Point Kinetics Equation

Modelli matematici e Applicazioni

Alessandro Wiget

Laurea Triennale in Ingegneria Matematica

Indice

Sommario

L'obiettivo di questa tesi, per il corso di laurea triennale in ingegneria matematica, è di analizzare lo stato dell'arte dei metodi numerici studiati fino a ora per la risoluzione delle equazioni che descrivono la cinetica puntiforme dei neutroni all'interno di un reattore a fissione nucleare. Oltre ad analizzare la precisione dei diversi metodi già esistenti sarà proposto un nuovo metodo e valutato secondo appropriati benchmark.

Elenco delle tabelle

Elenco delle figure

Introduzione

In questa tesi verranno analizzati algoritmi matematici utilizzati in fisica nucleare atti ad approssimare il sistema di EDO chiamato Point Kinetics Equations, che descrive la densità di neutroni N(t) liberi nel reattore in un determinato istante di tempo e la concentrazione dei precursori all'interno dello stesso reattore, $C_i(t)$, dove con precursori intendiamo tutti gli elemente attraverso cui passa la nostra reazione temonucleare, poichè a qualsiasi stadio verranno prodotti altri neutroni per via della reazione di fissione nucleare, considereremo 6 precursori, come standard nella letteratura:

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

Come è possibile notare osservando le equazioni la linearità o non-linearità del sistema dipende dalla funzione $\rho(t)$, chiamata reattività, ed espressa in \$, un numero puro dato dalla normalizzazione del numero di neutroni liberi in un istante di tempo con il numero di neutroni ritardati in grado di iniziare una nuova reazione termonucleare $\beta_{eff} = \beta I$, con I detto fattore di importanza, cioè la probabilità che un elettrone prodotto da una successiva reazione nucleare ne generi una nuova a sua volta.
I termini λ_i e β_i sono propri del reattore e del comportamento del combustibile considerato, infatti β_iè la frazione di neutroni prodotti dal precursore i-esimo. Il termine Λ invece rappresenta il tempo di generazione dei neutroni all'interno del reattore.

Come standard nella letteratura si utilizzano una moltitudine di combinazioni di Λ, λ_i e β_i, noi seguiremo il paper [1], che propone i seguenti valori:

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

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

$$\Lambda = 0.00002$$

Tutti gli algoritmi sono stati scritti da zero su MATLAB, a meno che non presenti già nei paper come codice e sono reperibili alla fine della tesi.

Derivazione della PKE

Risoluzione esatta della PKE

Il Modello PCA

Il primo modello cha andremo ad analizzare è stato presentato da M. Kinard e E.J. Allen in [2] ed è chiamato PCA o piecewise constant approximation.
Per risolvere il sistema di equazioni differenziali con questo metodo, una volta definito il passo h desiderato si procede con l'approssimazione delle funzioni $\rho(t)$ e $\vec{F}(t)$ a funzioni costanti in ciascuno degli intervalli considerati, scegliendo come valore il valore del punto medio dell'intervallo:

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

$$\vec{F}(t) \approx \vec{F}\left(\frac{t_i + t_{i+1}}{2}\right) = \vec{F}_i \tag{3}$$

Una volta completato lo stadio di approssimazione il problema, su ogni sottointervallo i, risulta della forma:

$$\begin{cases} \frac{d\vec{x}(t)}{dt} = (A + Bi)\vec{x}(t) + \vec{F}_i \\ \vec{x}(t_i) = \vec{x_i} \end{cases} \tag{4}$$

Ovvero un problema di Cauchy che può essere risolto in modo esatto per ogni sottointervallo. per fare ciò è necessario tuttavia trovare gli autovalori e autovettori della matrice $A + B_i$. Nella letteratura tuttavia gli autovalori sono le radici dell'*inhour equation*, che possono essere trovati tramite iterazioni del metodo di Newton:

$$\rho_i = \beta + \Lambda\omega - \sum_{j=1}^{m} \frac{\beta_j \lambda_j}{\omega + \lambda_j} \tag{5}$$

Per reattività ρ costanti possiamo quindi trovare gli autovalori:

$\rho = 0.003$	$\rho = 0.007$	$\rho = 0.008$
-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

Tabella 1: Soluzioni dell'*Inhour Equation* per vari ρ

E, utilizzando il metodo presente in [3], è possibile risalire agli autovettori della matrice per ottenere le matrici del cambio di base per la diagonalizzazione P_i e P_i^{-1}. A questo punto il metodo PCA ci permette di ottenere un metodo a passo singolo esplicito, facilmente risolvibile da un qualsiasi calcolatore.

$$\vec{x}_{i+1} = P_i e^{D_i h} P_i^{-1}[\vec{x}_i + P_i D^{-1} P_i^{-1} \vec{F}_i] - P_i D^{-1} P_i^{-1} \vec{F}_i \tag{6}$$

Per questo metodo abbiamo analizzato il suo comportamento per varie casistiche di reattività $\rho(t)$ presenti in letteratura, è necessario far notare che il codice presente nel paper di Kinard e Allen non era sufficiente per eseguire tutte le simulazioni. Abbiamo quindi riadattato, quando necessario il il codice MATLAB. É importante notare tuttavia che, seppur utilizzando il medesimo codice contenuto all'interno del paper di Kinard e Allen, i risultati non combaciano perfettamente con quelli riportati, le differenze sono riportate nelle seguenti tabelle:

