Numerical Methods for the Point Kinetics Equations

Bachelor's Thesis in
Mathematical Engineering - Ingegneria Matematica

Author: **Alessandro Wiget**

Student ID: 981405
Advisor: Prof. Marco Verani
Academic Year: 2023-2024

Acknowledgements

I want to express my gratitude to my advisor, professor Marco Verani, whose feedback and help has been invaluable for the completion of this thesis.

This endeavour would not have been possible without the emotional support of my parent, Barbara e Francesco, my little sister Annalisa and my girlfriend Valentina, who have always been there to motivate.

Lastly I want to thank all my friends and colleagues which I had the pleasure to meet during these three years at Politecnico di Milano.

Abstract

The Point Kinetics Equations (PKE) represent a fundamental mathematical framework for modeling the behavior of nuclear reactors. Their understanding is fundamental for harnessing nuclear power safely.

This thesis aims to implement various numerical methods from literature that currently are the state of the art. The thesis will focus also on their performance under different reactor condition, such as: constant, linear or sinusoidal addition of nuclear fuel, and it will also cover a real-world scenario, by implementing an adiabatic feedback.

The results of the numerical simulations seek to show the strengths and limitations of each of the considered numerical schemes. The inclusion of the test code in the Appendix 4 provides a basis for further performance analysis on different reactor conditions.

Keywords: Point Kinetics Equations, Numerical Methods, Richardson Extrapolation, MATLAB, inhour equation, PCA, TSE, CORE, BEFD

Abstract in lingua italiana

Le Equazioni della Cinetica Puntuale (PKE) rappresentano un quadro matematico fondamentale per modellare il comportamento dei reattori nucleari. La loro comprensione è essenziale per sfruttare l'energia nucleare in sicurezza.

Questa tesi mira a implementare vari metodi numerici dalla letteratura che attualmente rappresentano lo stato dell'arte. La tesi si concentrerà anche sulle loro prestazioni in diverse condizioni del reattore, come: l'aggiunta costante, lineare o sinusoidale di combustibile nucleare, e coprirà anche uno scenario più realistico, ovvero implementando il comportamento di un reattore con un feedback adiabatico.

I risultati delle simulazioni numeriche cercano di mostrare i punti di forza e le limitazioni di ciascuno degli schemi numerici considerati. L'inclusione del codice di test nell'Appendice 4 fornisce una base per ulteriori analisi delle prestazioni in diverse condizioni del reattore.
Parole chiave: Equazioni della Cinetica Puntuale, Metodi Numerici, Estrapolazione di Richardson, MATLAB, Equazione inhour, PCA, TSE, CORE, BEFD

Contents

Introduction

This thesis will analyze mathematical algorithms used in nuclear physics to approximate the system of ODEs called Point Kinetics Equations. These equations describe the density of free neutrons $N(t)$ in the reactor at a given time and the concentration of precursors within the same reactor, $C_i(t)$. Precursors are the elements through which our thermonuclear reaction passes during fission, an example of this precursors is provided in Figure 1. At any stage, additional neutrons will be produced due to the nuclear fission reaction, by being expelled from the precursors due to high energy collisions. The standard in the literature is to consider 6 precursors, this thesis will do the same, if not otherwise specified:

$$\begin{cases} \dfrac{dN(t)}{dt} = \dfrac{(\rho - \beta)}{\Lambda} N(t) + \displaystyle\sum_{i=1}^{6} \lambda_i c_i(t) \\ \dfrac{dc_i(t)}{dt} = \dfrac{\beta_i}{\Lambda} N(t) - \lambda_i c_i(t) \quad i = 1, ..., 6 \end{cases} \tag{1}$$

As observed from the equations, the linearity or non-linearity of the system depends on the function $\rho(t)$ called reactivity, expressed in dollars (\$), a pure number given by the normalization of the number of free neutrons at a given time with the number of delayed neutrons capable of starting a new thermonuclear reaction $\beta_{eff} = \beta I$ called the importance factor, i.e., the probability that a neutron produced by a subsequent nuclear reaction will generate a new one in turn.

The terms λ_i and β_i are specific to the reactor and the behavior of the considered fuel. Indeed, $\beta_i$is the fraction of neutrons produced by the i-th precursor. The term Λ, on the other hand, represents the neutron generation time for each precursor within the reactor.

As standard in the literature, a multitude of combinations of Λ, λ_i and β_i are used. We will follow the paper [7], which proposes the following values:

$$\lambda = [0.0127, 0.0317, 0.115, 0.311, 1.4, 3.87]$$

$$\beta = [0.000266, 0.001491, 0.001316, 0.002849, 0.00896, 0.000182]$$

$$\Lambda = 0.00002$$

All algorithms described from now on have been implemented in MATLAB and, unless already present in the cited papers, can be found as an appendix to the thesis.

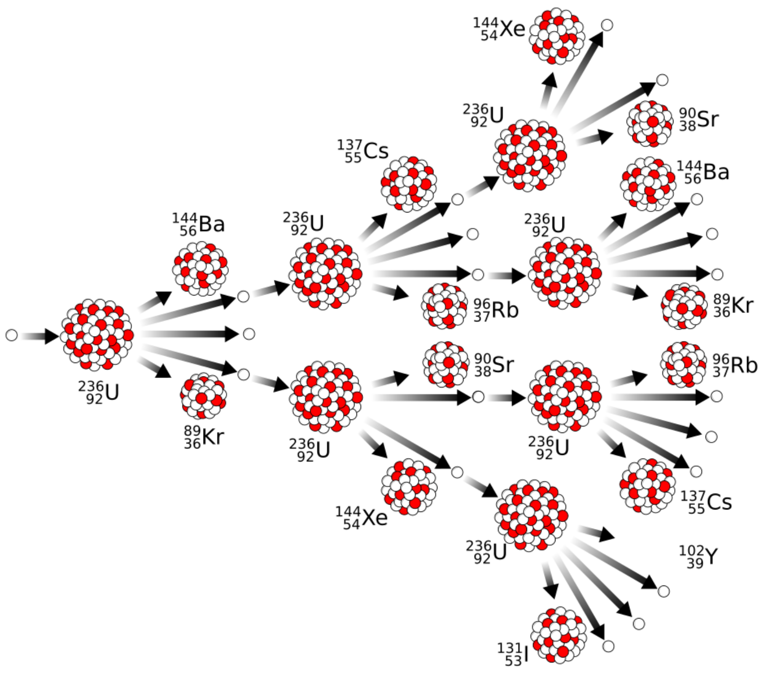

Figure 1: Simplified interactions during nuclear fission.

1 | The Numerical Schemes

1.1. The Piece-wise Constant Approximation Scheme

The first model we will analyze was presented by M. Kinard and E.J. Allen in [4] and is called PCA or piecewise constant approximation.
To solve the system of differential equations with this method, once the desired step size h is defined, the functions $\rho(t)$ and $\vec{F}(t)$ are approximated to constant functions in each of the considered intervals, choosing the value at the midpoint of the interval:

$$\rho(t) \approx \rho\left(\frac{t_i + t_{i+1}}{2}\right) = \rho_i \tag{1.1}$$

$$\vec{F}(t) \approx \vec{F}\left(\frac{t_i + t_{i+1}}{2}\right) = \vec{F}_i \tag{1.2}$$

Once the approximation stage is completed, the problem in each subinterval i takes the form:

$$\begin{cases} \frac{d\vec{x}(t)}{dt} = (A + B_i)\vec{x}(t) + \vec{F}_i \\ \vec{x}(t_i) = \vec{x}_i \end{cases} \tag{1.3}$$

That is, a Cauchy problem that can be solved exactly for each subinterval. To do this, however, it is necessary to find the eigenvalues and eigenvectors of the matrix $A + B_i$. In the literature, however, the eigenvalues are the roots of the *inhour equation*, which can be found by iterations of the Newton method:

$$\rho_i = \beta + \Lambda\omega - \sum_{j=1}^{m} \frac{\beta_j \lambda_j}{\omega + \lambda_j} \tag{1.4}$$

For constant reactivity ρ we find the zeros of 1.4, reported in Table 1.1.

