ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

PAUL SCHERRER INSTITUT
PSI

Physics-informed neural networks (PINN) with a mathematically informed architecture for the nuclear decay equation

Semester Project

Department of Physics

ETH Zurich

Miha Pompe

supervised by

Dr. A. Adelmann

in collaboration with

A. Albà
Dr. R. Boiger

May 29, 2023

Abstract

In this semester project we propose a novel method for calculating the time evolution of nuclide concentration in nuclear waste, which is crucial for safe nuclear waste disposal and accurate simulations of nuclear cores. The method utilizes physics-informed neural networks (PINNs) with a mathematically informed architecture. The time evolution of concentration is governed by the decay equation, and the complexity of the problem arises from the stiffness and size of the decay matrix. Traditional numerical methods, such as CRAM, Páde, and Runge-Kutta, have limitations in solving stiff systems of equations with a large number of nuclides. The proposed method formulates the problem as a neural network task and uses a mathematically-informed architecture to capture the decay behavior. The implementation involves defining a special loss function that combines the differential equation and initial condition terms, with a weight ratio parameter to adjust the network's focus. In the project we address the application of the method to both decay and burnup problems, considering different network architectures and constraints. Additionally, performance improvements are explored, including gradual increase of stiffness, gradual expansion of training data, and dynamic weight ratio. The proposed method offers a promising approach to accurately and efficiently calculate the time evolution of nuclide concentration in nuclear waste, addressing the challenges posed by the size and stiffness of the decay matrix.

Contents

1 Introduction

Nuclear waste comprises of up to a couple thousand different nuclides, whose concentration constantly changes due to nuclear reactions. For purposes of safe nuclear waste disposal it is important to know how this concentration changes over time, so that we can avoid overheating the waste. Fast and accurate knowledge of concentrations is also needed in simulations of nuclear cores. The goal of this project is to develop and evaluate a novel method for calculating the time evolution of concentration.

The time evolution of the nuclide concentration is governed by the decay equation, given some initial conditions:

$$\frac{d\mathbf{N}(t)}{dt} = A\mathbf{N}(t)\,, \qquad \text{with} \quad \mathbf{N}(t=0) = \mathbf{N}_0\,, \tag{1}$$

where $\mathbf{N}(t) \in \mathbb{R}^n$ is the vector of concentrations at time t, $n \in \mathbb{N}$ is the number of nuclides, and $A \in \mathbb{R}^{n\times n}$ is the decay matrix. The elements of the A matrix are inversely proportional to the lifetime of one nuclide decaying into another one.

The complexity of this problem arises from the stiffness and size of the decay matrix. It is common that the number of nuclides ranges from $n = 2000$ to $n = 4000$, and that the stiffness of A matrix goes up to $\kappa(A) \approx 10^{20}$. Stiffness is defined as the ratio between the biggest and the smallest eigenvalue of the matrix, $\kappa(A) = \frac{|\operatorname{Re}\lambda_{max}|}{|\operatorname{Re}\lambda_{min}|}$.

In this project we propose a novel approach to solving the nuclear decay equation using physics-informed neural networks (PINNs) with a mathematically informed architecture.

1.1 Decay equation solution

Given a radioactive nuclide one can show experimentally that the sample activity A is proportional to the number N of nuclides in the sample, $A = \lambda N$. Activity is defined as the number of decays per second, $A\Delta t = N(t) - N(t+\Delta t)$, which in the limit of small Δt becomes $A = -\frac{dN}{dt}$. If we combine the above equations we get to the decay equation for one nuclide and it's solution:

$$-\frac{dN}{dt} = \lambda N, \quad N(t) = N(0)e^{-\lambda t},$$

where we can define $\lambda = \frac{1}{\tau}$, where τ is the lifetime of the nuclide.

Instead of examining the decay on just one nuclide, we are usually interested in the decay chain of a nuclide. When a radioactive decay of a nuclide feeds the decay of another nuclide we call this a decay chain:

$$\mathcal{A} \xrightarrow{\lambda_1} \mathcal{B} \xrightarrow{\lambda_2} \ldots \xrightarrow{\lambda_{n-1}} \mathcal{Z}$$

To solve this problem we use the Bateman equation. The Bateman equation is a model describing how the concentrations in a decay chain vary as a function of time, given decay rates and initial abundances:

$$\begin{aligned}
\frac{dN_1}{dt} &= -\lambda_1 N_1\,,\\
\frac{dN_i}{dt} &= \lambda_{i-1}N_{i-1} - \lambda_i N_i\,,\\
\frac{dN_n}{dt} &= \lambda_{n-1}N_{n-1},
\end{aligned}$$

where $i \in [1, n]$, $N_i(t)$ is the concentration of nuclide i at time t, λ_i is the decay constant for the decay of nuclide N_i into N_{i+1}. Assuming $N_1(0) \neq 0$ and $N_i(0) = 0$ for $i > 1$, a general solution can be found by employing the Laplace transform to get to [1]:

$$N_i(t) = N_1(0) \left(\prod_{j=1}^{i-1} \lambda_j \right) \sum_{j=1}^{i} \frac{e^{-\lambda_j t}}{\prod_{k=1, k\neq j}^{i} (\lambda_k - \lambda_j)}.$$

We can rewrite the problem above in a more general form by using the matrix A as defined in the introduction. A matrix is two diagonal for the Bateman problem. In general we are interested in solving a problem with multiple decay chains in which case the A matrix can be permuted to a triangular matrix, with real eigenvalues. We will call this the decay problem. For cases that include loops in the decay chains the A matrix cannot be permuted to a triangular matrix and can have complex eigenvalues. Such a matrix is called a burnup matrix. The diagonal matrix elements correspond to the total loss rates, whereas the off-diagonal elements correspond to production rates.