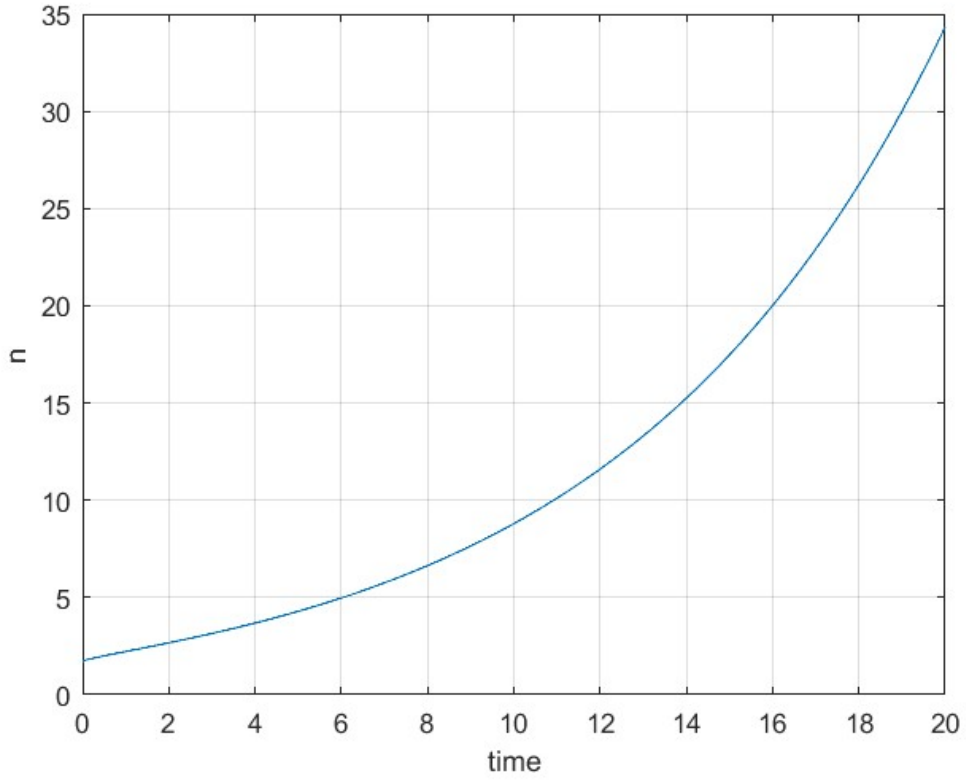

Figura 1: PCA con $\rho = 0.003$, stadio subcritico.

Time (s)	Kinard e Allen	Codice MATLAB
t = 1	2.2098	2.2248
t = 10	8.0192	8.79935
t = 20	2.8297×10^1	3.433×10^1

Tabella 2: PCA per $\rho = 0.003$ e $h = 0.01$

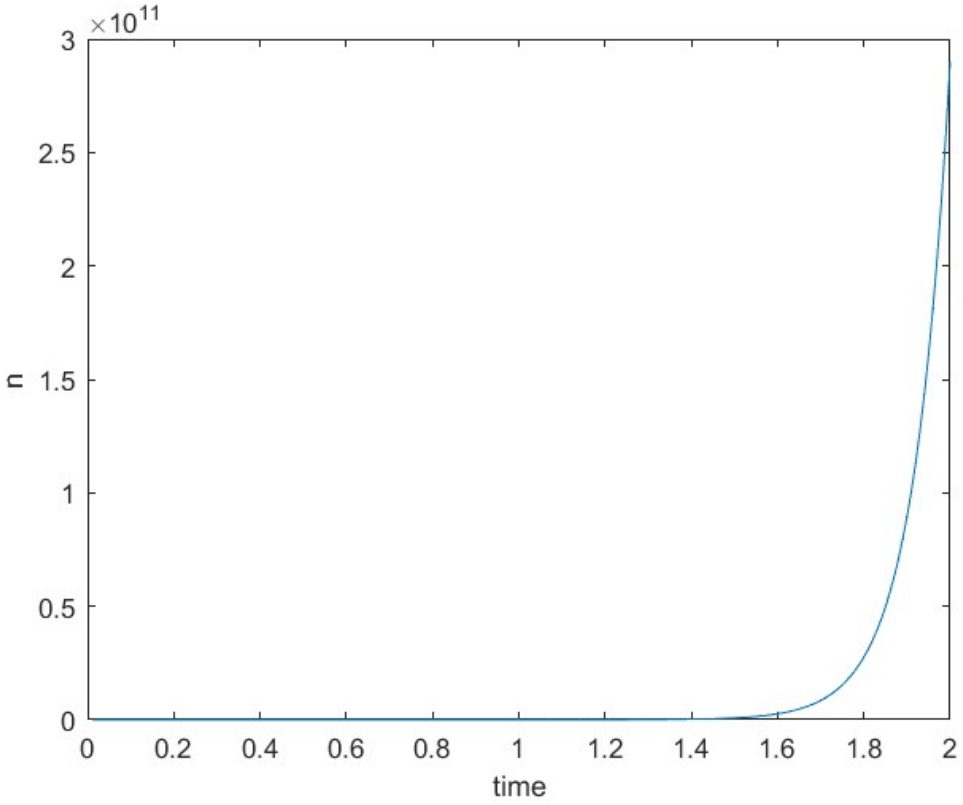

Figura 2: PCA con $\rho = 0.007$, stadio critico.

Time (s)	Kinard e Allen	Codice MATLAB
t = 0.01	4.5088	4.5090
t = 0.5	5.3459 $\times 10^3$	5.60221 $\times 10^3$
t = 2	2.0591 $\times 10^{11}$	2.5669 $\times 10^{11}$