$\rho = 0.003$	$\rho = 0.007$	$\rho = 0.008$
-0.01352232	-0.01311999	-0.01307385
-0.05132931	-0.03956146	-0.03847475
0.13358988	-0.18140456	-0.17877200
-0.00197812	-0.78204969	-0.66916654
-1.15031117	-3.18620888	-2.55452158
-3.72314162	11.75999017	-5.17703320
-200.77787280	-13.33804556	52.85064194

Table 1.1: Solutions of 1.4 for various ρ

Using the method presented in [3], it is possible to find the eigenvectors of the matrix to obtain the change of basis matrices for diagonalization P_i and P_i^{-1}. At this point, the PCA method allows us to obtain a single-step explicit method, easily solvable by any computer.

$$\vec{x}_{i+1} = P_i e^{D_i h} P_i^{-1}[\vec{x}_i + P_i D^{-1} P_i^{-1} \vec{F}_i] - P_i D^{-1} P_i^{-1} \vec{F}_i \tag{1.5}$$

In the paper [4], the convergence of order 2 of the presented method is also demonstrated, which implies the zero stability of the method, by the Lax Richtmyer theorem. To conclude the analysis of this method, I will briefly study the absolute stability of the method.
Let us then consider the model problem in the case of a system of ordinary differential equations:

$$\begin{cases} \dot{\vec{y}}(t) = A\vec{y}(t) \\ \vec{y}(t_0) = \vec{y}_0 \end{cases} \tag{1.6}$$

Since the PCA method 1.5, setting $\vec{F} = \vec{0}$, becomes: $\vec{x}_{i+1} = e^{Ah}\vec{x}_i$, that is, a forward Euler method for ODE systems, whose absolute stability is guaranteed only in the case of all negative eigenvalues of the matrix A, thus the PCA method is a method of limited absolute stability.

Algorithm 1.1 Piece-wise Constant Approximation algorithm:

```
1: Start from the initial conditions of $N(0)$, $C_i(0) \forall i = 1, ..., 6$
2: for $i \in [1 : N]$ do
3:     1. Compute the new $\rho_i$ and $\vec{F}_i$ using 1.1 and 1.2.
       2. Find the roots of the inhour equation 1.4 using $\rho = \rho_i$.
       3. Compute the change of basis matrices for the diagonalization, $P_i$ and $P_i^{-1}$
       4. Calculate $\vec{x}_{i+1}$ using equation 1.5.
4: end for
5: Output the vector containing the Neutron Density
```

1.2. The Taylor Series Expansion Scheme

The second numerical method, which I will not elaborate on extensively, is based on truncating the Taylor expansion of 1 to the first order. This algorithm is a simple mechanical application of an Explicit Euler method for ODE systems. It is computationally extremely fast, but it lacks the precision of other one-step methods for a given fixed h. This algorithm and its results are taken from the paper [2].

$$\begin{cases} N(t+h) = N(t) + h\dfrac{\rho(t)-\beta}{\Lambda}N(t) + h\displaystyle\sum_{i=1}^{6}\lambda_i C_i(t) \\ C_i(t+h) = C_i(t) + h\dfrac{\beta_i}{\Lambda}N(t) - h\lambda_i C_i(t) \end{cases} \tag{1.7}$$

In my codes, I modified the algorithm in two ways to obtain results, at least in theory, with lower approximation error:

- Taylor expansion to the first and second order.
- Richardson extrapolation to obtain a better initial condition at the next time step.

Algorithm 1.2 Taylor Series Expansion algorithm:

```
1: Start from the initial conditions of $N(0)$, $C_i(0) \forall i = 1, ..., 6$
2: for $t \in [0 : T]$ do
3:     1.  Calculate $N(t+h)$ and $C_i(t+h) \forall i$ using equation 1.7.
       2.  Perform Richardson Extrapolation to fix the initial conditions that are passed
           to the next iteration. (We'll talk about it better in the BEFD scheme section)
4: end for
5: Output the vector containing the Neutron Density
```

1.3. The COnstant REactivity Scheme

The third numerical method analyzed for solving the Point Kinetics Equation is the CORE (COnstant REactivity) algorithm, developed by Quintero-Leyva in [6]. Similar to the PCA method, this model approximates the reactivity function locally as a constant to calculate the next state. However, unlike a simple forward one-step method, it introduces a dependency on the derivative of the number of precursors. To derive the algorithm, it is necessary to apply the Laplace transform to the ODE system 1, and solve for the neutron density $N(t)$, as follows:

$$\begin{cases} s\mathcal{N} = \dfrac{(\rho-\beta)}{\Lambda}\mathcal{N} + \displaystyle\sum_{i=1}^{6} \lambda_i \mathcal{C}_i \\ s\mathcal{C}_i = \dfrac{\beta_i}{\Lambda}\mathcal{N} - \lambda_i \mathcal{C}_i + c_i(0^-) \end{cases} \tag{1.8}$$

$$\begin{cases} s\mathcal{N} = \dfrac{(\rho-\beta)}{\Lambda}\mathcal{N} + \displaystyle\sum_{i=1}^{6} \lambda_i \mathcal{C}_i \\ \mathcal{C}_i = \dfrac{\beta_i \mathcal{N}}{\Lambda(s+\lambda_i)} + \dfrac{c_i(0^-)}{s+\lambda_i} \end{cases} \tag{1.9}$$

From 1 we have: $\lambda_i c_i(0^-) = \frac{\beta_i}{\Lambda}N(0^-) - \frac{dc_i(0^-)}{dt}$. Substitute and solve:

$$\begin{cases} \mathcal{N} = \dfrac{N(0-) + \sum_{i=1}^{6} \frac{1}{s+\lambda_i}\left(\frac{\beta_i}{\Lambda}N(0^-) - \frac{dc_i(0^-)}{dt}\right)}{s\left(\frac{\rho-\beta}{\Lambda}\right) - \sum_{i=1}^{6} \frac{\lambda_i\beta_i}{\Lambda}\frac{1}{s+\lambda_i}} \\ \mathcal{C}_i = \dfrac{\beta_i \mathcal{N}}{\Lambda(s+\lambda_i)} + \dfrac{c_i(0^-)}{s+\lambda_i} \end{cases}$$

Observe that the denominator is the inhour equation, which I know has a number of roots equal to the number of precursors plus one. Therefore, I can decompose $\mathcal{N}$ as follows:

$$\mathcal{N} = \frac{R_1}{s-s_1} + \cdots + \frac{R_7}{s-s_7}$$

where s_k are the roots of the inhour equation. Finally, applying polynomial division and using $\lambda_i c_i(0^-) = \frac{\beta_i}{\Lambda}N(0^-) - \frac{dc_i(0^-)}{dt}$, I derive:

$$R_k = \frac{N(0-) + \sum_{i=1}^{6} \frac{1}{s_k+\lambda_i}\left(\frac{\beta_i}{\Lambda}N(0^-) - \frac{dc_i(0^-)}{dt}\right)}{1 + \sum_{i=1}^{6} \frac{\lambda_i\beta_i}{\Lambda}\frac{1}{(s+\lambda_i)^2}} \tag{1.10}$$

Applying the Heaviside expansion to return to time-variable functions:

$$N(t) = \textstyle\sum_{k=1}^{7} R_k e^{s_k t}$$

Substituting, it is possible to also derive $C_i(0^-))$ and $\frac{dC_i(0^-)}{dt}$, and at this point, it is possible to define an algorithm to calculate $N(t)$.

Adding steps of the algorithm and how to calculate C and $\dot{C}$.