A simple solution to equation (1) is to take the exponential of the matrix A:

$$\mathbf{N}(t) = e^{At}\mathbf{N}(0),$$

where the exponential of the matrix can be expanded as a power series:

$$e^{At} = \sum_{n=0}^{\infty} \frac{1}{n!}(At)^n,$$

with an additional constraint that $A^0 = I$. Numerous numerical methods exist that compute the solution by calculating the exponent of the matrix, which will be covered in the next section.

A general solution of equation (1) is a linear combination of functions of the form [2]:

$$\mathbf{N}(t) = \sum_{i=1}^{n} \mathbf{a}_i \cos(\mathrm{Im}(\lambda_i)t)e^{\mathrm{Re}(\lambda_i)t} + \mathbf{b}_i \sin(\mathrm{Im}(\lambda_i)t)e^{\mathrm{Re}(\lambda_i)t}, \tag{2}$$

where $\mathbf{a}_i, \mathbf{b}_i \in \mathbb{R}^n$ and λ_i is the i-th eigenvalue of the A matrix, $\lambda_i \in \mathbb{C}$. This solution is applicable to problems with distinct eigenvalues, $\lambda_i \neq \lambda_j$ for $i \neq j$. For A matrices with equal eigenvalues power factors of time to the algebraic multiplicity have to be added. If we restrict ourselves to decay matrices which only contain real eigenvalues the solution is a linear combination of exponential functions [2]:

$$\mathbf{N}(t) = \sum_{i=1}^{n} \mathbf{a}_i e^{\lambda_i t}, \tag{3}$$

which again is applicable only to problems with distinct eigenvalues.

1.2 Existing methods for the decay equation

There are several numerical methods that can be used to solve the decay equation, including CRAM, Páde, and Runge-Kutta methods. Each of these methods has its own advantages and drawbacks.

The CRAM method is a type of rational approximation method that uses Chebyshev polynomials to approximate the exponential function [2]. The method is known for its accuracy and stability, making it a popular choice for solving stiff systems of equations. One advantage of

CRAM is that it can be applied to a wide range of decay rates and initial conditions. However, the method can be computationally expensive and may require more iterations to achieve convergence. CRAM method can achieve very high accuracy as long as the eigenvalues of the A matrix are all on the real negative axis. The method becomes less reliable for matrices with more complex eigenvalues.

Páde method is another type of rational approximation method that uses a rational function to approximate the exponential function [2]. The method is known for its simplicity and can be computationally efficient, especially for long-term simulations. However, the accuracy of Páde method may suffer when the decay constant is large, leading to numerical instability.

Runge-Kutta method is a type of explicit method that uses a series of intermediate values to approximate the solution to the differential equation. The method is known for its accuracy and versatility, making it a popular choice for solving a wide range of differential equations. However, Runge-Kutta method can be computationally expensive, especially for high order methods and very stiff problems, since we have to increase the number of time steps. Páde and Runge-Kutta can both solve problems, with both real and complex eigenvalues.

1.3 Neural networks

Neural networks are a type of machine learning model that are inspired by the structure and function of the human brain. They are composed of multiple interconnected layers of artificial neurons that process and transmit information.

The architecture of a neural network can vary depending on the problem it is trying to solve. However, a common type of neural network architecture is the feed-forward neural network. In this architecture, the neurons are organized into layers, with each layer having connections to the next layer. The first layer is called the input layer, the last layer is called the output layer, and the layers in between are called hidden layers. The number of neurons in the input layer depends on the number of features in the input data, while the number of neurons in the output layer depends on the type of problem being solved.

The activation function ($\sigma(\mathbf{z})$) is a non-linear function applied to the output of each neuron. The purpose of the activation function is to introduce non-linearity into the network, allowing it to learn more complex patterns in the data. Some common activation functions include the sigmoid function, the ReLU function, and the tanh function:

$$\sigma(z) = \frac{1}{1+e^{-z}}, \quad \text{ReLU}(z) = \max(0, z), \quad \tanh(z) = \frac{e^{z} - e^{-z}}{e^{z} + e^{-z}},$$

where z is the weighted sum of inputs to the neuron.

The loss function is used to evaluate how well the neural network is performing. It compares the predicted output of the network with the actual output and produces a scalar value that represents the difference between the two. The goal of the training process is to minimize this loss function. Some common loss functions include the mean squared error (MSE) and the cross-entropy loss. The MSE loss is given by:

$$\text{MSE} = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2,$$

where n is the number of data points, y_i is the actual output, and $\hat{y}_i$ is the predicted output. The cross-entropy loss is given by:

$$\text{CE} = -\frac{1}{n}\sum_{i=1}^{n} y_i \log \hat{y}_i + (1 - y_i) \log (1 - \hat{y}_i),$$

where n is the number of data points, y_i is the actual output, and $\hat{y}_i$ is the predicted output.

The weights and biases are the parameters of the neural network that are adjusted during the training process. The weights (w_i) are used to scale the input values, while the biases ($\mathbf{b_i}$) are used to shift the activation function. The weights and biases are initialized randomly to avoid bias towards a certain direction. Common methods of initialization include normal distribution and uniform distribution. The output ($\mathbf{z}$) of the layer $i+1$ would therefore be:

$$\mathbf{z}_{i+1} = \sigma(w_i \mathbf{z}_i + \mathbf{b}_i), \quad \text{where } 0 \leq i < L, \tag{4}$$

whereas the output of the output layer does not include the activation function, $\mathbf{z}_{L+1} = w_L \mathbf{z}_L + \mathbf{b}_L$.