Tabella 3: PCA per $\rho = 0.007$ e $h = 0.01$

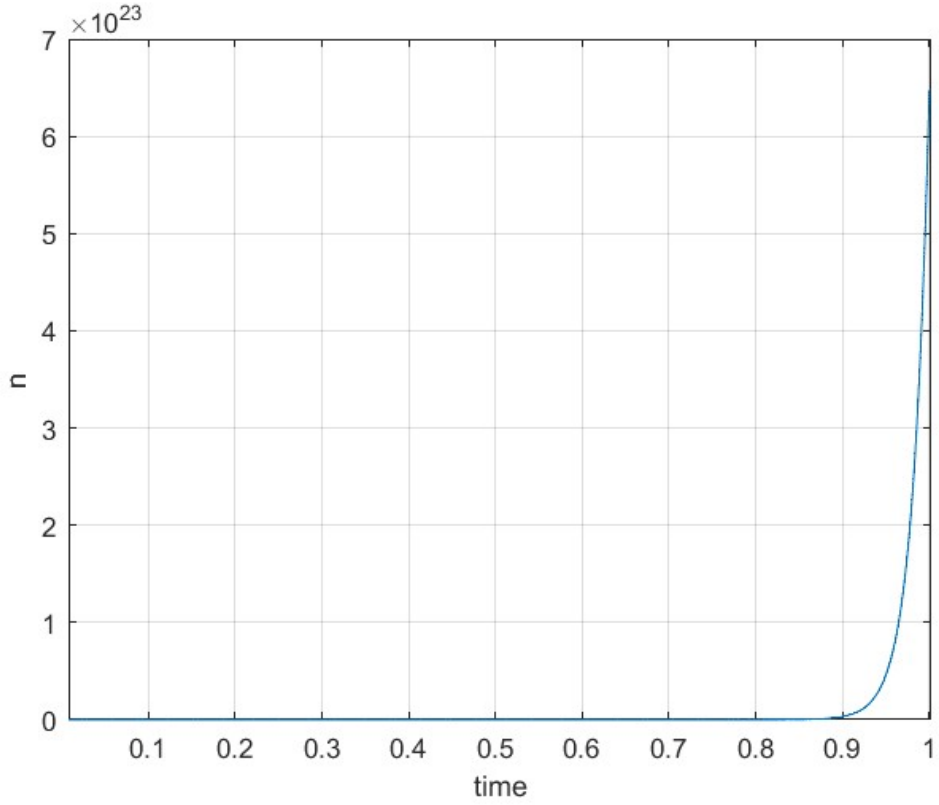

Figura 3: PCA con $\rho = 0.008$, stadio supercritico.

Time (s)	Kinard e Allen	Codice MATLAB
t = 0.01	6.2046	6.20308
t = 0.1	1.4104 $\times 10^3$	1.4147 $\times 10^3$
t = 1	6.1634 $\times 10^{23}$	6.44975 $\times 10^{23}$

Tabella 4: PCA per $\rho = 0.008$ e $h = 0.001$

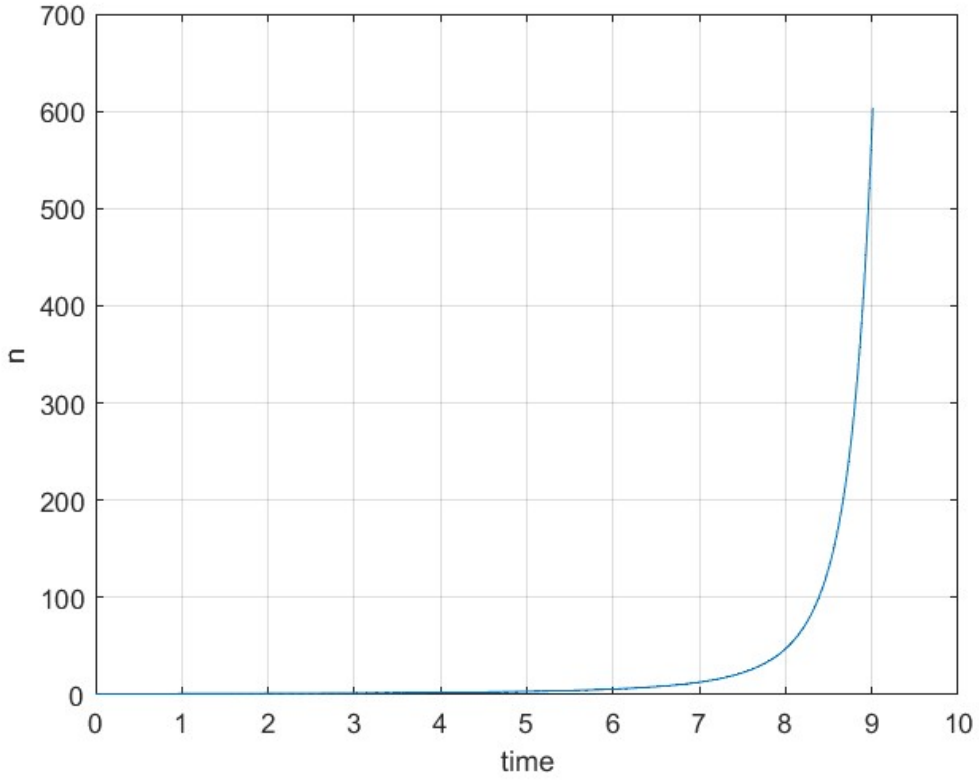

Figura 4: PCA con $\rho = 0.0007t$, rampa.

Time (s)	Kinard e Allen	Codice MATLAB
t = 2	1.3382	1.34332
t = 4	2.2278	2.26041
t = 6	5.5802	5.80714
t = 8	4.2772×10^1	4.7399×10^1
t = 9	4.8735×10^2	5.81301×10^2

Tabella 5: PCA per $\rho = 0.007t$ e $h = 0.01$

Nel paper [2] viene dimostrata la convergenza di ordine 2 del metodo appena presentato, che implica la zero stabilità del metodo, per il teorema di Lax Richtmeyer. Per concludere l'analisi di questo metodo studierò brevemente l'assoluta stabilità del metodo.
Consideriamo quindi il problema modello nel caso di un sistema di equazioni differenziali ordinarie:

$$\begin{cases} \dot{\vec{y}}(t) = A\vec{y}(t) \\ \vec{y}(t_0) = \vec{y}_0 \end{cases} \tag{7}$$

Dal momento che il metodo PCA 6, ponendo $\vec{F} = \vec{0}$, diventa: $\vec{x}_{i+1} = e^{Ah}\vec{x}_i$, ovvero un metodo Eulero in avanti per sistemi di EDO, la cui assoluta stabilità è garantita solo nel caso di autovalori della matrice A tutti negativi, dunque il metodo PCA è un metodo limitatamente assolutamente stabile.