To calculate $C_i(t)$, I start from the second differential equation of the system 1.8 and multiply both sides by $e^{\lambda_i t}$:

$$e^{\lambda_i t}\frac{dC_i(t)}{dt} = \frac{\beta_i}{\Lambda}N(t)e^{\lambda_i t} - \lambda_i C_i(t)e^{\lambda_i t} \tag{1.11}$$

Then, I proceed with the integration over the interval $[0, t]$:

$$\int_0^t e^{\lambda_i \hat{t}} dC_i(\hat{t}) = \int_0^t \left(\frac{\beta_i}{\Lambda}N\left(\hat{t}\right)e^{\lambda_i \hat{t}} - \lambda_i C_i\left(\hat{t}\right)e^{\lambda_i \hat{t}}\right)d\hat{t} \tag{1.12}$$

Integrating by parts the left side and separating the right side:

$$\left[C_i(t)e^{\lambda_i t} - C_i(0^-)\right] - \lambda_i \int_0^t e^{\lambda_i \hat{t}} C_i(\hat{t})d\hat{t} = \frac{\beta_i}{\Lambda}\sum_{k=1}^{7} R_k \int_0^t e^{(s_k+\lambda_i)\hat{t}}d\hat{t} - \lambda_i \int_0^t C_i\left(\hat{t}\right)e^{\lambda_i \hat{t}}d\hat{t} \tag{1.13}$$

Simplifying, rearranging terms, and solving the integral:

$$C_i(t)e^{\lambda_i t} = \frac{\beta_i}{\Lambda}\sum_{k=1}^{7}\frac{R_k}{s_k+\lambda_i}\left[e^{(s_k+\lambda_i)t} - 1\right] + C_i(0^-) \tag{1.14}$$

Finally:

$$C_i(t) = \frac{\beta_i}{\Lambda}\sum_{k=1}^{7}\frac{R_k}{s_k+\lambda_i}\left(e^{s_k t} - e^{-\lambda_i t}\right) + C_i(0^-)e^{-\lambda_i t} \tag{1.15}$$

To conclude, we also check $\dot{C}_i(t)$ for $t \to 0^-$. Therefore, we derive :

$$\dot{C}_i(t) = \frac{\beta_i}{\Lambda} \sum_{k=1}^{7} \frac{R_k}{s_k + \lambda_i} \left(s_k e^{s_k t} + \lambda_i e^{-\lambda_i t} \right) + \lambda_i C_i(0^-) e^{-\lambda_i t} \tag{1.16}$$

$$\dot{C}_i(0^-) = \frac{\beta_i}{\Lambda} \sum_{k=1}^{7} \frac{R_k}{s_k + \lambda_i} (s_k + \lambda_i) + \lambda_i C_i(0^-) \tag{1.17}$$

Noting that $N(0^-) = \sum_{k=1}^{7} R_k$:

$$\dot{C}_i(0^-) = \frac{\beta_i}{\Lambda} N(0^-) + \lambda_i C_i(0^-) \tag{1.18}$$

The CORE scheme thus becomes, in the case of 6 precursors:

Algorithm 1.3 COnstant REactivity algorithm:

```
1: Start from the initial conditions of $N(t_{n-1})$, $C_i(t_{n-1})$, and $\dot{C}_i(t_{n-1}) \forall i = 1, ..., 6$
2: for $iin[1 : N]$ do
3:     1. Find the roots of the inhour equation $s_k^n \forall k = 1, ..., 7$ with $\rho = \rho(t_n)$.
       2. Calculate $R_k^n$ using equation 1.10.
       3. Find $N^n = \sum_{k=1}^{7} R_k^n e^{s_k^n \Delta t}$. The initial condition at the next time step will be:
          $N^{n+1}(0) = N^n$.
       4. Use equation 1.3 to obtain $C_i(t_n) \forall i = 1, ..., 6$.
       5. To find the initial conditions at the next instant for the derivative $\dot{C}_i(t_{n+1}^-)$,
          use 1.18.
4: end for
5: Output the vector containing the Neutron Density
```

1.4. The Backward Euler Finite Difference Scheme

The last method we will analyze is an implicit method, which should be its strength. The method in question is called BEFD (Backward Euler Finite Difference), suggested by the paper [1], and formally it is an implicit Euler method applied to the system of ODEs 1. It appears to be very accurate in its solution, especially in the nonlinear case, as an eighth equation is added to the system 1:

$$\frac{d\rho(t)}{dt} = \frac{d\rho_0(t)}{dt} - BN(t)$$

where $\rho_0(t)$ is my reactivity over time, and B is the absolute value of the reactivity temperature coefficient, thus allowing the simulation of system evolution even at non-constant temperature.

Writing this system in matrix form, it can be represented as:

$$\dot{Y}(t) = A(t, N(t))Y(t)$$

with:

$$A = \begin{bmatrix} (\rho(t,N(t)) - \beta)/\Lambda & \lambda_1 & \lambda_2 & ... & \lambda_6 & 0 \\ \frac{\beta_1}{\Lambda} & -\lambda_1 & 0 & ... & ... & 0 \\ \frac{\beta_2}{\Lambda} & 0 & -\lambda_2 & ... & ... & 0 \\ ... & ... & ... & .... & .... & ... \\ \frac{\beta_6}{\Lambda} & 0 & ... & 0 & -\lambda_6 & 0 \\ -B & 0 & 0 & ... & ... & 0 \end{bmatrix}$$

At this point, we apply backward finite differences to obtain the general form of our method:

$$Y(t_j) = Y(t_{j+1}) - h\left.\frac{dY(t)}{dt}\right|_{t_{j+1}} = Y(t_{j+1}) - h[A(t_{j+1}, N(t_{j+1}))Y(t_{j+1}) + q(t_{j+1})] \quad (1.19)$$

$$[I - hA(t_{j+1}, N(t_{j+1}))]Y(t_{j+1}) = Y(t_j) - hq(t_{j+1}) \quad (1.20)$$

From this, it is possible to obtain the solution of the system at the next time step using a fixed-point method.

Additionally, Richardson extrapolation was applied to the result in this algorithm, in order to obtain a more reliable initial value for the calculation at the next time step. This method involves solving the previous equation at progressively shorter intervals of h, and then reconstructing the optimal solution using the following recursive formula:

$$\varphi_s(h) = \frac{2^s \varphi(h/2) - \varphi_{s-1}(h)}{2^s - 1} \tag{1.21}$$

In the algorithm, $s = 14$ was used, as suggested by the paper [1].

Algorithm 1.4 Backward Euler Foward Difference algorithm:

```
1: Start from the initial conditions of $N(0)$, $C_i(0)$, and $\dot{C}_i(0) \forall i = 1, ..., 6$
2: for $i \in [1 : N]$ do
3:     1. Evaluate $\rho = \rho(t_i)$.
       2. Calculate the matrix $A$.
       3. Use the backward finite difference scheme to compute the new vector $Y(t_{n+1})$
          as in 1.20.
       4. Apply the Richardson Extrapolation in 1.21 to obtain a more reliable initial
          condition for the next iteration.
4: end for
5: Output the vector containing the Neutron Density
```

2 | Simulation and Analysis

2.1. Analysis with Constant and Ramp Reactivity

Before analyzing how the algorithms just discussed perform within the established benchmark, let's check the accuracy of the results for cases that have analytical solutions or are well-known and tabulated in the literature.
We will analyze three cases of step reactivity ρ and one ramp case. The three cases of constant reactivity for positive times represent the stages in which the reactor can be found, while the ramp reactivity is useful for simply representing the reactor startup.

Specifically, the stages obtained from constant reactivity will be:

- *Subcritical Stage*: $\rho = 0.003$
- *Critical Stage*: $\rho = 0.007$
- *Supercritical Stage*: $\rho = 0.008$

It should be noted that these values of ρ are not absolute but depend on the precursors considered for the reaction. In fact, the critical stage is reached when the reactivity is such that: $\rho_{critical} = \beta = \sum_{i=1}^{6} \beta_i = 0.007$ in the specific case I considered.

Focusing now on the step reactivity case, we observe that for constant ρ, the system transitions from a nonlinear system and a numerically *stiff* problem to a simple system of linear Ordinary Differential Equations with constant coefficients. This can be solved analytically through the following steps:

1. Impose $y(t) = [N(t), C_1(t), ..., C_6(t)]$
2. Rewrite the system as $\dot{y}(t) = Ay(t)$
3. Find the eigenvalues λ and the matrix of eigenvectors X of the matrix A
4. The number or density of neutrons will be $N(t) = \sum_{i=1}^{7} c_i e^{\lambda_i t} X_{1i}$
5. The constants $c_i \forall i = 1, ..., 7$ can be obtained by solving $[c_1, ..., c_7]' = X^{-1} y_0$

2.2. Analysis with Sinusoidal Reactivity

Let's focus on a particular case of reactivity, the sinusoidal one: $\rho = \rho_0 \sin\left(\frac{\pi t}{T}\right)$ with $\rho_0 = 0.005333$ and $T = 50s$. Additionally: $\Lambda = 10^{-7}$, $\beta = 0.0079$, and $\lambda = 0.077$. We are dealing with a "fast" reactor, referring to the neutron generation time, Λ, which is of the order of 10^{-8}. Moreover, the considered reactor has only one precursor, unlike the six considered so far to evaluate the algorithms. The neutron density profile in this case can be observed in Figure **??**.