The process of training a neural network involves adjusting the weights and biases to minimize the loss function. This is typically done using backpropagation, a technique that involves calculating the gradient of the loss function with respect to the weights and biases. The gradient is then used to update the weights and biases through an optimization algorithm, such as stochastic gradient descent (SGD). During training, the network is presented with a set of input data and the corresponding target outputs. The input data is fed forward through the network to produce a predicted output. The difference between the predicted output and the actual output is calculated using the loss function, and the weights and biases are updated through backpropagation. This process is repeated for multiple epochs until the network converges to a solution with a low enough loss. The hyperparameters of the network, such as the learning rate and number of epochs, can be tuned to optimize the training process.

1.4 Physics-informed neural networks (PINNs)

Physics-informed neural networks (PINNs) are a type of neural network that incorporate physics-based constraints into their architecture, allowing for more accurate and efficient predictions [3]. The architecture of a PINN typically consists of two parts: a neural network and a physics-based loss function. The neural network is responsible for mapping the input variables to the output variables, while the physics-based loss function enforces the physical constraints on the system.

The neural network component of a PINN for the decay equation can be any type of neural network, such as a feed-forward neural network or a convolutional neural network, but in this work we will focus on the former. It consists of layers of neurons that take in the input variables (such as the time) and output a predicted value (the concentration of nuclides at a given time). Each layer consists of a set of neurons, which are connected to the neurons in the previous layer through weighted connections.

The physics-based loss function is what sets PINNs apart from traditional neural networks. It is designed to enforce physical constraints on the system being modeled. For the decay equation, we can use the original differential equation as the physics-based loss function:

$$\mathcal{L}_{ODE} = \left| \frac{d\mathbf{N}(t)}{dt} - A\mathbf{N}(t) \right|^2 .$$

In our case we also want the solution to obey the initial conditions, which we can enforce by adding another loss function:

$$\mathcal{L}_{IC} = \left| \mathbf{N}(0) - \mathbf{N}_0 \right|^2 ,$$

where $\mathbf{N}(0)$ is the model prediction at time $t = 0$ and $\mathbf{N}_0$ is the initial concentration of nuclides.

The physics-based loss function is added to the overall loss function of the network, and the weights of the network are adjusted to minimize both the prediction error and the physics-based loss. In this work we will use only the physics loss function to train the model.

Training a PINN involves two steps: generating training data and optimizing the network parameters. The training data is generated by randomly sampling the input variables and calculating the corresponding output variables using the physics-based constraints. For the decay equation, we can use numerical methods to generate the training data. In our case data generation will not be employed as we will only rely on the physics-based loss function. The network parameters are optimized using an optimization algorithm, which adjusts the weights and biases of the network to minimize the overall loss function.

2 Implementation

2.1 Loss function and performance metrics

Our goal is to create a network that takes as an input time t and returns the concentration of radioactive nuclides at that time $\mathbf{N}(i) \in \mathbb{R}^n$, such that the solution satisfies the differential equation (1) and initial conditions. This will be enforced by the two terms in the loss function:

$$\mathcal{L}_{ODE} = \frac{1}{N_t} \sum_{i=1}^{N_t} \left| \frac{d\mathbf{N}}{dt}(t_i) - A\mathbf{N}(t_i) \right|^2,$$
$$\mathcal{L}_{IC} = |\mathbf{N}(0) - \mathbf{N}_0|^2,$$

where N_t is the number of time data points used for training and $\mathbf{N}_0$ is the initial concentration of nuclides. The total loss is a weighted sum of both losses:

$$\mathcal{L} = \frac{w_{ODE}\mathcal{L}_{ODE} + w_{IC}\mathcal{L}_{IC}}{w_{ODE} + w_{IC}}.$$

Since the total loss $\mathcal{L}$ is invariant under scaling of the weights we can set $w_{IC} = 1$ without loss of generality. Let us define the weight ratio as $w = w_{IC}/w_{ODE}$, which is now a hyperparamter of the network. Changing w therefore changes the focus of the network to either give more importance to the initial conditions ($w >> 1$) or to the differential equation ($w \approx 1$).

It is desired that the network is able to accurately and quickly solve big and stiff problems, for example $n = 2000$ and $\kappa(A) = 10^{20}$. To evaluate the performance of the model adequate performance and error metrics are needed. Intrinsic model metrics that we will use are $\mathcal{L}$, $\mathcal{L}_{IC}$, and $\mathcal{L}_{ODE}$. But to verify our results we need external reference results which will either be analytical or numerical solutions (CRAM method). Let's define the total absolute error and the final absolute error as:

$$\Delta N = \sum_{t=0}^{t_{max}} \left| \hat{\mathbf{N}}(t) - \mathbf{N}(t) \right|,$$
$$\Delta N(t_{max}) = \left| \hat{\mathbf{N}}(t_{max}) - \mathbf{N}(t_{max}) \right|,$$

where $\hat{\mathbf{N}}$ is the reference solution and t_{max} the upper time limit on the training data points, $t \in [0, t_{max}]$. Let's also define the total relative error as $\delta N = \Delta N / \sum_{t=0}^{t_{max}} |\hat{\mathbf{N}}(t)|$ and final relative error as $\delta N(t_{max}) = \Delta N(t_{max})/|\hat{\mathbf{N}}(t_{max})|$.