Il Modello Taylor

Il secondo metodo numerico, su cui non mi dilungherò particolarmente, si basa su un troncamento dell'espansione di Taylor della 1 al primo ordine. Questo algoritmo è una semplice applicazione meccanica di un metodo Eulero Esplicito per sistemi di EDO, infatti computazionalmente è estremamente veloce, pecca tuttavia, della precisione degli altri metodo ad un passo per un particolare h fissato. Questo algoritmo, ed i suoi risultati sono presi dal paper [4].

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

Nei miei codici ho modificato in due modi l'algoritmo per ottenere risultati, almeno in teoria, come meno errore di approssimazione:

- Espansione di Taylor al primo e secondo ordine.
- Estrapolazione di Richardson per ottenere una condizione iniziale all'istante successivo migliore.

Come possiamo osservare dalle figure sotto, l'unica simulazione nel quale le modifiche danno un contributo significativo è nel caso supercritico, (in rosso il codice standard, in verde la versione modificata):

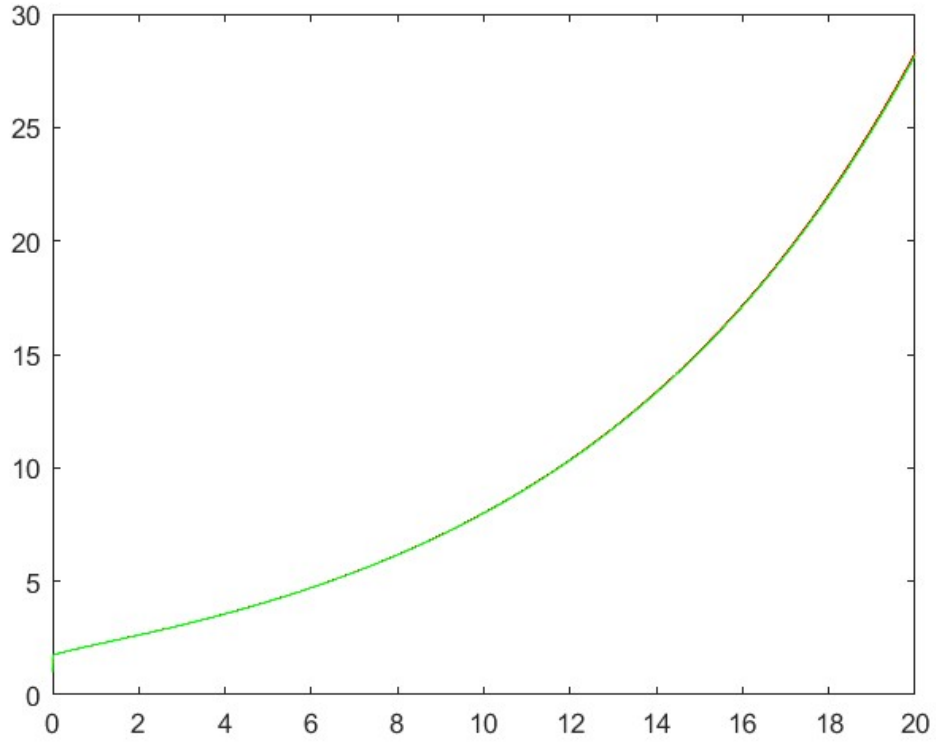

Figura 5: TS con $\rho = 0.003$, stadio subcritico.

Time (s)	MacMahon	MATLAB	MATLAB + modifiche
t = 1	2.2099	2.20943	2.20943
t = 10	8.0192	8.01754	8.01754
t = 20	2.8297×10^1	2.82895×10^1	2.8163×10^1

Tabella 6: TS per $\rho = 0.003$ e $h = 0.001$

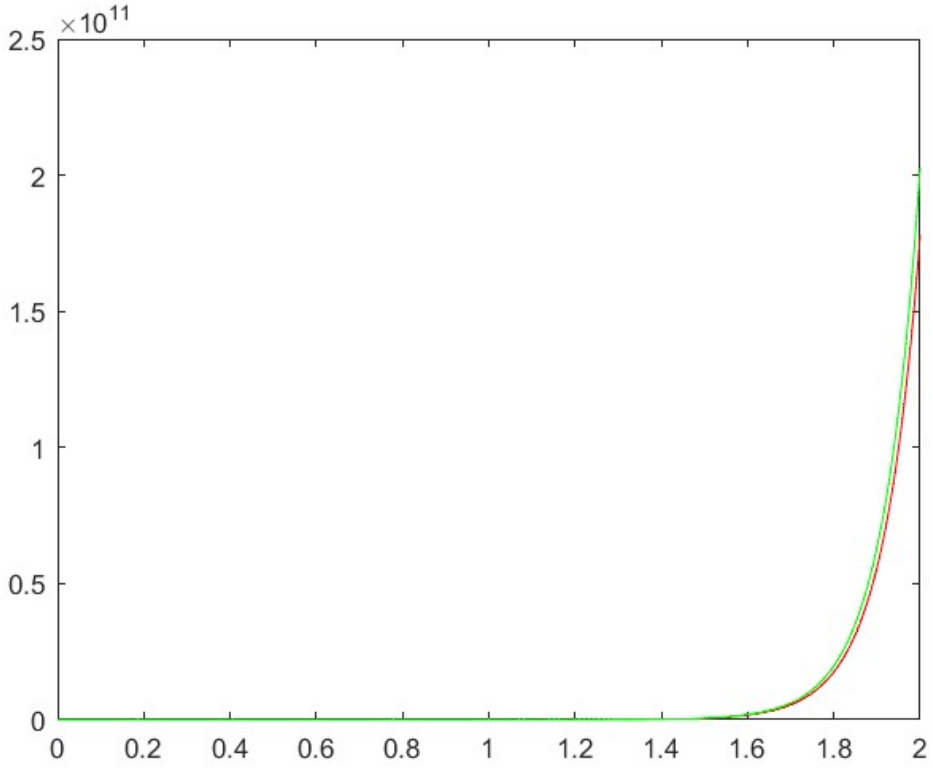

Figura 6: TS con $\rho = 0.007$, stadio critico.

