec{\Omega}) = \sigma_\ell Y_\ell^m(\vec{\Omega}) \tag{A.5}$$

As we can see from Eq. (A.5), the spherical harmonics are eigenfunctions of the scattering operator with eigenvalues equal to the cross-section moments.

Appendix B

Quadrature Set Codes

This appendix contains the code required to make and test the quadrature sets and graphs described in Sec. 5.1.3.1 using Python 3. The following code is required for all code in this section:

```
import numpy as np
import matplotlib.pyplot as plt

plt.rc('text', usetex=True)
plt.rc('font', family='serif')

def z_circle(ax, z):
    phi = np.linspace(0, np.pi/2, 100)
    zs = np.ones_like(phi) * z
    xs = np.sqrt(1 - z*z) * np.cos(phi)
    ys = np.sqrt(1 - z*z) * np.sin(phi)
    ax.plot(xs, ys, zs, '--', c='k', alpha=0.05)
```

B.1 Product Gaussian Quadrature

```
n = 16
m = 2*n

[l_points, l_weights] = np.polynomial.legendre.leggauss(n)

xs = []
ys = []
zs = []
ws = []
phis = [np.pi/n * j for j in range(m + 1)]

for i in range(n):
    z = l_points[i]
    for j in range(m):
        x = np.sqrt(1 - z*z) * np.cos(phis[j])
        y = np.sqrt(1 - z*z) * np.sin(phis[j])
        if (x >= 0 and y >= 0 and z >= 0):
            xs.append(x)
            ys.append(y)
            zs.append(z)
            ws.append(l_weights[i])

plot_weight = [500*weight for weight in ws]
fig = plt.figure(figsize=plt.figaspect(1.0)*3.0)
ax = fig.add_subplot(projection='3d')
ax.set_box_aspect(aspect = (1,1,1))
ax.scatter(xs, ys, zs, c='b', alpha=1.0, s=plot_weight)
ax.text(0.01, 0, .01, "$O$", zdir=None, fontsize=16)
ax.scatter(0, 0, 0, c='k')

for z in zs: z_circle(ax, z)

plt.xlim([0, 1])
plt.ylim([1, 0])
ax.set_zlim(0, 1)
ax.set_xlabel('$x$', fontsize=16)
ax.set_ylabel('$y$', fontsize=16)
ax.set_zlabel('$z$', fontsize=16)

plt.show()
```

B.2 Level-Symmetric-Like Gaussian Quadrature

```
order = 16
n_points = int(order/2) # The number of points and levels per octant
[l_points, l_weights] = np.polynomial.legendre.leggauss(order)

xs = []
ys = []
zs = []
ws = []

for level in range(n_points):
    points_this_level = level + 1
    mu = -l_points[level]
    dphi = np.pi/(2 * points_this_level)
    weight = l_weights[level] * np.pi/(points_this_level)

    for j in range(points_this_level):
        phi = (j + 0.5) * dphi
        xs.append(np.sqrt(1 - mu * mu) * np.cos(phi))
        ys.append(np.sqrt(1 - mu * mu) * np.sin(phi))
        zs.append(mu)
        ws.append(weight)

plot_weight = [500*weight for weight in ws]
fig = plt.figure(figsize=plt.figaspect(1.0)*3.0)
ax = fig.add_subplot(projection='3d')
ax.set_box_aspect(aspect = (1,1,1))
ax.scatter(xs, ys, zs, c='b', alpha=1.0, s=plot_weight)
ax.text(0.01, 0, .01, "$O$", zdir=None, fontsize=16)
ax.scatter(0, 0, 0, c='k')

for z in zs:
    z_circle(ax, z)

plt.xlim([0, 1])
plt.ylim([1, 0])
ax.set_zlim(0, 1)
ax.set_xlabel('$x$', fontsize=16)
ax.set_ylabel('$y$', fontsize=16)
ax.set_zlabel('$z$', fontsize=16)

plt.show()
```

B.3 Calculating the Error for a Quadrature Set

The following code was used to test the above-generated quadrature sets to calculate the error.

```
import scipy.special as spec
import pandas as pd

phis = []
thetas = []
for i in range(len(xs)):
    phis.append(np.arctan(ys[i]/xs[i]))
    thetas.append(np.arccos(zs[i]))

data = []
l_max = 4
for l in range(l_max + 1):
    for m in range(-l, l + 1):
        integration = 0
        for i in range(len(phis)):
            harmonic = spec.sph_harm(m, l,phis[i], thetas[i])
            integration += ws[i] * np.absolute(harmonic * harmonic)
        error = np.abs(1 - 4*integration)
        data.append([l, m, error])
df = pd.DataFrame(data, columns = ['l', 'm', 'error'])
```