alpha-eigen:

Performing dynamic-mode decomposition to calculate time-eigenvalues of the time-dependent radiation transport equation.

Kayla Clements

Center for Exascale Monte-Carlo Transport (CEMeNT), Oregon State University

Contents

1 Introduction

In radiation transport theory, time-eigenvalues (also called alpha-eigenvalues) are calculated to understand the exponential time-behavior of angular flux in a system. Deterministically solving for this time-eigenvalue is complex, and various methods have been developed to calculate it. One of the most common of these methods is a k-eigenvalue search, in which guesses of the time-eigenvalue are iterated upon until the practitioner converges to the time-eigenvalue which results in a critical k-eigenvalue. This requires solving a deterministic problem for angular flux, typically finely discretized in time, energy, space, and angle, repeatedly until convergence is reached. Depending on the complexity of the system, this method can be discouragingly computationally expensive unless guesses of the system's alpha-eigenvalues are known a-priori. These methods also only calculate the fundamental eigenvalue mode, and while this is the most important, other modes can be useful for understanding transient behavior in a system or how perturbations might affect it.

Recently, Ref. [3] introduced a novel method of calculating time-eigenvalues using dynamic mode decomposition (DMD). Rather than explicitly calculate the transport operator of the radiation transport equation, the angular fluxes at a series of time steps are used to estimate properties of the transport operator such as the time-eigenvalue. This is useful because it doesn't require an iterative search, was shown to be accurate, and finds many eigenvalue modes, not just the fundamental one.

The goal of this project was to determine whether DMD could be performed on scalar flux, rather than angular flux, to calculate time-eigenvalues. Full angular flux solutions are typically required to calculate time-eigenvalues, which will be shown in more depth later. However, scalar flux is much more physically meaningful - in an experimental detector configuration, the detector wouldn't tally the angular incidence of radiation, just that it occurred. Along those lines, if angular flux were not needed, Monte Carlo radiation transport methods could then be used to calculate the scalar fluxes for the DMD process. In this project, I aimed to develop a package which applied DMD based on McClarren's work in [3] and test whether the same methodology could be applied to scalar flux solutions. Ideally, the software would eventually be expanded to include DMD applications on non-deterministic solutions, like scalar flux tallies from CEMeNT's Monte Carlo Dynamic Code (MCDC), a time-dependent Monte Carlo radiation transport solver.

2 Methodology

In the hopes of minimizing the amount of in-depth math needed to fully explain DMD, I point the interested reader to [4] for a deeper description of DMD applied to fluid-dynamical and transport processes, and [3] for a full derivation of how they're applied here. The time-dependent radiation transport equation can be written as

$$\frac{\partial \psi}{\partial t} = A\psi\left(x, \Omega, E, t\right) \tag{1}$$

where $\psi\left(x, \Omega, E, t\right)$ is the angular flux at position x, angle Ω, energy E, and time t. The transport operator A is

$$A = v(E)\left(-\Omega\dot{\nabla} - \sigma_t + S + f\right) \tag{2}$$

where the scattering and fission operators S and F are defined as

$$S\psi = \int_{4\pi} d\Omega' \int_0^\infty dE' \sigma_s\left(\Omega' \to \Omega, E' \to E\right) \times \psi\left(x, \Omega, E, t\right) \tag{3}$$

$$F\psi = \frac{\chi(E)}{4\pi} \int_0^\infty dE' \nu\sigma_f(E')\phi\left(x, E, t\right) \tag{4}$$

These are in terms of

- $\sigma_s(\Omega' \to \Omega, E' \to E)$, the scattering cross section from direction Ω' and energy E' to direction Ω and energy E;
- $\nu\sigma_f(E')$, the fission cross section times the average number of fission neutrons born from energy E';
- and $\chi(E)$, the probability of a fission neutron being emitted with energy E.

The scalar flux $\phi(x, E, t)$ is the integral of the angular flux over the unit sphere, *ie*, the contribution to the flux at (x, E, t) from radiation traveling in all directions.