Time (s)	MacMahon	MATLAB	MATLAB + modifiche
t = 0.01	4.5086	4.15637	4.15637
t = 0.5	5.3447×10^3	5.1095×10^3	5.2802×10^3
t = 2	2.0566×10^{11}	1.77922×10^{11}	2.0284×10^{11}

Tabella 7: TS per $\rho = 0.007$ e $h = 0.001$

Per il caso supercritico ci rifacciamo quindi ai risultati di Kinard e Allen:

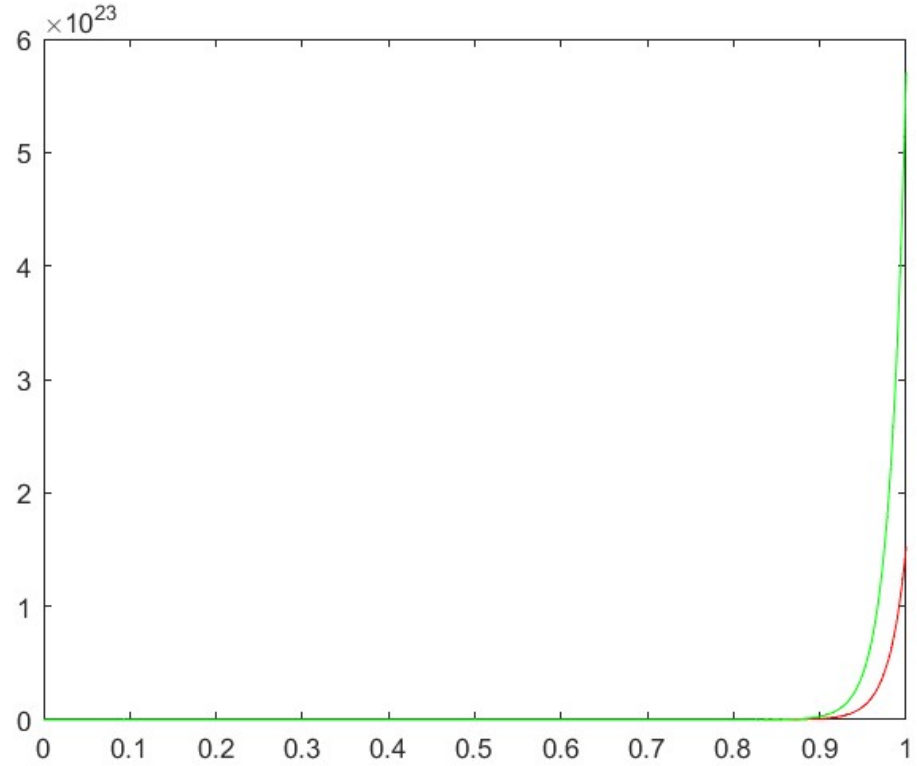

Figura 7: TS con $\rho = 0.008$, stadio supercritico.

Time (s)	Kinard e Allen	MATLAB	MATLAB + modifiche
t = 0.01	6.0229	5.41656	5.55332
t = 0.5	1.4104 $\times 10^3$	1.17007 $\times 10^3$	1.33447 $\times 10^3$
t = 1	6.1634 $\times 10^{23}$	1.52237 $\times 10^{23}$	5.70911 $\times 10^{23}$

Tabella 8: TS per $\rho = 0.008$ e $h = 0.001$

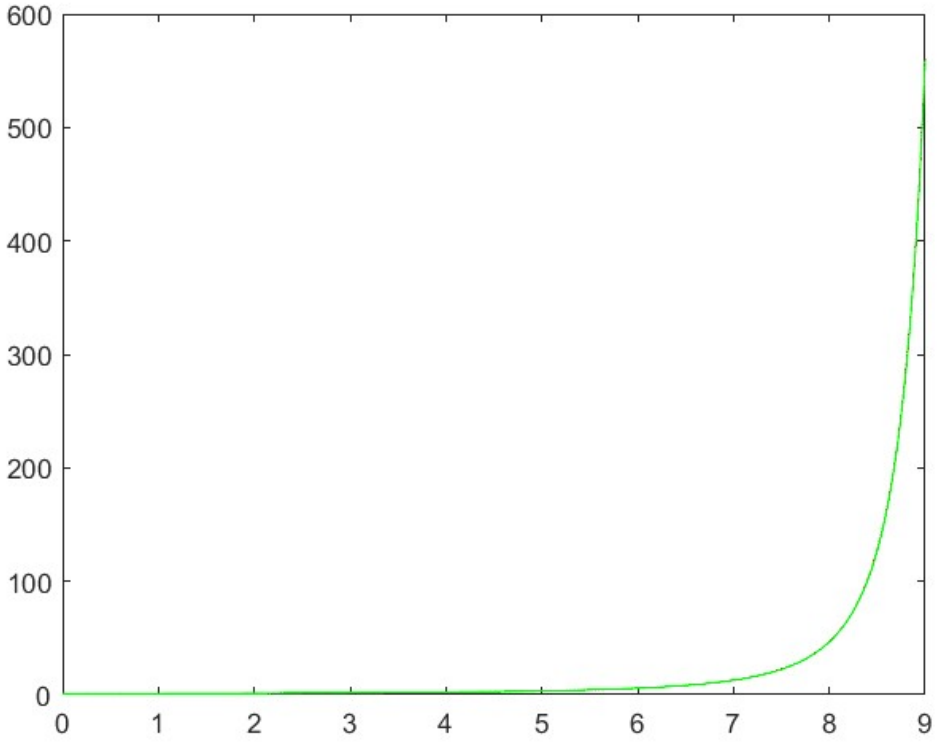

Figura 8: TS con $\rho = 0.0007t$, rampa.