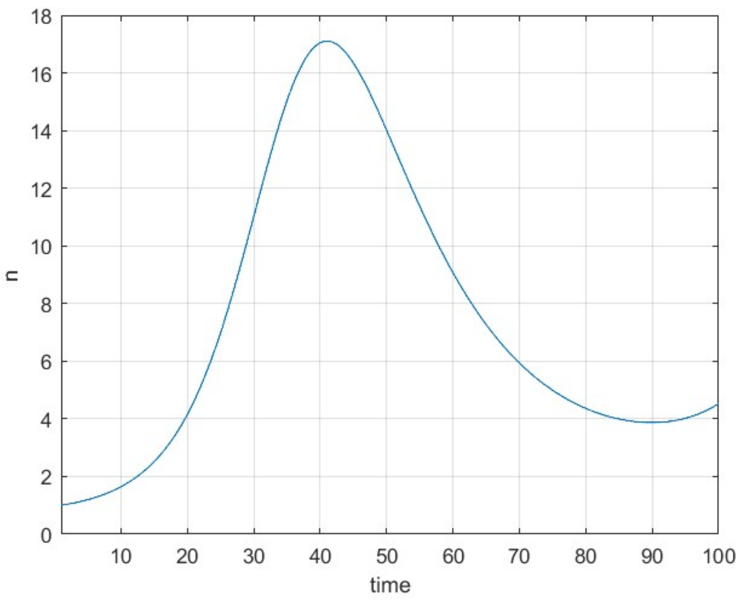

Figure 2.1: Profile with $\rho = \rho_0 \sin\left(\frac{\pi t}{T}\right)$, sinusoidal.

All Matlab codes were run on an HP X360 laptop with Windows 11 Home operating system, an Intel Core i5-1240P, and 16 GB of RAM. Relative errors are calculated with respect to the values given in the paper [1], and for the analysis of the TSE scheme behavior, the PCA method was used, which seems to obtain the smallest relative error among the remaining methods. For the relative error calculation, only the error at the last instant of the simulation is considered.

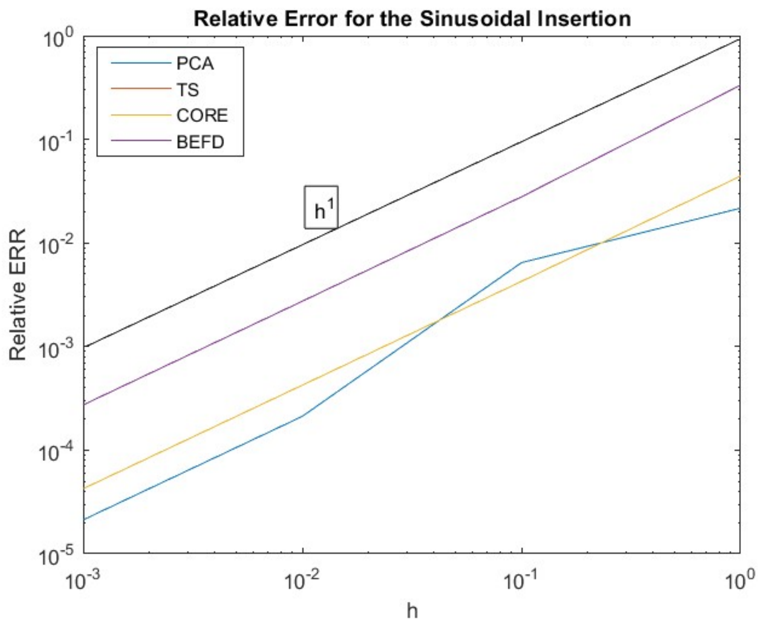

Figure 2.2: Relative error of the schemes as a function of the step h.

From Figure 2.2, we can infer, also with the help of the h^1 line, that the error decrease is linear for all analyzed methods. However, it is not possible to compute the methods for smaller steps due to computational times, which also increase linearly with the step.

2.2.1. A digression on the TSE scheme in the sinusoidal case

As can be seen from Figure 2.2, the TSE method is not present because it does not converge for the parameters assigned by the problem. In this section, I want to investigate which parameters do not allow the scheme to converge and, if possible, find which subset of sinusoidal problems can be solved with the TSE method.

At each iteration, the scheme performs the following update:

$$\begin{cases} N(t+h) = N(t) + h\dfrac{\rho(t)-\beta}{\Lambda}N(t) + h\lambda_1 C_1(t) \\ C_1(t+h) = C_1(t) + h\dfrac{\beta_1}{\Lambda}N(t) - h\lambda_1 C_1(t) \\ \rho(t+h) = \rho_0 \sin\left(\dfrac{\pi(t+h)}{T}\right) \end{cases} \tag{2.1}$$

And we notice that Λ, i.e., the neutron generation speed inside the reactor, is 10^{-8}, so unless h steps are of comparable order, it could be the parameter that makes the Taylor Series Expansion method diverge in this case.

The simulations were carried out by calculating the relative error with respect to the PCA method, which in Figure 2.2 we observed to converge for the reactor with high generation speed. Due to computational power limitations, I limited myself to steps of $h = 10^{-4}$ and limited the computational time to $t = 10s$. Figures 2.3, 2.4, and 2.5 show the convergence graphs for $\Lambda = 10^{-3}, 10^{-4}, 10^{-5}$:

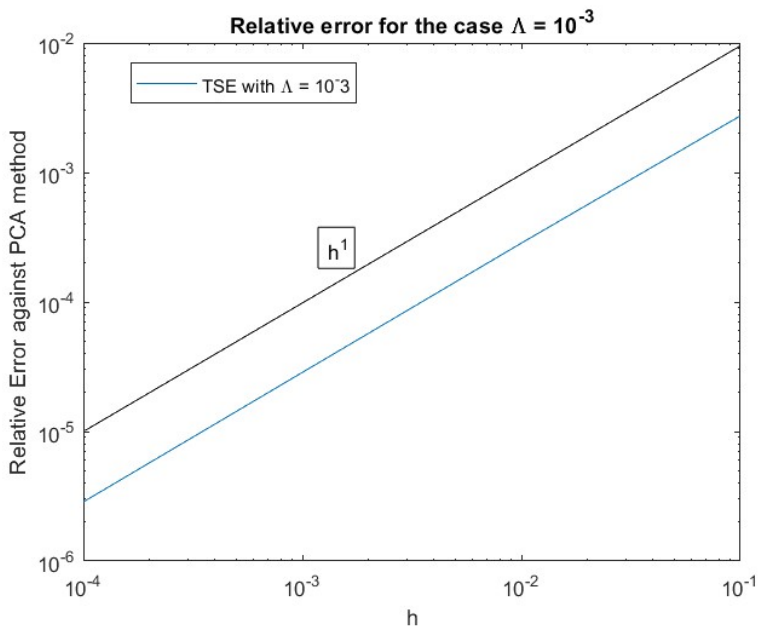

Figure 2.3: Decrease in relative error for $\Lambda = 10^{-3}$

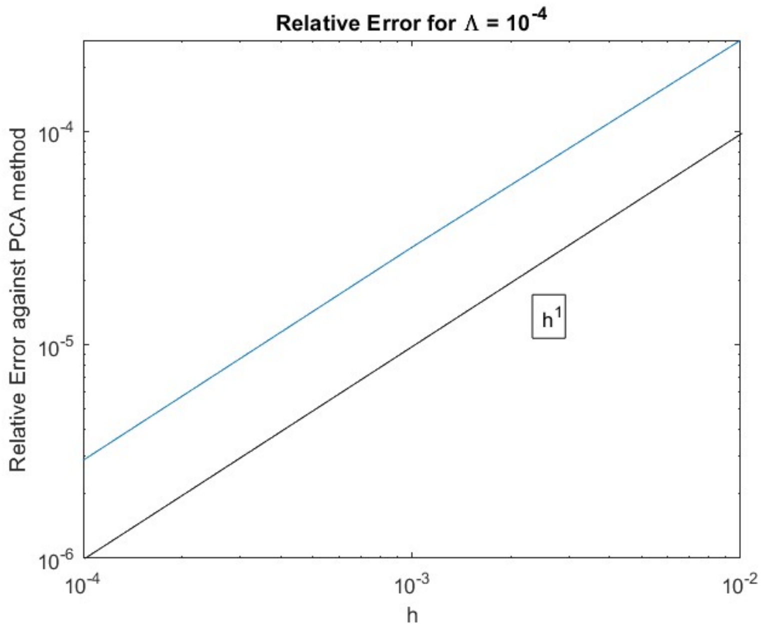

Figure 2.4: Decrease in relative error for $\Lambda = 10^{-4}$

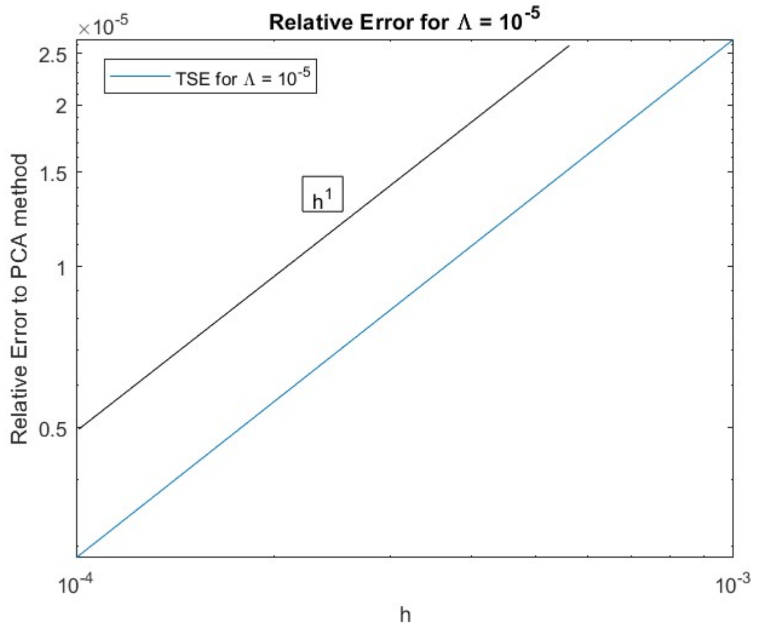

Figure 2.5: Decrease in relative error for $\Lambda = 10^{-5}$

I also eliminated all steps h from which the scheme diverged. This leads us to observe that, for example, for $\Lambda = 10^{-5}$, we cannot use steps $h \geq 10^{-3}$, for $\Lambda = 10^{-4}$, we cannot use steps $h \geq 10^{-2}$. This condition seems to hold true for all experiments.

I therefore suppose that there is a relationship between the "step" parameter and the "reactor speed" parameter and that for $\frac{h}{\Lambda} \leq 100$, we have linear convergence of the TSE scheme.

2.3. Benchmark

As the final benchmark, we will consider a more realistic case of a nuclear reactor, in which reactivity is not independent but is a function of the number of neutrons in the system, a system known as feedback. The choice of using this benchmark stems from the absence, in the literature, of a study of the solutions of the Point Kinetics Equations in such a case, except for the BEFD algorithm developed in [1], which we will use to adapt the other algorithms discussed previously.

For the algorithm, we refer to the paper [5], where the adiabatic feedback benchmark is illustrated. First, we will use the following λ and β:

$$\lambda = [1.26 \cdot 10^{-3}, 3.17 \cdot 10^{-2}, 1.15 \cdot 10^{-1}, 3.11 \cdot 10^{-1}, 1.4, 3.87]$$

$$\beta = [266 \cdot 10^{-6}, 1491 \cdot 10^{-6}, 1316 \cdot 10^{-3}, 2849 \cdot 10^{-3}, 896 \cdot 10^{-3}, 182 \cdot 10^{-3}]$$

$$\Lambda = 5 \cdot 10^{-5}$$

Additionally, the adiabatic feedback will be represented by the differential equation:

$$\frac{d\rho(t,N)}{dt} = \frac{d\rho_0(t)}{dt} - \alpha N(t) \tag{2.2}$$

Where $\rho_0(t) = 0.1t$ is a ramp and $\alpha = 10^{-11}$. In Figure 2.6, we can see the profile.

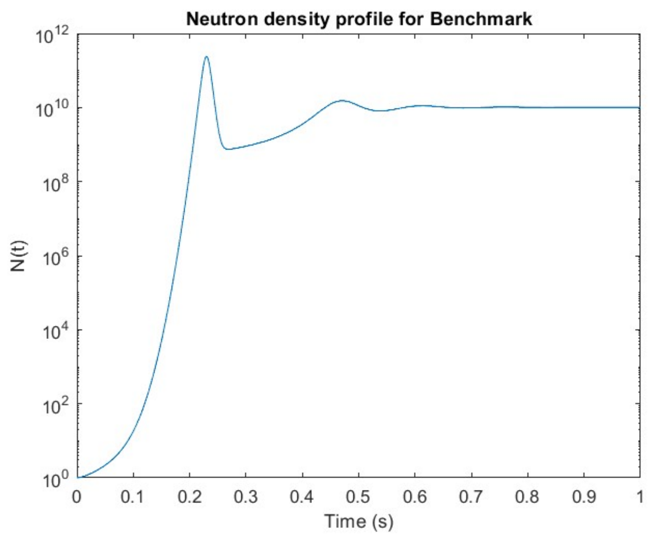

Figure 2.6: Neutron profile in the Benchmark case.

Referring to the results cited in the paper by [5], we implement adiabatic feedback in the previously discussed schemes. Then, in Figure 2.6, we can see the neutron density profile $N(t)$. In Figure 2.7, we analyze the mean square error (MSE) as a function of the step used, using values reported in the paper [5] as reference, which, however, refer only to the interval $T \in [0, 0.250]$ seconds. Subsequently, in Figure 2.8, there is a graph, on a logarithmic scale, of the time (in seconds) required to complete the simulation.

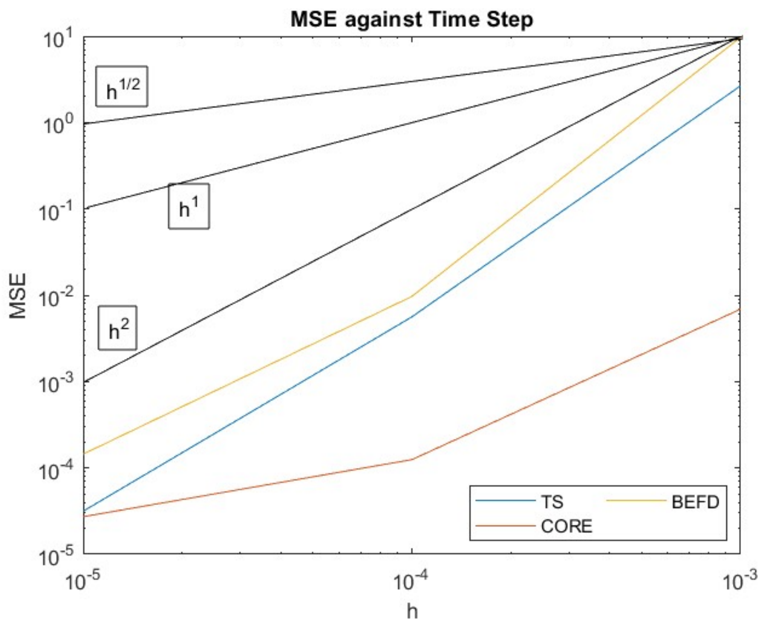

Figure 2.7: MSE of the schemes as a function of the step.

In Figure 2.7, as in 2.8, the PCA algorithm is not present. The reason is the impossibility of reaching a value in the rootfinder function halfway through the computation for values other than $h = 0.005$. However, we observe that the TSE method suggests a quadratic relationship with the mean square error, while both the CORE and BEFD algorithms decrease with a k such that $h^1 < h^k < h^2$. However, to confirm these assumptions, it would be necessary to simulate the benchmark for steps smaller than $h < 10^{-5}$; however, as shown in Figure **??**, this proves to be complex:

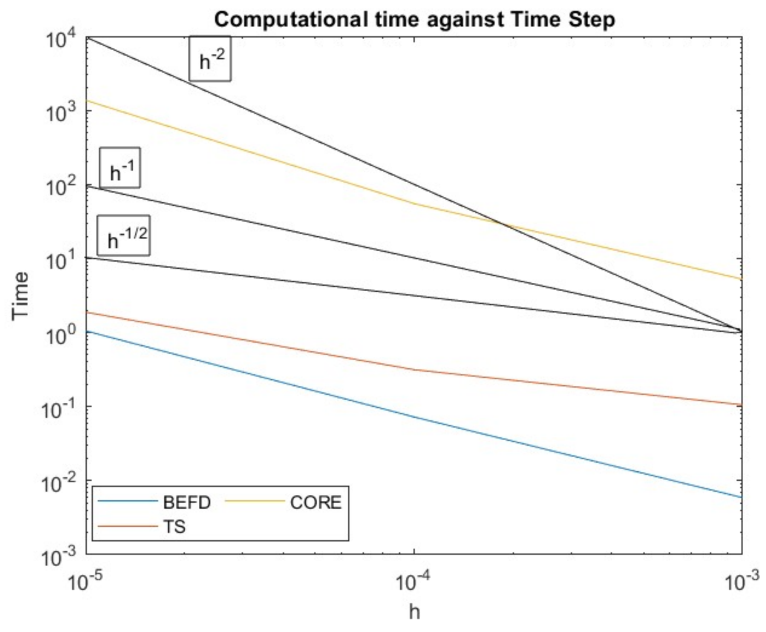

Figure 2.8: Computational time of the schemes as a function of the step.