If we treat Equation 1 as a discrete linear algebra problem of the form

$$\frac{\partial \mathbf{\Psi}}{\partial t} = \mathbf{A}\mathbf{\Psi}(t) \tag{5}$$

then at discrete time-step n this can also be written in terms of the matrix exponential as

$$\psi_{\mathbf{n}} = e^{\mathbf{A}\Delta t}\psi_{\mathbf{n-1}}. \tag{6}$$

Across all time steps, this relation can be written

$$\mathbf{\Psi}_+ = e^{\mathbf{A}\Delta t}\mathbf{\Psi}_- \tag{7}$$

where Ψ_+ is the matrix of n solutions and Ψ_- is the matrix of $n-1$ solutions. It was shown that rather than directly constructing $\mathbf{A}$ and finding its eigenvalues, one could instead perform a singular value decomposition of the left operator $\mathbf{\Psi}_- = \mathbf{U\Sigma V}^*$, construct an surrogate matrix $\tilde{\mathbf{S}} = \mathbf{U}^*\mathbf{\Psi}_+\mathbf{V\Sigma}^{-1}$, and that $eig(\tilde{\mathbf{S}}) = eig(e^{\mathbf{A}\Delta t})$ [3]. This is extremely useful for problems such as radiation transport, because the transport operator $\mathbf{A}$ would be very large and expensive to compute. Instead, a numerical approximation can be used to deterministically solve for ψ at N time steps to create the matrix $\mathbf{\Psi}_-$, then create the surrogate $\tilde{\mathbf{S}}$ and find its eigenvalues.

How does this relate to the time-eigenvalue? Solutions to the transport equation can be written in terms of a time-independent angular flux solution $\hat{\psi}(x, \Omega, E)$ and the exponential time-eigenvalue,

$$\psi(x, \Omega, E, t) = \hat{\psi}(x, \Omega, E)\, e^{\alpha t} \tag{8}$$

hence the alternate name 'alpha-eigenvalue.' Because α is an eigenvalue of $\mathbf{A}$, $e^{\alpha\Delta t}$ is an eigenvalue of $e^{\mathbf{A}\Delta t}$ [3]. So, to find the alpha eigenvalue, we would need to:

1. Use a given problem definition to deterministically calculate Ψ at N dependent time steps.
2. Perform a singular value decomposition on the resulting $\mathbf{\Psi}_-$ matrix to form the surrogate matrix $\tilde{\mathbf{S}}$.
3. Find the eigenvalues of $\tilde{\mathbf{S}}$.
4. Compute the time-eigenvalues using the relation $eig(\tilde{\mathbf{S}}) = eig(e^{\mathbf{A}\Delta t}) = e^{\alpha\Delta t}$.

To compare the performance of DMD on scalar flux to its performance on angular flux, we could store both the angular and scalar fluxes at each time step and compare the alpha-eigenvalues calculated by performing DMD on both.

2.1 Discretization

Though the solution of the transport equation is just a stepping-stone to what I'm really interested in, the alpha-eigenvalues, a fair amount of the work of this project did end up being writing the transport solver itself in a way that implemented best software-development practices. The problem is discretized in time, energy, angle, and space. Though the angular flux at each time-step is solved independently of the other time-steps, a backward Euler approximation is used in time,

$$\frac{\partial \psi}{\partial t} = A\psi \tag{9}$$

$$\approx \frac{\psi^{k+1} - \psi^{k}}{\Delta t} = A\psi^{k+1} \tag{10}$$

approximating the exponential matrix operator $e^{\mathbf{A}\Delta t}$ as $(\mathbf{I} - \Delta t\mathbf{A})$. Within one time-step, the problem can be approximated as steady-state. Energy would typically be discretized by group, but in the interest of focusing on the software design rather than on the ins-and-outs of developing transport code, I chose to focus on solving one-speed problems. Angle is discretized using an S_N Gauss-Legendre quadrature set, meaning that ψ is found in N discrete directions, and scalar flux ϕ is approximated as the weighted sum $\sum_{m=1}^{N} w_m \psi_m(x)$ rather than the integral over all spherical angles. Equation 1 can then be written in a simplified and discrete form,

$$\mu_n \frac{\partial \psi_n}{\partial x}(x) + \sigma_t \psi_n(x) = Q(x) \tag{11}$$

where $Q(x)$ is the source term and μ_n is the cosine of the S_N angle.