Time (s)	PCA	MATLAB	MATLAB + modifiche
t = 2	1.3382	1.34128	1.34048
t = 4	2.2285	2.254	2.25176
t = 6	5.5823	5.77368	5.76462
t = 8	4.2789×10^1	4.66422×10^1	4.65306×10^1
t = 9	4.8752×10^2	5.603×10^2	5.59268×10^2

Tabella 9: TS per $\rho = 0.007t$ e $h = 0.001$

Il Modello CORE

Il terzo metodo numerico analizzato per risolvere la Point Kinetics Equation sarà l'algoritmo CORE (COnstant REactivity), sviluppato da Quintero-Leyva in [5]. Si tratta di un modello che, in modo simile al metodo PCA approssima localmente la funzione di reattività ad una costante per calcolare lo stato successivo; tuttavia, al contrario di un semplice metodo ad un passo in avanti, si introdurrà una dipendenza dalla derivata del numero di precursori. Per derivare l'algoritmo è necessario applicare la trasformata di Laplace al sistema di EDO 1, e risolvere per il numero di neutroni N(t), nel seguente modo:

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

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

Dalla 1 ricavo: $\lambda_i c_i(0^-) = \frac{\beta_i}{\Lambda}N(0^-) - \frac{dc_i(0^-)}{dt}$. Sostituisco e risolvo:

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

Osservo che il denominatore è l'inhour equation, che so avere numero di radici pari al numero di precursori più uno. Posso quindi scomporre $\mathcal{N}$ nel seguante modo:

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

Dove s_k sono le radici dell'inhour equation.
Infine, applicando la divisione polinomiale e utilizzando $\lambda_i c_i(0^-) = \frac{\beta_i}{\Lambda}N(0^-) - \frac{dc_i(0^-)}{dt}$ ricavo:

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

Applicando l'espansione di Heaviside per ritornare alle funzioni a variabile temporale:

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

Sostituendo è possiblile ricavare anche $c_i(0^-)$ e $\frac{dc_i(0^-)}{dt}$, e a questo punto è possiblile definire un algoritmo per calcolare N(t).

Aggiungere passaggi dell'algoritmo e come calcolare C e $\dot{C}$

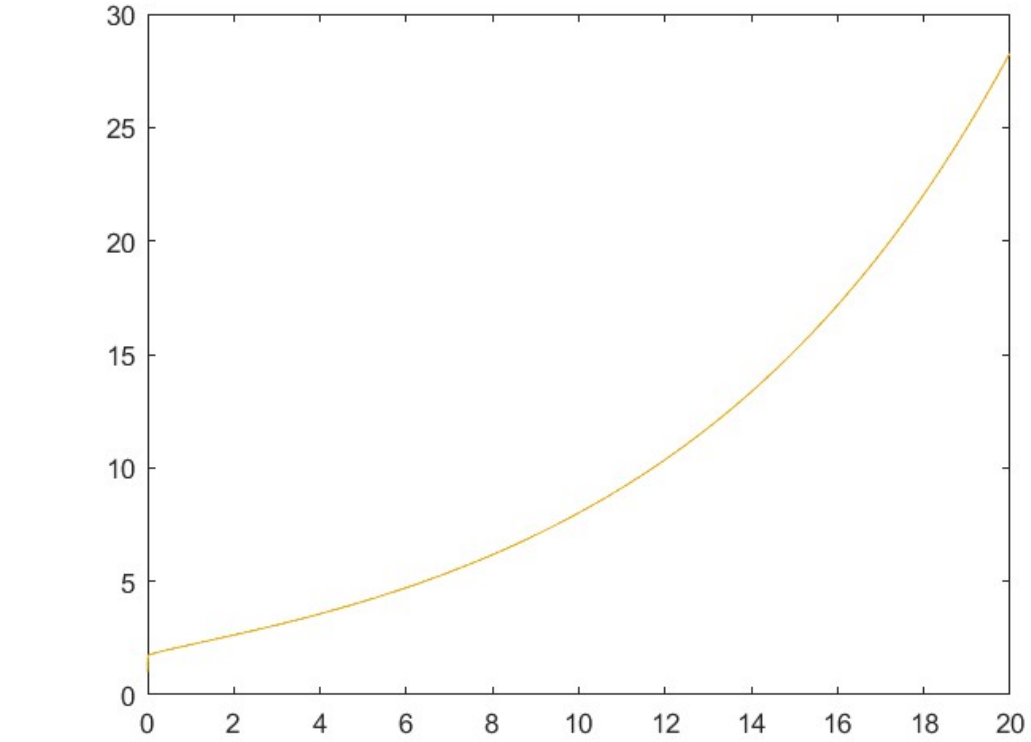

Figura 9: CORE con $\rho = 0.003$, stadio subcritico.