Indeed, if the Taylor Series Expansion (TSE) based algorithm has a linear growth of computational time, the BEFD scheme has an almost quadratic growth of computational time with respect to the chosen step.

3 | Conclusions

This paper aimed to implement algorithms in MATLAB for solving the Point Kinetics Equations, which, until now, except for rare exceptions, have not been appended to papers where their theoretical treatment is carried out. Additionally, these algorithms have been tested both under simple conditions with tabulated results and under more realistic conditions, such as a reactor with adiabatic feedback.

From the simulations conducted, it appears that some algorithms are more versatile than others, such as the CORE and BEFD schemes, which have proven to perform well under all reactivity conditions. From the results obtained for the TSE algorithm in the sinusoidal case, we have suggested a possible relationship between the parameter Λ and the step h that ensures the convergence of the scheme.

4 | Appendix - MATLAB Code

```
%function elapsedTime = CTest()
    clc;clear all; close all;

    H = [1e-2, 1e-1, 1]; %1e-4,

    nmodels = 4;
    nh = length(H);

    tic
        for j=1:nh

            h=H(j);
            n=1;
            [T_pca,N_pca] = initPCA(n,h);
            [T_ts, N_ts] = initTS(n,h);
            [T_core, N_core]=initCORE(n,h);
            [T_befd, N_befd] = initBEFD(n,h);
            out_values = Exact_const;

            err_rel(:,j) = abs(([N_pca(end), N_ts(end),
               N_core(end), N_befd(end)] - out_values(end))/
               out_values(end));
        end

        %Grafico in scala logaritmica dell'errore relativo
           per ogni metodo
        figure()
        xlabel('h')
        ylabel('Relative Error')
        for k=1:nmodels
```

```
            loglog(H, err_rel(k,:))
            hold on
        end
        legend({'PCA', 'TS', 'CORE', 'BEFD'}, 'Location','northwest','NumColumns',2) %
        hold off
    toc
    elapsedTime=toc;
%end
```

```
function sinTest()
    clc;clear all; close all;

    n = 5;
    tic
    [T_pca,N_pca,color_pca] = initPCA(n);
    toc
    %[T_ts, N_ts, color_ts, N1_ts, color1_ts] = initTS(n);
    tic
    [T_core,N_core,color_core] = initCORE(n);
    toc
    tic
    [T_befd, N_befd, color_befd] = initBEFD(n);
    toc


    xlabel('Time')
    ylabel('Neutron density')
    plot(T_pca,N_pca, Color =color_pca)
    hold on
    %plot(T_ts,N_ts, Color = color_ts)
    %plot(T_ts,N1_ts, Color = color1_ts)
    plot(T_core,N_core, Color=color_core)
    plot(T_befd,N_befd, Color = color_befd)
    legend({'PCA','CORE', 'BEFD'}, 'Location','northwest','NumColumns',2)
end
```

```
function elapsedTime = BenchTest()
    clc;clear all; close all;

    rval = [1.00095540, 1.06882056, 1.81008431e1, 5.01107924
       e3, 1.84748246e8, 3.17939153e9];
    H = [1e-5,1e-4,1e-3]; %1e-6,
    T = [0.001, 0.010, 0.100, 0.150, 0.200, 0.250];

    nmodels = 3;
    nh = length(H);

    ERR = zeros(nmodels, nh);
    Times = zeros(nmodels, nh);

    for i=0:0
        for j=1:nh
            h=H(j);
            n=i;
            val = round(T/h);
            %{
            tic
            [T_pca,N_pca] = initPCA(n,h);
            toc
            PCA_t = toc;
            PCA_val = sum(abs((rval-N_pca(val))./rval));
            %}
            tic
            [T_ts, N_ts] = initTS(n,h);
            toc
            TS_t = toc;
            TS_val = sum(((rval-N_ts(val))./rval).^2/length(
               rval));

            tic
            [T_core, N_core]=initCORE(n,h);
            toc
```

```matlab
            CORE_t = toc;
            CORE_val = sum(((rval-N_core(val)')./rval).^2/length(rval));

            tic
            [T_befd, N_befd] = initBEFD(n,h);
            toc
            BEFD_t = toc;
            BEFD_val = sum(((rval-N_befd(val))./rval).^2/length(rval));

            ERR(:,j) = [TS_val; CORE_val; BEFD_val];
            Times(:,j) = [TS_t; CORE_t; BEFD_t];
            [i,j]
        end

        %Grafico in scala logaritmica dell'errore relativo per ogni metodo
        figure()
        for k=1:nmodels
            loglog(H, ERR(k,:))
            hold on
        end
        xlabel('h')
        ylabel('MSE')
        legend({'TS', 'CORE', 'BEFD'}, 'Location','northwest','NumColumns',2) %
        hold off

        %Grafico in scala logaritmica del tempo computazionale per ogni metodo
        figure()
        for k=1:nmodels
            loglog(H, Times(k,:))
            hold on
        end
        xlabel('h')
        ylabel('Time')
        legend({'PCA', 'TS', 'CORE', 'BEFD'}, 'Location','northwest','NumColumns',2) %
        hold off
    end
end
```

4.1. PCA

The entire code for the PCA method was written by Kinard and Allen in [4]; I will only add the code that I personally modified.

```
function [t,n,color]=initPCA(n,h)

addpath('C:\Users\alewi\Desktop\Tesi\MATLAB\BDEF - Ganapol')

[lambda, beta, L, rho, alpha, q,tmax] = cases(n);
beta_sum = sum(beta);

f_case = 1;
init_cond = [1; beta./(L*lambda)];
rval = zeros(length(beta)+1,1);

y = piecewise_const(lambda, beta, beta_sum, L, tmax, h, rho,
    f_case, init_cond,rval, alpha);
t = y(1,:);
n = y(2,:);
c = y(3:end,:);

color = 'b';

end
```

```
function y = piecewise_const(lambda, beta, beta_sum, L,
    target, h, rho, f_case, init_cond,rval, alpha)
% This function will calculate the solution to the point
    kinetics equation
% starting at time = 0. The following is a summary of the
    arguments for the
% function:
% lambda = a vector of the decay constants for the delayed
    neutrons
% beta = a vector containing the delayed-neutron fraction
% beta_sum = the sum of all of the betas
% L = neutron generation time
% target = the final, target time of the function
% h = step size
% f_case = similar to "rho_case" but it determines the source term
% to be used

% Determine the number of delay groups, thereby the size of our solution

m = length(lambda) + 1;
% Calculate the values of several constants that will be needed in
% the control of the iterations as well as set up some basic matrices.
x = init_cond; time = 0;
f_hat = zeros(m,1);
d_hat= zeros(m,m);
big_d = zeros(m,m);
i = 1;
iterations = ceil(target / h);
result = zeros(m+1,iterations);
% Begin time dependent iterations
d=rval;
while time < (target)
    time = time + h;
    % Calculate the values of the reactivity and source at the midpoint
    mid_time = time + (h)/2;

    rho_alpha = @(t) rho(t) + alpha*h*sum(result(2,:));

    p = rho_alpha(mid_time);
    source = f(f_case, mid_time);

    % Caculate the roots to the inhour equation
    d = inhour(lambda, L, beta, p, d);

    % Calculate the eigenvectors and the inverse of the
      matrix
    % of eigenvectors
    Y = ev2(lambda, L, beta, d);
    Y_inv = ev_inv(lambda, L, beta, d);

    % Construct matrices for computation
    for k = 1:m
        d_hat(k,k) = exp(d(k)*h);
        big_d(k,k) = d(k);
    end

    f_hat(1) = source;
    big_d_inv = zeros(m,m);
    for k =1:m
        big_d_inv(k,k) = 1/big_d(k,k);
    end

    % Compute next time step
    x = (Y * d_hat * Y_inv)*(x + (Y*big_d_inv*Y_inv*f_hat)) -
        (Y*big_d_inv*Y_inv*f_hat);

    % Store results in a matrix
    for j = 1:m
        result(1,i) = time;
        result(j+1,i) = x(j);
    end
    if floor(i/iterations*100)~=floor((i-1)/iterations*100)
        floor(i/iterations*100)
    end
    % Update counters
    i = i + 1;
end

y=[[0; init_cond],result];
```