The problems I focused on are one-dimensional slabs. I discretized in space with the commonly-used diamond-difference numerical scheme, assuming the angular flux at the midpoint of one spacial cell is the average of the angular fluxes at the left and right boundaries of the cell, and the spatial derivative is approximated using central-differencing. All data are then assumed to describe the cell midpoint. This isn't entirely accurate; cell-averaged angular flux is actually the integral of the angular flux across the whole cell. By discretizing with this diamond difference assumption instead of using a continuous integral to solve for the cell-averaged angular flux, we introduce some discretization error. Applied to Equation 11, this gives us

$$\mu_n \left[\psi_n(x_{i+1/2}) - \psi_n(x_{i-1/2})\right] + \sigma_{t,i}\psi_{n,i}\Delta x_i = \Delta x_i Q_i. \tag{12}$$

Depending on the direction μ_n, we can sweep either left or right, starting with a boundary condition and calculating ψ_n at the next boundary across the slab. After looping through all angles, the scalar flux can be calculated as a weighted sum.

$$\psi_{n,i} = \frac{1}{2}\left(\psi_n(x_{i-1/2}) + \psi_n(x_{i+1/2})\right) \tag{13}$$

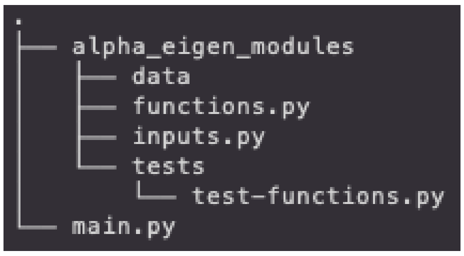
The basic scheme to solve the problem is source iteration, in which a guess is made for the scalar flux ϕ, that guess is used to calculate the source term Q, and that source term is used to calculate the angular flux at each cell boundary ψ. The angular fluxes can then be summed to get out a new scalar flux, which will either have converged to a correct answer or will be used to calculate a new source term until convergence is reached.

Once angular flux has been computed at every time step, it can be appended together to create the left-hand-matrix $\mathbf{\Psi}_{-}$, and summed to create the scalar-flux equivalent $\mathbf{\Phi}-$. Singular-value decomposition is performed on both, surrogate matrices formed, and the alpha-eigenvalues from both can be compared.

3 Implementation

3.1 Structure and usage

The alpha-eigen package was designed to neatly compartmentalize the needed structures and functions to perform radiation transport and dynamic mode decomposition. The primary functionality is to, given a specified problem type, perform a deterministic radiation transport simulation either for a steady-state k-eigenvalue solution, or a time-dependent flux solution and alpha-eigenvalues. The main package, a simplified version of which is shown in Figure 1, contains `alpha_eigen_modules` and `main.py`. `alpha_eigen_modules` is comprised of the `data` sub-directory, which stores `hdf5` files of nuclear data for the various included problems; functions files exemplified by `functions.py` and their accompanying tests; and `inputs.py`, which stores the outermost functions that can be called to run a particular problem type.



```
.
├── alpha_eigen_modules
│   ├── data
│   ├── functions.py
│   ├── inputs.py
│   └── tests
│       └── test-functions.py
└── main.py
```

Figure 1: Package structure.

From the main `alpha-eigen` directory, `main.py` can be run from the command-line with a series of arguments to help establish the desired geometry. The only required parameter is 'problem,' which specifies which of the included problem types the user would like to run. For example, `python3 main.py --problem kornreich_parsons_symmetric` will run the symmetric one-speed slab benchmark from [2]. All available command-line arguments are:

- `--problem` Required. Specifies problem-type to be built and run.
- `--length` Optional. Slab length in units of mean-free-path.
- `--cells_per_mfp` Optional. The number of cells into which each mfp is discretized, so the total number of spacial cells is length × cells_per_mfp.
- `--num_angles` Optional. The number of angles over which to discretize.
- `--steps` Optional. The number of time-steps over which to calculate the time-dependent fluxes.