Time (s)	Kinard e Allen	MacMahon	CORE
t = 1	2.2098	2.2099	2.20984
t = 10	8.0192	8.0192	8.0192
t = 20	2.8297×10^1	2.8297×10^1	2.8297×10^1

Tabella 10: CORE per $\rho = 0.003$ e $h = 0.01$

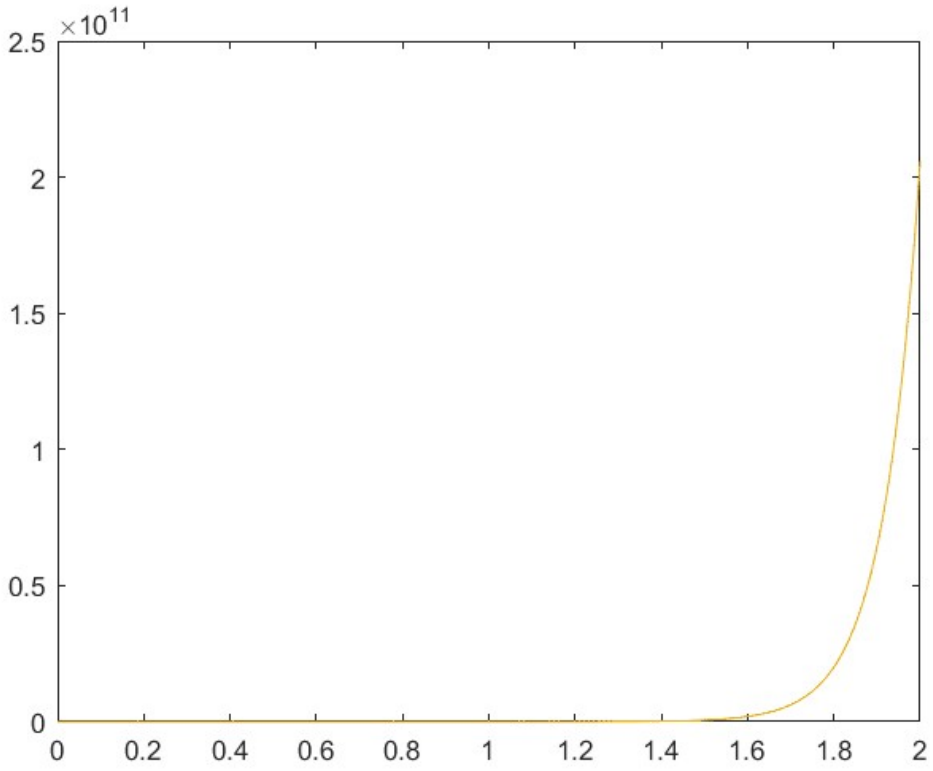

Figura 10: CORE con $\rho = 0.007$, stadio critico.

Time (s)	Kinard e Allen	MacMahon	CORE
t = 0.01	2.2098	2.2099	4.50886
t = 0.5	8.0192	8.0192	5.34589×10^3
t = 2	2.8297×10^1	2.8297×10^1	2.05916×10^{11}

Tabella 11: CORE per $\rho = 0.007$ e $h = 0.01$

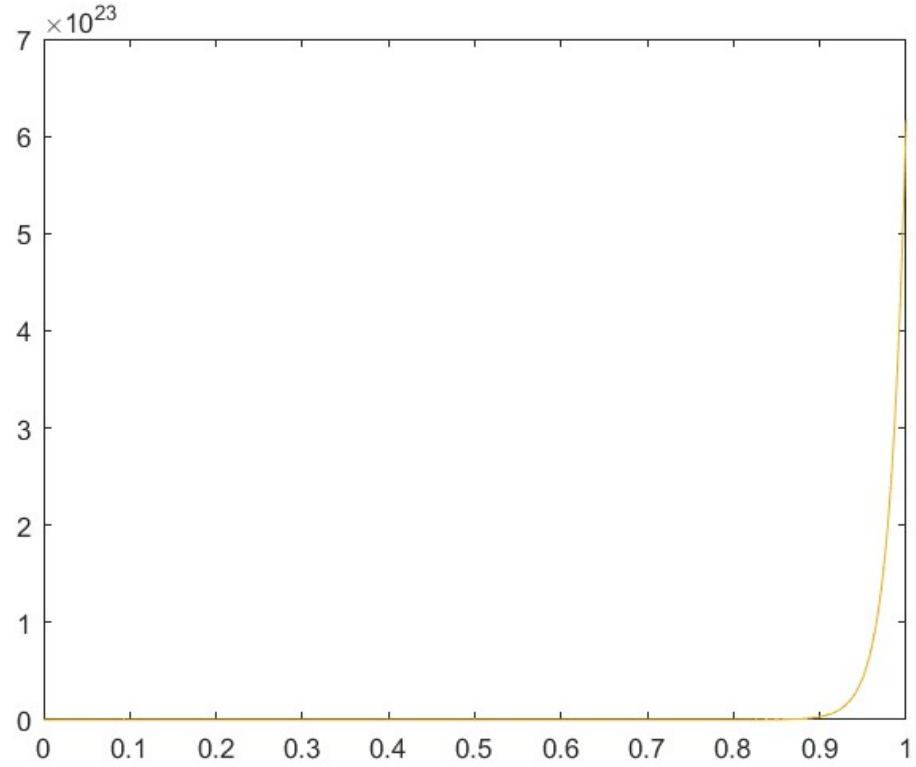

Figura 11: CORE con $\rho = 0.007$, stadio supercritico.

Time (s)	Kinard e Allen	MacMahon	CORE
t = 0.01	2.2098	2.2099	6.20285
t = 0.1	8.0192	8.0192	1.41042×10^3
t = 1	2.8297×10^1	2.8297×10^1	6.1633×10^{23}