2.2 Decay problem

The decay matrix $A \in \mathbb{R}^{n \times n}$ is a matrix with real eigenvalues λ_i and if those are distinct the solution can be written as a linear combination of exponential functions $\{\mathbf{a}_i e^{\lambda_i t}\}$, as described

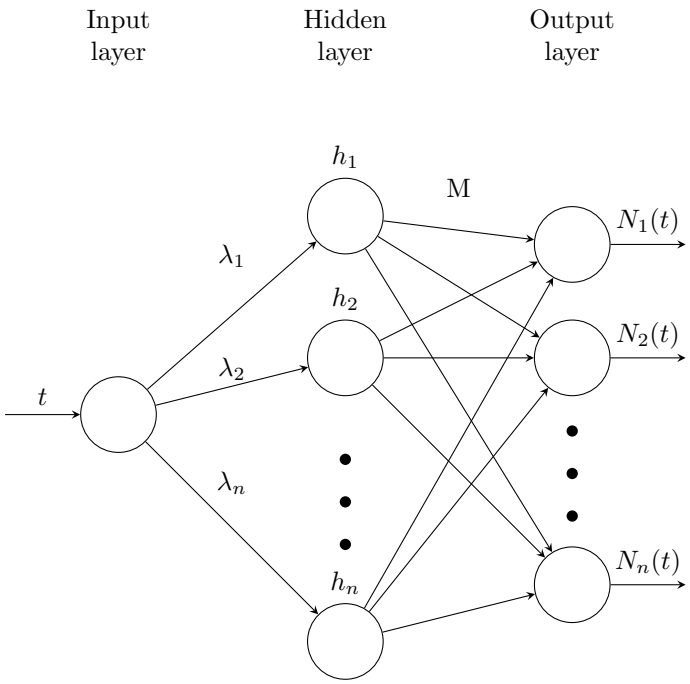


Figure 1: PINN architecture for solving the decay problem. The network consists of three layers. The input layer has one node whose input is time, the weights of this layer a fixed and set to the eigenvalues of matrix A, λ_i. The hidden layer has n nodes that connect to every other of the n nodes in the output layer. The goal of the network is to learn the weight between the hidden and the output layer. The outputs of the network are concentrations at time t, $N_i(t)$.

in (3). We can reformulate this solution in terms of a neural network with one hidden layer with n neurons, an input layer with one neuron and an output layer with n neurons. The activation function of the network is $\sigma(z) = e^z$, with known weights of the input layer $\mathbf{w}_1 = (\lambda_1, ..., \lambda_n)$. The output of the input layer is therefore $h_i = e^{\lambda_i t}$. All the biases in the network are set to zero and are not updated. Given this setup we can write the solution as:

$$\mathbf{N}(t) = M\mathbf{h}, \quad \text{with } \mathbf{h} = (e^{\lambda_1 t}, ..., e^{\lambda_n t}).$$

Therefore the network only needs to learn the coefficients of matrix $M \in \mathbb{R}^{n\times n}$, which correspond to the coefficients $\mathbf{a}_i$ in the linear combination. A graphic representation of the network is shown in Figure 1. The network needs to learn n^2 parameters and we have also constrained the size and shape of the network. Compared to regular neural networks far fewer parameters need to be learned and we do not have to perform a hyperparameter search to find the optimal network architecture.

Besides fixing the weights of the input layer and removing biases we can also constrain the weights of the hidden layer (coefficients of matrix M). Every decay matrix can be permuted into a triangular matrix. It is allowed to simultaneously swap of two columns and two rows with the same indices. By applying this operation multiple times we can permute the matrix into lower or upper triangular. In our case we choose lower triangular, but the choice is arbitrary. Given a lower triangular matrix A we can constrain the matrix M to be lower triangular. Such condition

can be enforced by adding a new term to the loss function that penalizes non-triangular matrices:

$$\mathcal{L} = \sum_{i=1}^{n} \sum_{j>i}^{n} |M_{ij}|^2.$$

A better approach is to initialize M as lower triangular and at each training step set the gradients of the upper triangle to zero. This approach is better because the network does not have to learn the coefficients of the upper triangle, thereby reducing the number of parameters to $\frac{n(n+1)}{2}$.

Due to measurement error of matrix A it is not uncommon for two eigenvalues to be the same, which poses an issue since the basis functions are not the solutions of the problem. Additionally given two equal eigenvalues two outputs of the input layer would be indistinguishable for all times, the solution wouldn't be unique. The solution is to slightly change the eigenvalues, such that the difference is smaller than the measurement error. A problem also arises when the eigenvalue is zero, in this case the particular nuclide does not contribute to the reaction and can therefore be excluded, the row and column of the nuclide are removed.

2.3 Burnup problem

The burnup matrix A consists of eigenvalues with a complex part, therefore the solution is no longer a linear combination of exponential functions but rather a combination of exponential function paired with sines and cosines, as described in equation (2). The network architecture is therefore more complex. Instead of using one network, two networks will be used, one to calculate the linear combination of cosines (Figure 2 blue box) and the other of sines (Figure 2 red box), such that the output of the two input layers will be:

$$\begin{aligned} h_i &= \cos(\mathrm{Im}(\lambda_i)t)e^{\mathrm{Re}(\lambda_i)t}, && i \in [1, n] \\ h_{n+i} &= \sin(\mathrm{Im}(\lambda_i)t)e^{\mathrm{Re}(\lambda_i)t}, && i \in [1, n]. \end{aligned}$$

The network then feeds into two output layers, where the network will learn the matrices $M, M' \in \mathbb{R}^{n\times n}$, $\tilde{N}_j = M_{ij}h_i$ and $\tilde{N}_{n+1} = M'_{ij}h_{n+i}$. The number of parameters is $2n^2$. The last part (merging layer) is a pairwise summation of sine and cosine parts, $N_i = \tilde{N}_i + \tilde{N}_{n+i}$.

The network weights can be further constrained. For real eigenvalues h_{n+i} will be equal to zero, therefore they provide a source of instability in the network. To fix this we can set the i-th row and column in matrix M' to zero and constantly set the gradient of these elements to zero.

2.4 Performance improvements

In order to improve the performance of the model the following techniques have been tested.

LBFGS (limited-memory BFGS) optimizer was chosen instead of more common ones like stochastic gradient descent or Adam. The impact of the optimizer was already studied in [5]. LBFGS was chosen due to faster convergence time and in order to eliminate a hyperparameter, the learning rate.