If optional arguments aren't specified, each input has a default steady-state configuration. The default output for a steady-state problem is a command-line print of k-eigenvalue as it converges. The default output for a time-dependent problem is a command-line print of progress through time-steps, and a final output of the calculated alpha-eigenvalues. If a user desired, they could also run through an IDE and have access to other information such as flux throughout the problem.

3.2 Classes

The package relies on classes to contain all of the necessary problem data and organize relevant functions. When `main` is called with a `--problem`, the associated input function will run. If they were so-inclined, a user could build their own input function and add the option to main. Once initiated, all inputs will call classes and their methods in the following structure.

SimpleMaterial() The first thing an input does is define all of the materials in the problem, each of which is an instance of SimpleMaterial(). SimpleMaterial() contains the material name, the number of energy groups the problem is discretized into, and nuclear data such as cross-sections.

SimpleMaterial() is primarily intended for use with the 'toy' problems defined in the package. The sub-class Material() builds on this by reading from hdf5 files in the `data` directory.

Slab() Once the materials have been defined, one instance of the Slab() class is instantiated to store data that's constant throughout the problem like the number of cells, the slab length, the number of angles, as well as all of the materials contained in the problem so that as long as Slab() is accessible, so is the geometric definition of the entire problem. Slab() is intended for homogeneous problems, and the sub-classes SymmetricHomogeneousSlab() and MultiRegionSlab() build on this by specifying problem configurations of increasing complexity.

OneGroup(), Zone(), and OneDirection() With the problem defined, we can actually begin running the transport simulation. OneGroup(), Zone(), and OneDirection() represent discretizations in energy group, space, and angle, respectively. Of the three, OneGroup() is the outer-most class which contains steady-state functions for performing power-iterations and source iterations to converge on a solution for scalar flux. Within OneGroup(), there are instances of Zone(), which store zone-dependent information like where the midpoint of a zone is, what material it is, etc. There are also instances of OneDirection(), which primarily handles the sweeps to calculate angular flux. To expand out to multigroup problems, a larger class MultiGroup() is constructed to contain many OneGroup() instances, but this capability has not been validated against other numerical solvers yet.

TimeUnit() For time-dependent problems, the one instance of the TimeUnit() class stores time-dependent data and performs time-dependent functions, often by looping through methods of OneGroup() and OneDirection(), though some time-dependent-specific methods are also included. The method to actually compute the time-eigenvalues is also store in TimeUnit().

3.3 Testing

Throughout development, the package was checked against other numerical solvers for cross-code validation. The main integration test is in the `tests/` directory, and is configured to run with PyTest. The integration test confirms that a steady-state benchmark from Kornreich and Parsons [2] calculates the expected k-eigenvalue, using a smaller number of angles and cells than one might use for realistic problem-solving. A longer test is also available in the `tests/` directory, titled `long_kornreich_parsons.py`; it's not named in the standard PyTest fashion to prevent running the high-fidelity version of this benchmark after every change, but would just need to have 'test_' appended to the filename to be included in the test suite. A `context` file in the `tests/` directory ensures that it has access to all of the necessary modules through references, which can sometimes give PyTest trouble.

3.4 Dependencies

The setup.cfg file is set to find all required dependencies, in case a future developer were to add more and not add it explicitly to the requirements file. The required modules are quite standard - NumPy, Matplotlib, h5py, and tables. PyTest is a required install if the user would like to run integration or unit tests.