4.2. Taylor Series Expansion

```
function [t, N, color, N_1, color1]=initTS(n,h)
    %Questo script permette di simulare l'equazione PKE
       modificandone i
    %parametri iniziali, attraverso una espansione in serie
       di Taylor del
    %sistema di EDO, come descritto da: "A Taylor series
       solution of the
    %reactor point kinetics equations" di McMahon e Pierson.

    addpath('C:/Users/alewi/Desktop/Tesi/MATLAB/PCA - Kinard'
       )
    addpath('C:/Users/alewi/Desktop/Tesi/MATLAB/BDEF -
       Ganapol')

    t0 = 0;

    %Parametri propri del reattore considerato
    [lambda, beta, LAMBDA, rho_0, alpha, q, t_max] = cases(n)
       ;

    sum(beta)
    %Valori iniziali per risolvere il problema di Cauchy
       associato al sistema
    %di EDO.
    N0 = 1;
    C0 = beta./(LAMBDA*lambda);
    y0 = [N0; C0];
    tic
    [t, N, C, N_1, C_1] = taylorPkeSolver(LAMBDA, lambda,
       beta, y0, rho_0, t0,t_max, h, alpha);
    toc
    color = 'r';
    color1 = 'g';

    %plot(t,N_1,'r')
```

```
end
```

```
function [t,N, C,N_1, C_1] = taylorPkeSolver(LAMBDA, lambda, beta, y0, rho_0, t0,t_max, h, alpha)
    n = ceil((t_max-t0)/h);

    BETA = sum(beta);

    N = [y0(1), zeros(1, n-1)];
    C = [y0(2:end), zeros(length(y0)-1, n-1)];

    N_1 = [y0(1), zeros(1, n-1)];
    C_1 = [y0(2:end), zeros(length(y0)-1, n-1)];

    u_0 = y0;

    t(1) = t0;
    %figure()

    %plot((1:n)*h, rho((1:n)*h))
    for i=2:n+1
        t(i) = (i-1)*h;

        rho = @(t) rho_0(t) + alpha*h*sum(N);
        rho1 = @(t) rho_0(t) + alpha*h*sum(N_1);

        D = diag([-lambda]);
        v = [lambda'];
        w = [beta/LAMBDA];
        A = @(h) [(rho(t(i)+h)-BETA)/LAMBDA, v; w, D];

        u_0 = [N(i-1);C(:,i-1)];

        u_01 = [N_1(i-1);C_1(:,i-1)];

        %myRichardson(A, zeros(length(y0),1),[N_1(i-1);C_1(:,i-1)], h, 5, 1e-9);
```

```matlab

        N(i) = (1+h*(rho(i*h)-BETA)/LAMBDA)*u_0(1) + h*sum(lambda.*u_0(2:end));
        C(:,i) = (h/LAMBDA*beta)*u_0(1) + (1-h*lambda).*u_0(2:end);

        %{
        N_1(i) = (1+h*(rho1(i*h)-BETA)/LAMBDA +h^2/2*(((rho1(i*h)-BETA)/LAMBDA)^2 + ...
                    sum(lambda.*beta/LAMBDA)))*u_01(1) + h*sum(lambda.*u_01(2:end)) + ...
                    h^2/2*((rho1(i*h)-BETA)/LAMBDA)*sum(lambda.*u_01(2:end)) - h^2/2*sum(lambda.^2.*u_01(2:end));

        C_1(:,i) = (h/LAMBDA*beta+h^2/2*((beta*(rho1(i*h)-BETA)/LAMBDA^2)-lambda.*beta/LAMBDA))*u_01(1)+...
                 (1-h*lambda-(h*lambda).^2/2).*u_01(2:end) + h^2/2*beta/LAMBDA*sum(lambda.*u_01(2:end));
        %}
        if floor(i/n*100)~=floor((i-1)/n*100)
            floor(i/n*100)
        end
    end

end
```

```matlab
function y = myRichardson(A, q, u_0, h, n, toll)
    R(:, 1, 1) = (1+h*A(h))*u_0;
    for i=1:100
       h = h/2;

       R(:, i + 1, 1) = (1+h*A(h))*u_0;

       for j=1:i
          R(:,i + 1, j + 1) = (2^j*R(:,i + 1, j) - R(:,i, j))
```

```
                 /(2^j - 1);
        end

        if ( norm( R(:,i + 1, i + 1) - R(:,i, i) ) < toll )
            break;
        elseif ( i == 100 )
            error( 'Richardson extrapolation failed to converge
                ' );
        end

    end
    y = R(:,i+1, i+1);
```

4.3. CORE

```
function [t, N, color]=initCORE(n,h)
%In questo codice MATLAB verra costruito l'algoritmo CORE
   come descritto
%all'interno del paper: "CORE: A numerical algorithm to solve
    the point
%kinetics equation" di Quintero Leyva.
%questo file contiene i dati iniziali del problema e richiama
    il solutore
%per l'equazione inhour e il file cases con i vari test
addpath('C:\Users\alewi\Desktop\Tesi\MATLAB\PCA - Kinard')
addpath('C:\Users\alewi\Desktop\Tesi\MATLAB\BDEF - Ganapol')

%Dati del problema e dati reattore
t_0 = 0;
[lambda, beta, LAMBDA, rho, alpha, q,t_max] = cases(n);

%Condizioni iniziali del problema (situazione critica), come
   elencate nel
%paper
N_0 = 1;
C_0 = (beta./(lambda*LAMBDA)*N_0)';
C_dot_0 = zeros(1, 6);

y0 = [N_0, C_0, C_dot_0];

[t,N, C, C_dot,S_k] = CORE_solver(y0, rho, t_0, t_max, h,
   beta,lambda,LAMBDA,alpha);

color = [0.9290 0.6940 0.1250];

end
```

```
function [t, N, C, C_dot, S_k] = CORE_solver(y0, rho_0, t0,
   t_max, h, beta,lambda,LAMBDA,alpha)
    n = ceil((t_max-t0)/h);
    addpath('C:\Users\alewi\Desktop\Tesi\MATLAB\PCA - Kinard'
        )
    N = zeros(n, 1);
    C = zeros(n, 6);
    C_dot = zeros(n,6);

    N(1) = y0(1);
    C(1, :) = y0(2:7);
    C_dot(1, :) = y0(8:end);

    t = 0:h:t_max;

    for i = 2:(n+1)

        rho = @(t) rho_0(t) + alpha*h*sum(N);

        C_dot(i,:) = (beta./LAMBDA)'*N(i-1) - lambda'.*C(i
            -1,:);

        %S_k = inhour(lambda, LAMBDA, beta, rho(i*h), zeros(
            length(beta)+1,1));

        D = diag([-lambda]);
        v = [lambda'];
        w = [beta/LAMBDA];
        A = [(rho(t(i))-sum(beta))/LAMBDA, v; w, D];

        S_k = eig(A);

        R_k = R_calculation(N(i-1), C_dot(i,:), S_k, beta,
            lambda, LAMBDA);
        R_k = R_k';

        N(i) = sum(R_k.*exp(S_k*h));

        C(i,:) = C_calculation(R_k, S_k, C(i-1,:), h, beta,
            lambda, LAMBDA);

        if floor(i/(n+1)*100)~=floor((i-1)/(n+1)*100)
            floor(i/(n+1)*100)
        end

    end

    N = N';
end
```

```
function R_k = R_calculation(N, C_dot, S_k, beta, lambda,
   LAMBDA)
    R_k = zeros(1, length(S_k));
    for i = 1:length(S_k)
        s1 = 0;
        for j=1:length(beta)
            add = beta(j)/(S_k(i) + lambda(j));
            s1 = s1 + add;
        end
        s1 = s1/LAMBDA;
        s1 = s1+1;


        s2 = 0;
        for j=1:length(beta)
            add = lambda(j)*beta(j)/(S_k(i) + lambda(j))^2;
            s2 = s2 + add;
        end
        s2 = s2/LAMBDA;
        s2 = s2+1;

        s3 = 0;
        for j=1:length(beta)
            s3 = s3 + C_dot(j)/(S_k(i)+lambda(j));
        end

        R_k(i) = (N*s1 - s3)/s2;
    end
```

```
function C = C_calculation(R_k, S_k, C_prec, h, beta, lambda,
    LAMBDA)
    C = zeros(1, 6);
    for i=1:length(beta)
        res = 0;
        for j=1:length(S_k)
            res = res + R_k(j)*(exp(S_k(j)*h) - exp(-lambda(i
               )*h))/(S_k(j)+lambda(i));
        end
        res = res*beta(i)/LAMBDA;
        res = res + C_prec(i)*exp(-lambda(i)*h);

        C(i) = res;
    end
end
```