Gradual increase of stiffness. Solving the problem is easier when the stiffness of matrix A is low. Therefore the proposed method is to gradually increase the stiffness of matrix A. The network would be first trained on the matrix A with reduced stiffness. These weights would be then used as initial weights when training the model on a bit stiffer matrix. This would then be repeated until we come to the original matrix. In order to adjust the stiffness of the matrix every eigenvalue (λ_i) was first scaled such that $\tilde{\lambda}_i = \left(\frac{\lambda_i - \lambda_{min}}{\lambda_{max} - \lambda_{min}}(S-1) + 1\right)\frac{\lambda_{min}}{\lambda_i}$, where S is the desired stiffness, $\lambda_{min} = \min \lambda_i$ and $\lambda_{max} = \max \lambda_i$. The columns of matrix A are then

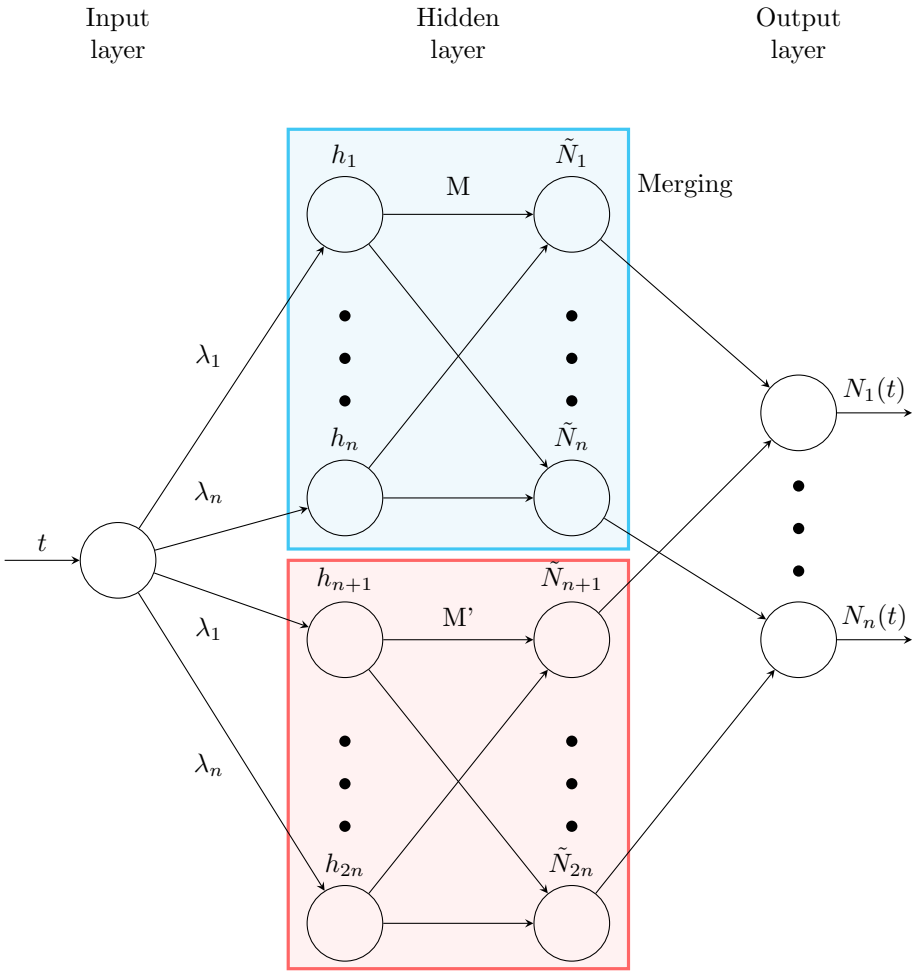


Figure 2: PINN architecture for solving the burnup problem. The network consists of two subnetworks that are similar to those used for the decay problem. The activation function for the blue network is given by equation **??** and for the red network by equation **??**. The output of both networks is summed pairwise.

scaled by the scaled eigenvalues, $\tilde{A}_j = A_{ij}\tilde{\lambda}_i$. Tests were performed with a matrix of stiffness 10^{16}. Firstly the stiffness was reduced to 10^{10} and then after each training increased by one magnitude. The resulting error was 8% lower compared to not using this method.

Gradual expansion of training data. The training data of the model is a list of times, $t_i \in [0, t_{max}]$. In this method we gradually increase t_{max} every epoch up to a desired value, $t_{max}(epoch) = t_{max}\frac{epoch}{v_{epoch}}$ [4]. Doing so allows the model to first learn the initial conditions and the behavior of quickly decaying nuclides and only afterwards it will focus on the rest of the time steps. Implementing this method reduced the relative final error at most by 18 % compared to not using this method. The effect of the method depends on the speed of increase of t_{max}, which is set by v_{epoch}. The average reduction for different v_{epoch} of relative final error is 8,1 %.

Dynamic weight ratio. The weight ratio determines which loss ($\mathcal{L}_{IC}$ or $\mathcal{L}_{ODE}$) the network should focus on. Normally this ratio is determined by the user, such that the optimal result is achieved, and it does not change with each epoch. This method proposes a use of a function, $f(epoch)$, to dynamically change the weight ratio. The function always started with a given ratio $w = w_{IC}/w_{ODE} = w_0$. The following functions were tested:

- Step function
$$f(epoch) = \begin{cases} w_0, & epoch \leq \alpha, \\ w_1, & epoch > \alpha, \end{cases}$$
where w_1 is the final desired ratio and α is the cutoff point. $w_1 < w_0$ since we first want the network to learn the initial conditions and then the differential equation.