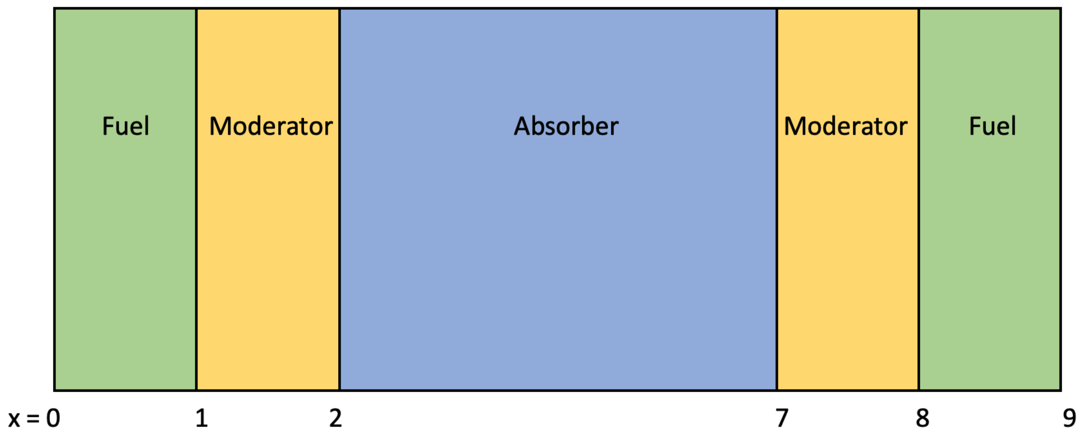


Figure 2: Heterogeneous slab as described in [2].

Table 1: Comparing benchmark results from [2] to results from alpha-eigen.

Eigenvalue mode	$\nu\Sigma_f$	k-eigenvalue [2]	k-eigenvalue [1]	α-eigenvalue [2]	α-eigenvalue [1]
Fundamental	0.3	0.4243163	0.4243178	-0.3196537	-0.3195801
Second	0.3	0.4241317	0.4241312	-0.3229855	-0.3229634
Fundamental	0.7	0.9900716	0.9900791	-0.006156369	-0.006166847
Second	0.7	0.9896407	0.9896412	-0.006440766	-0.006475483

4 Results

The primary reference solutions used to cross-reference are found in [2] and [3]. Both papers contain solutions to benchmarks described in [2], and Ryan McClarren has also provided software used to develop the results in [3], used to cross-reference results not directly reported in the papers, such as fluxes.

A test problem of a heterogeneous slab in Figure 2 comprised of fuel, moderator, and absorber was run in steady-state to confirm properly functioning power- and source-iteration functionality. The scalar flux profile of the problem in steady-state in low-fidelity (50 cells per mfp, N=16) can be seen in Figure 3. Even in this low-fidelity configuration, the two solvers agree, and alpha-eigen converges to the benchmark k-eigenvalue reported in [2].

The heterogeneous slab problem was run using various configurations. Figure 3 shows results for a super-critical case in which the fuel-regions had ($\nu\sigma_f = 0.7, \Sigma_s = 0.8$), the absorber region had ($\nu\sigma_f = 0.0, \Sigma_s = 0.1$), and the moderator had ($\nu\sigma_f = 0.0, \Sigma_s = 0.8$). A subcritical form of the problem was also run, the only change for which is that the fuel $\nu\sigma_f = 0.3$. Table 1 compares the benchmark results for both the super- and sub-critical case to the results from alpha-eigen. Most of the results match up to 10^{-5}, which is what [3] also reports, though in my results some of the α-eigenvalue results are only within 10^{-4}.

Unfortunately, I didn't end up meeting my goal of matching the alpha-eigenvalues using scalar flux. More theoretical work is needed to understand what causes the difference and if mitigating that is possible. In terms of the software, the capability is there.