4.4. BEFD

```
function [T, N, color] = initBEFD(n,h)

    format long;
    [lambda, beta, L, rho_0, alpha, q,tmax] = cases(n);

    u_0 = [1; beta./(lambda*L);0];

    tmin = 0;

    n=ceil((tmax-tmin)/h);

    toll = 1e-10;
    T(1) = 0;
    u(:,1) = u_0;
    R = [];

    for i = 2:(n+1)

        T(i) = (i-1)*h;

        rho = @(n, t) rho_0(t)+alpha*(sum(u(1,:)*h)+n*h);

        D = diag([-lambda; 0]);
        v = [lambda', 0];
        w = [beta/L; alpha];
        A = @(n,t) [(rho(n,t)-sum(beta))/L, v; w, D];


        F = @(y) -(eye(length(u_0)) - h*A(y(1),(i+1)*h))*y +
           u(:,i-1) + h*q;
        options = optimoptions('fsolve','TolFun', toll,'
           Display','off');

        %Richardson Extrapolation
        u_0 = myRichardson_Benchmark_fun(A, T(i), q, u(:,i-1)
            , h, 10, toll);
        u(:,i) = fsolve(F, u_0, options);

        R(i) = rho(u(1,i),i*h);

        round(i/n*100,2)

    end

    N = u(1,:);
    color = 'm';
end
```

```
function y = myRichardson_Benchmark_fun(A, t, q, u_0, h, n,
   toll)

    F = @(y) -(eye(length(u_0)) - h*A(y(1),t+h))*y + u_0 + h*
       q;
    options = optimoptions('fsolve','TolFun', toll, 'Display'
       ,'off');

    R(:, 1, 1) = fsolve(F, u_0, options);

    for i=1:100
       h = h/2;

       F = @(y) -(eye(length(u_0)) - h*A(y(1),t+h))*y + u_0 +
           h*q;
       options = optimoptions('fsolve','TolFun', toll, '
          Display','off');

       R(:, i + 1, 1) = fsolve(F, u_0, options);

       for j=1:i
          R(:,i + 1, j + 1) = (2^j*R(:,i + 1, j) - R(:,i, j))
             /(2^j - 1);
```

```
        end

        if ( norm( R(:,i + 1, i + 1) - R(:,i, i) ) < toll )
           break;
        elseif ( i == 100 )
           error( 'Richardson extrapolation failed to converge
              ' );
        end

    end
    y = R(:,i+1, i+1);
```

```

function [lambda, beta, L, rho_0, alpha, q,tmax] = cases(n)

    if(n==0)
        %Benchmark
        lambda = [1.26e-3; 3.17e-2; 1.15e-1; 3.11e-1; 1.4;
           3.87];
        beta = [266e-6; 1491e-6; 1316e-6; 2849e-6; 896e-6;
           182e-6];
        L = 5e-5;

        rho_init = 0.1;
        rho_0 = @(t) rho_init*t;

        alpha = -1e-11;
        q = zeros(8,1);
        tmax = 1;

    elseif(n==1)
        %Step Insertion of rho = 0.003
        beta = [0.000266, 0.001491, 0.001316, 0.002849,
           0.000896, 0.000182]';
        lambda = [0.0127, 0.0317, 0.115, 0.311, 1.4, 3.87]';
        L = 2e-5;

```

```matlab
        rho_init = 0.003;
        rho_0 = @(t) rho_init;

        alpha = 0;
        q = zeros(8,1);
        tmax=20;

    elseif(n==2)
        %Step Insertion of rho = 0.007
        beta = [0.000266, 0.001491, 0.001316, 0.002849,
           0.000896, 0.000182]';
        lambda = [0.0127, 0.0317, 0.115, 0.311, 1.4, 3.87]';
        L = 2e-5;

        rho_init = 0.007;
        rho_0 = @(t) rho_init;

        alpha = 0;
        q = zeros(8,1);
        tmax=2;

    elseif(n==3)
        %Step Insertion of rho = 0.008
        beta = [0.000266, 0.001491, 0.001316, 0.002849,
           0.000896, 0.000182]';
        lambda = [0.0127, 0.0317, 0.115, 0.311, 1.4, 3.87]';
        L = 2e-5;

        rho_init = 0.008;
        rho_0 = @(t) rho_init;

        alpha = 0;
        q = zeros(8,1);
        tmax=1;


    elseif(n==4)
```

```
        %Ramp Insertion of 0.1$
        beta = [0.000266, 0.001491, 0.001316, 0.002849,
           0.000896, 0.000182]';
        lambda = [0.0127, 0.0317, 0.115, 0.311, 1.4, 3.87]';
        L = 2e-5;

        rho_init = 0.007*0.1;
        rho_0 = @(t) rho_init*t;

        alpha = 0;
        q = zeros(8,1);
        tmax=9;

    elseif(n==5)
        %Sinusoidal Insertion T=50, rho_0 = 0.0053333
        beta = 0.0079;
        lambda = 0.077;
        L = 1e-8;

        rho_init = 0.0053333;
        T=50;
        rho_0 = @(t) rho_init*sin(pi*t/T);

        alpha = 0;
        q = zeros(3,1);
        tmax=100;


    elseif(n==6)
        %Step Insertion of rho = 0.007 with feedback
        lambda = [1.24e-2; 3.05e-2; 1.11e-1; 3.01e-1; 1.13;
           3];
        beta = [2.1e-4; 1.41e-3; 1.27e-3; 2.55e-3; 7.4e-4;
           2.7e-4];
        L = 5e-5;

        rho_init = sum(beta)*1.5;
        rho_0 = @(t) rho_init;

        alpha = -2.5e-6;
        q = zeros(8,1);
        tmax=100;


    elseif(n==7)
        %Ramp Insertion of 0.1$ with feedback
        lambda = [1.24e-3; 3.05e-2; 1.11e-1; 3.01e-1; 1.13;
           3];
        beta = [2.1e-4; 1.41e-3; 1.27e-3; 2.55e-3; 7.4e-4;
           2.7e-4];
        L = 5e-5;

        ramp_rate = 0.1;
        rho_0 = @(t) ramp_rate*t;

        alpha = -1e-11;
        q = zeros(8,1);
        tmax=10;

     elseif(n==8)
        %Benchmark Sinusoidale
        beta = 0.0079;
        lambda = 0.077;
        L = 1e-5;

        rho_init = 0.0053333;
        T=50;
        rho_0 = @(t) rho_init*sin(pi*t/T);

        alpha = 0;
        q = zeros(3,1);
        tmax=10;

     end
```

```
end
```

Bibliography

[1] G. B.D. A highly accurate algorithm for the solution of the point kinetics equations. *Annals of Nuclear Energy*, 62, 2012.

[2] A. P. D. McMahon. A taylor series solution of the reactor point kinetics equations.

[3] D. Hetrick. Dynamics of nuclear reactors. 1971.

[4] E. A. Matthew Kinard. Efficient numerical solution of the point kinetics equations in nuclear reactor dynamics. *Annals of Nuclear Energy*, 31:1039–1051, 2004.

[5] R. F. P. Picca, B.D. Ganapol. An accurate technique for the solution of the non-linear point kinetics equations. *International Conference on Mathematics and Computational Methods Applied to Nuclear Science and Engineering*, 2011.

[6] B. Quintero-Leyva. Core: A numerical algorithm to solve the point kinetics equations. *Annals of Nuclear Energy*, 35, 2008.

[7] A. A. Y.A. Chao. A resolution to the stiffness problem of reactor kinetics. *Nuclear Science and Engineering*, 90:40–46, 1985.

List of Figures

List of Tables