- Linear function
$$f(epoch) = \begin{cases} w_0 - (w_0 - w_1)\frac{epoch}{\alpha}, & epoch \leq \alpha, \\ w_1, & epoch > \alpha. \end{cases}$$

- Geometric decrease, $f(epoch) = w_0\beta^{epoch}$, where β is the decrease rate, $\beta < 1$. The problem with this method is that if the network does not learn the initial conditions in the beginning, it will never be able to do so. The output is very dependent on the choice of β.

- The final function does not depend on *epoch* but rather on the ratio between $\mathcal{L}_{IC}$ and $\mathcal{L}_{ODE}$, such that:
$$f(epoch) = \begin{cases} w_1, & \log(\mathcal{L}_{IC}/\mathcal{L}_{ODE}) \leq \gamma, \\ w_0, & \log(\mathcal{L}_{IC}/\mathcal{L}_{ODE}) > \gamma, \end{cases}$$
where γ is the cutoff ratio that switches between prioritizing $\mathcal{L}_{IC}$ over $\mathcal{L}_{ODE}$ and vice versa. This method allows the network to gradually decrease one loss and then switch to decreasing the other. The process repeats when the other loss gets low enough.

The performance of the model did not improve by employing these functions, even after extensive experimentation with different parameters α, β, and γ. In all cases it was equal or better to set the ratio to a constant value of w_1.

3 Results

The performance of the model will be evaluated on a data set called ENDFB68, a decay matrix benchmark provided by Los Alamos National Laboratory. It includes the decay matrix of 1624 nuclides, with stiffness $\kappa(A) = 9,8 \times 10^{29}$. Since the original matrix is too large for analysis we will define a sub-matrix that will serve as our default test case. The sub-matrix is of size 50 $\times$

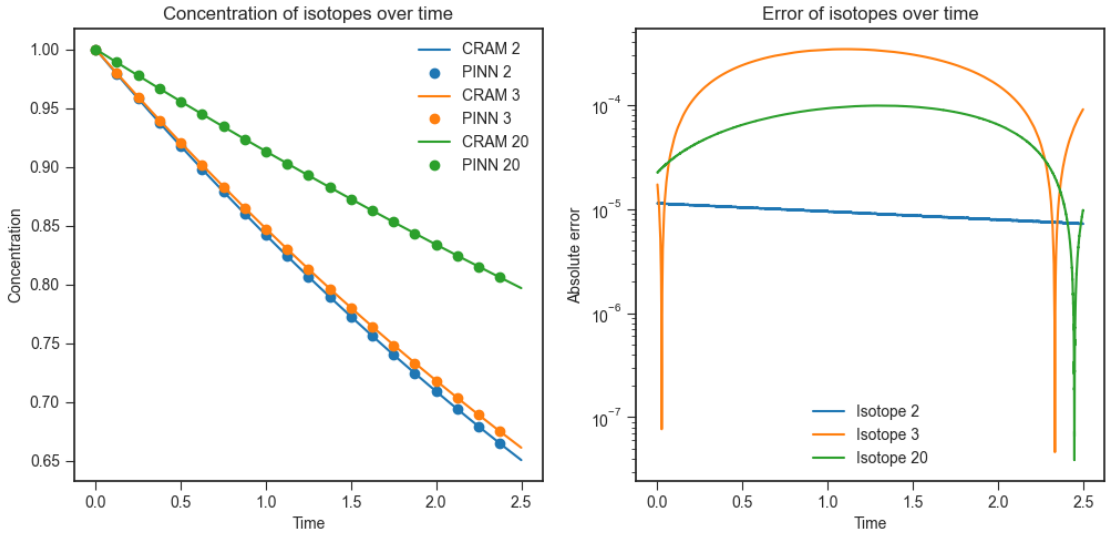


Figure 3: Time evolution of three nuclides for the default test case (decay matrix of size 50, stiffness 10^{16}) with initial conditions set to $N_i(0) = 1$ for all i. The left graph shows how the concentration over time, computed by CRAM and PINN methods. For clarity only a subset of all PINN data points is shown. The left graph shows the absolute difference between PINN and CRAM solution through time for the three nuclides.

50 and has a stiffness of $\kappa(A) = 10^{16}$. The initial conditions will be set to $N_1(0) = 100$ and $N_i(0) = 0$ for $i > 1$, if not stated otherwise. Other initial conditions were also measured and they produced similar results. To measure the accuracy of the model, its result will be compared with the result of the CRAM method. The comparison will be made within the time interval [0, 2.5], with 10.000 time steps.

An example of the network output is shown in Figure 3 for the default case. The initial conditions were set to $N_i(0) = 1$ for all i, in order to show that the network also performs well with other initial conditions. The first graph shows time evolution of concentration for some of the nuclides. We can graphically see that CRAM and PINN solutions match very well. Here we refer to the output of the network as the PINN solution. The second graph shows the absolute error of these nuclides over time, which is always under 10^{-3}. These nuclides were specifically chosen since they change the most throughout the evolution.

The aim of this section is to evaluate the performance limitations of the model under different conditions. The main hyper-parameter of the network is the weight ratio. It can greatly influence the error of the solution, which can be seen from Figure 4. For the best performance of the model it is key to perform a hyper-parameter search to find the best weight ratio. Figure 4 shows how the relative final error changes with weight ratio for the default test case. By choosing a wrong weight ratio the final relative error can be orders of magnitude higher. All following results feature a model whose weight ratio was optimized for lowest error. It is worth noting that there are also other factors that influence model performance such as the initial condition, the seed for random number generators used to assign initial weights, and criteria when to stop the training of the model.