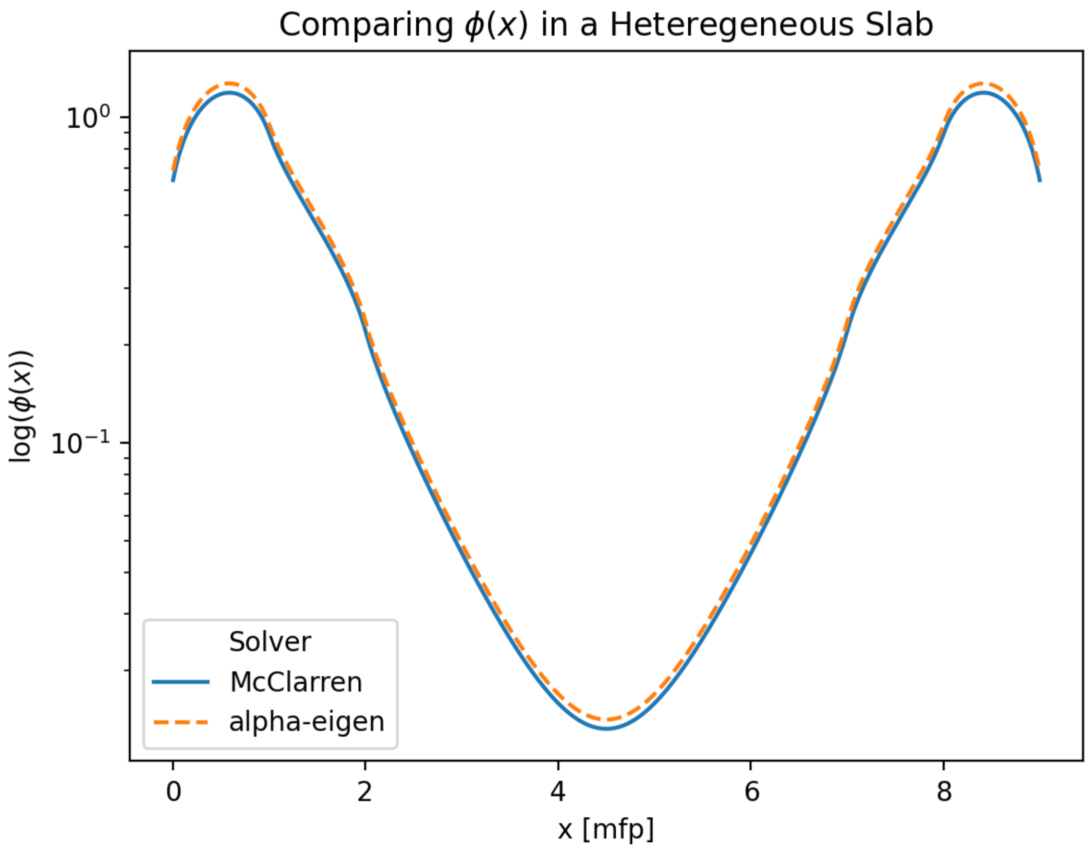


Figure 3: Comparing scalar flux across a heterogeneous slab as described in [2]. Code comparison between alpha-eigen and the code used to develop [3].

5 Conclusions and Future Work

This software package was created to build on work developed in [3], and attempt to apply the DMD method developed there on scalar flux rather than angular flux. Though that goal did not end up met, the infrastructure built with this package has made testing and development much more streamlined. I hope to continue by meeting the goals listed in my proposal - integrating scalar flux tally results from MCDC, and perhaps integrating testing between alpha-eigen and code developed for [3] to streamline code comparison.

6 Acknowledgements

This work was supported by the Center for Exascale Monte-Carlo Neutron Transport (CEMeNT), a PSAAP-III project funded by the Department of Energy, grant number: DE-NA003967.

References

[1] Kayla Clements. alpha-eigen, March 2022.

[2] D. R. Kornreich and D. Kent Parsons. Time-Eigenvalue Calculations in Multi-Region Cartesian Geometry Using Green's Functions. *Annals of Nuclear Energy*, 32:964–985, 2005.

[3] Ryan G. McClarren. Calculating Time Eigenvalues of the Neutron Transport Equation with Dynamic Mode Decomposition. *Nuclear Science and Engineering*, 193:854–867, 2019.

[4] Peter J. Schmid. Dynamic mode decomposition of numerical and experimental data. *Journal of Fluid Mechanics*, 656:5–28, 2010.