Figures 5a and 5b were generated to analyze the relationship between the performance of a model and the stiffness of matrix A. Figure 5a depicts the model's performance as a function of stiffness, revealing a strong dependence on the stiffness of matrix A. At lower stiffness values, the

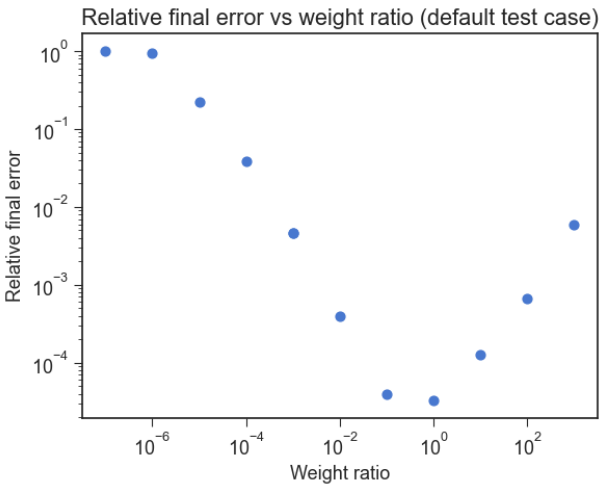


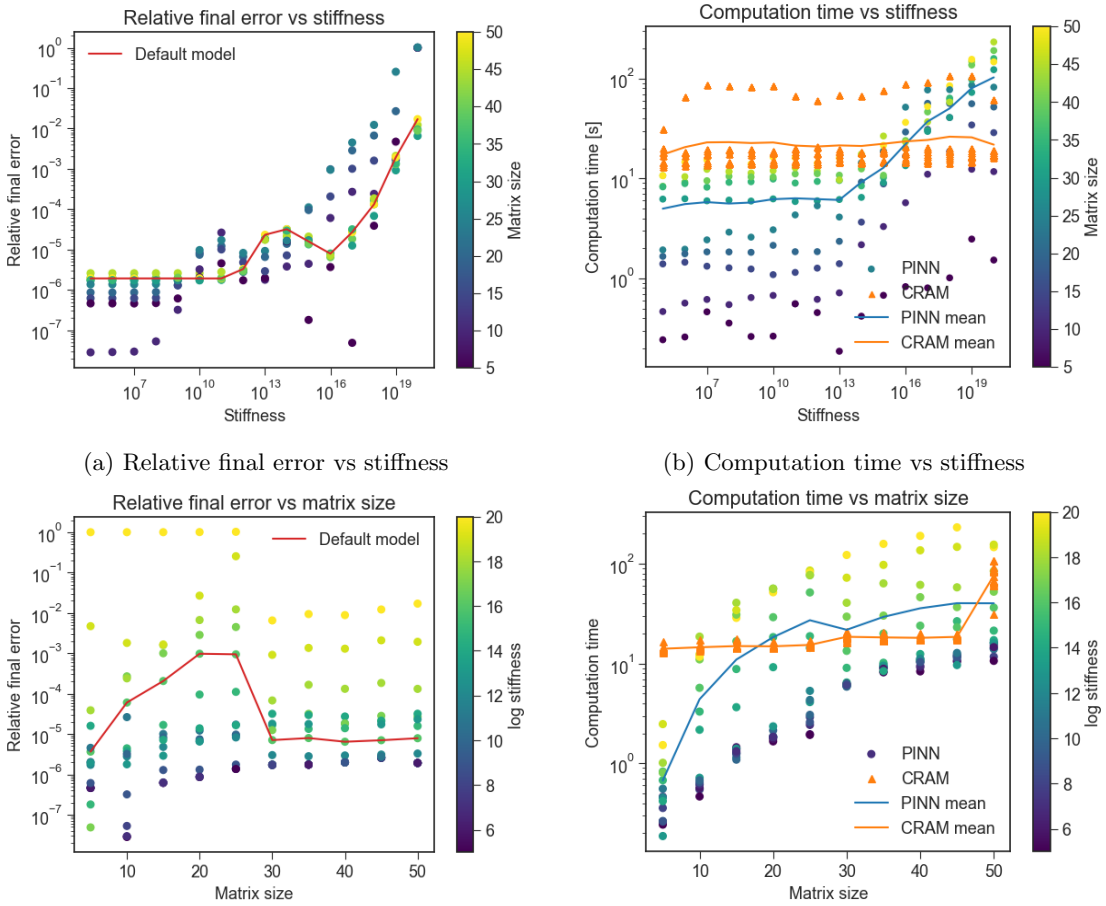
Figure 4: The graph shows how the relative final error changes with different weight ratios, for the default test case (decay matrix of size 50, stiffness 10^{16}) and initial conditions set to $N_1(0) = 100$ and $N_i(0) = 0$ for $i > 1$.

model's performance remains unaffected or is slightly increasing with stiffness. However, there is a noticeable cutoff point where the relative error suddenly jumps to more than 1, occurring at a stiffness of approximately 10^{19}. We presume that this abrupt increase in error is due to the numerical limits of the variables utilized. These conclusions are similar for all analyzed matrix sizes. Figure 5b showcases that the computation time of the CRAM method is irrespective of the stiffness. This is similar for the PINN method, when the stiffness remains below the aforementioned cutoff value. Above that the computation time starts increasing and surpassing the CRAM method. The increase in time is due to the network spending more time training in order to meet the stopping criteria. It is important to mention that all the PINN simulations were performed on a CPU. The model can run on a GPU, where the computation time is shorter.

Figures 5c and 5d were analyzed to understand the performance of a system for different matrix sizes. The first notable conclusion is that the system's performance remains relatively constant regardless of the matrix size. However, a sudden improvement in performance is observed between matrix sizes 25 and 30. This jump in performance can be attributed to the inclusion of a specific nuclide in the system. This nuclide introduces changes to the dynamics of the system, resulting in a marked improvement in performance. Another important observation is that as the matrix size increases, the computation time also increases. Performance analysis past matrix size 50 shows that this increase in computation time follows a linear trend for both the PINN and CRAM methods when the matrix size is greater than 45. On the other hand, for matrix sizes below 45, the computation time for the CRAM method remains constant.

All the aforementioned analysis has been done for the decay problem, but could have also been done for the burnup case. Figure 6 shows the final concentration and final absolute error per nuclide. In this case the burnup matrix contains 442 nuclides, stiffness $\kappa(A) = 7 \times 10^{11}$, and the initial conditions contain 3 nuclides with nonzero concentrations. Due to the large size of the matrix I only trained the network for 5 epochs due to time constraints. This and the limitations with numerical precision are the likely cause for the error.

The computation time of similar test cases is longer for the burnup problem as compared to the decay problem, which is due to a greater amount of parameters to train. With larger



(a) Relative final error vs stiffness

(b) Computation time vs stiffness

(c) Relative final error vs matrix size

(d) Computation time vs matrix size

Figure 5: Graphs in the left column shows how the relative final error changes with stiffness and matrix size. The red line represents how the default test case performs. The color of markers is proportional to matrix size and the log of stiffness respectively. All the tests were performed with the decay matrix. The right column shows how the computation time changes with stiffness and matrix size, for CRAM (orange triangles) and PINN (colored dots) methods. The lines represent the mean of computation time with respect to stiffness and matrix size. The colors scheme is the same as in the left column.

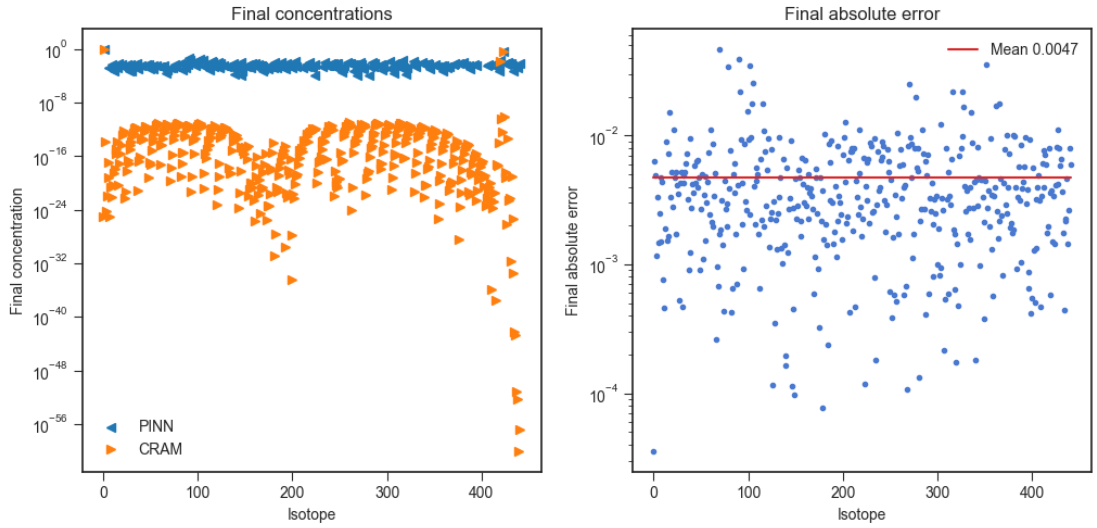


Figure 6: Performance of the model for the burnup problem, for a matrix with size 442 and stiffness 7×10^{11} and initial conditions that contain 3 nuclides with nonzero concentrations. The left graph shows the final concentration of all nuclides calculated with PINN and CRAM methods. The left graph shows the absolute difference between PINN and CRAM final solutions. The red line shows the mean final absolute error, which is 0,0047. Due to the size of the matrix only 5 epoch of training were performed. If the training was left on for more time better results would have been achieved.

matrices the computation time for calculating the eigenvalues of the matrix gets larger, but this increase is negligible to the time increase of the PINN method.

4 Conclusion

The goal of this project was to implement and test a novel mathematically informed PINN method to solve the decay equation. In the end we were able to solve the decay equation with both decay and burnup matrices, for matrices with stiffnesses up to about $\kappa(A) = 10^{19}$ and matrices of any size. The latter claim was tested on a burnup matrix of size 442. PINN method is faster than CRAM for matrices with stiffness under 10^{16}, with relative final error of around 10^{-5}. CRAM method is more accurate than PINN. But if we compare the PINN method to a general purpose PINN, such as presented in [5], the method performs better both in terms of accuracy and computation time. It is also able to solve bigger and stiffer problems compared to general purpose PINNs.

In future research, there are three main areas of focus to enhance the existing work. Firstly improving performance at higher stiffness levels, which can be done by adding support for variables with greater numerical precision, secondly on decreasing the use of memory for better efficiency when dealing with bigger matrices, and thirdly on finding an optimal way of setting the weight ratio.

References

[1] J. Cetnar *General solution of Bateman equations for nuclear transmutations.* Annals of Nuclear Energy (2013) https://www.sciencedirect.com/science/article/pii/S0306454906000284#:~:text=The%20Bateman%20equations%20for%20radioactive,decay%20constant%20of%20ith%20nuclide.

[2] M. Pusa and J. Leppänen. *Computing the Matrix Exponential in Burnup Calculations.* Nuclear Science and Engineering: 164, 140–150, (2010) https://www.tandfonline.com/doi/abs/10.13182/NSE09-14

[3] M. Raissi, P. Perdikaris, and G. E. Karniadakis. *Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations.* Journal of Computational Physics, 378:686–707, (2019) https://www.sciencedirect.com/science/article/pii/S0021999118307125

[4] H. Baty. *Solving stiff ordinary differential equations using physics informed neural networks (PINNs): simple recipes to improve training of vanilla-PINNs.* (2023)

[5] G. Pacifico and A. Adelmann, *Solving the Decay Equation using a Physics Informed Neural Network (PINN).* (2023)