Tabella 12: CORE per $\rho = 0.008$ e $h = 0.01$

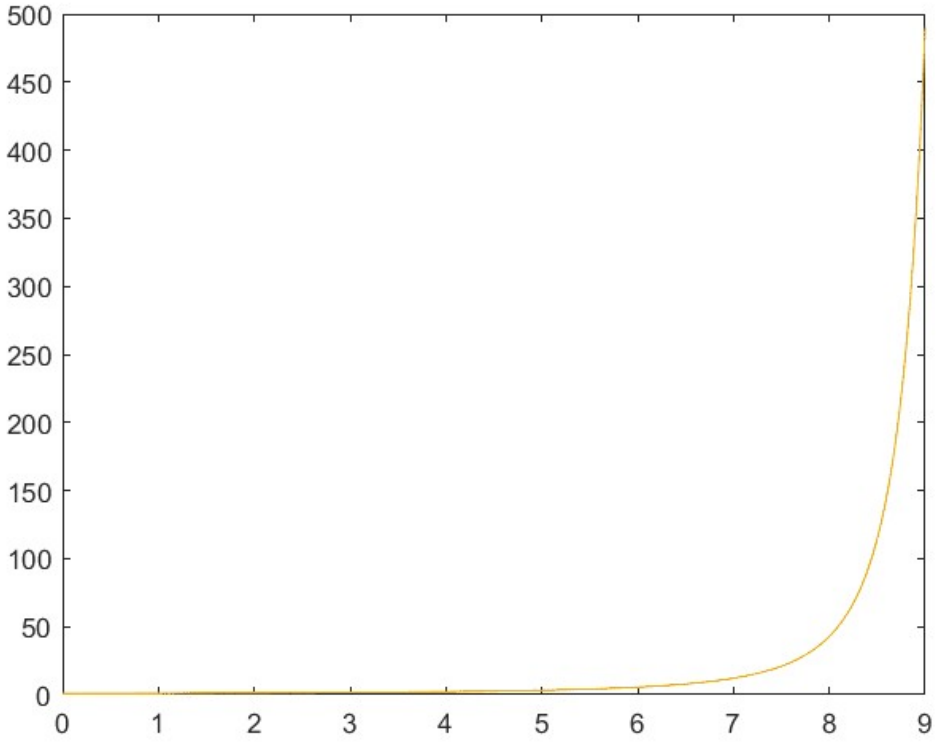

Figura 12: CORE con $\rho = 0.0007t$, rampa.

Time (s)	Kinard e Allen	MacMahon	CORE
t = 2	1.3382	1.34128	1.33859
t = 4	2.2285	2.254	2.22955
t = 6	5.5823	5.77368	5.58735
t = 8	4.2789×10^1	4.66422×10^1	4.28924×10^1
t = 9	4.8752×10^2	5.603×10^2	4.90165×10^2

Tabella 13: CORE per $\rho = 0.0007t$, rampa.

Il Modello BEFD

L'ultimo metodo che analizzeremo è un metodo non più esplicito come i precedenti, ma implicito, ed è qui che risiede la sua forza e accuratezza. Il metodo in questione si chiama BEFD (Backward Euler Finite Difference), suggerito dal paper [6] e che formalmente è un metodo di Eulero Implicito applicato al Sistema PKE [**?**] e risulta molto accurato nella sua soluzione specialmente nel caso non lineare, dal momento che viene aggiunta un'ottava equazione al sistema:

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

Nella quale $\rho_0(t)$ è la mia reattività nel tempo, quindi anche non costante, e B è il valore assoluto del coefficiente di temperatura della reattività, permettendo quindi di simulare una evoluzione del sistema anche a temperatura non costante.

Scrivendo questo sistema sotto forma di matrice possiamo vederlo come:

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

con:

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

A questo punto applichiamo il metodo delle differenze finite all'indietro per ottenere la forma generale del nostro metodo:

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

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

Da cui è possibile ottenere la soluzione del sistema all'istante di tempo successivo attraverso un metodo di punto fisso.

Inoltre anche in questo algoritmo è stato applicata l'estrapolazione di Richardson al risultato, in modo tale da ottenere un valore iniziale più affidabile per il calcolo dell'istante ancora successivo, tale metodo consiste nel risolvere la stessa equazione precedente ma a intervalli sempre più corti di h, successivamente si ricostruisce la soluzione ottimale attraverso la seguente formula ricorsiva:

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

Nell'algoritmo è stato utilizzato $s = 14$, come suggerito dal paper

Qui di seguito propongo dei risultati ottenuti per diverse tipologie di inserzioni di reattività nel sistema, non è presente un benchmark per quanto riguarda le inserzioni costanti, le comparerò quindi agli algoritmi precedenti.

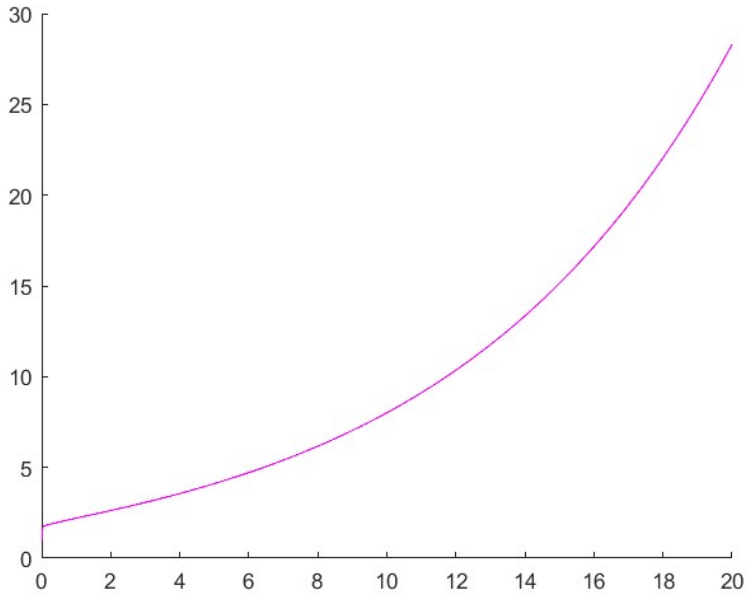

Figura 13: BEFD con $\rho = 0.003$, stadio subcritico.

Time (s)	Kinard e Allen	MacMahon	BEFD
t = 1	2.2098	2.2099	2.2055
t = 10	8.0192	8.0192	8.01508
t = 20	2.8297×10^1	2.8297×10^1	2.83056×10^1

Tabella 14: BEFD per $\rho = 0.003$ e $h = 0.01$

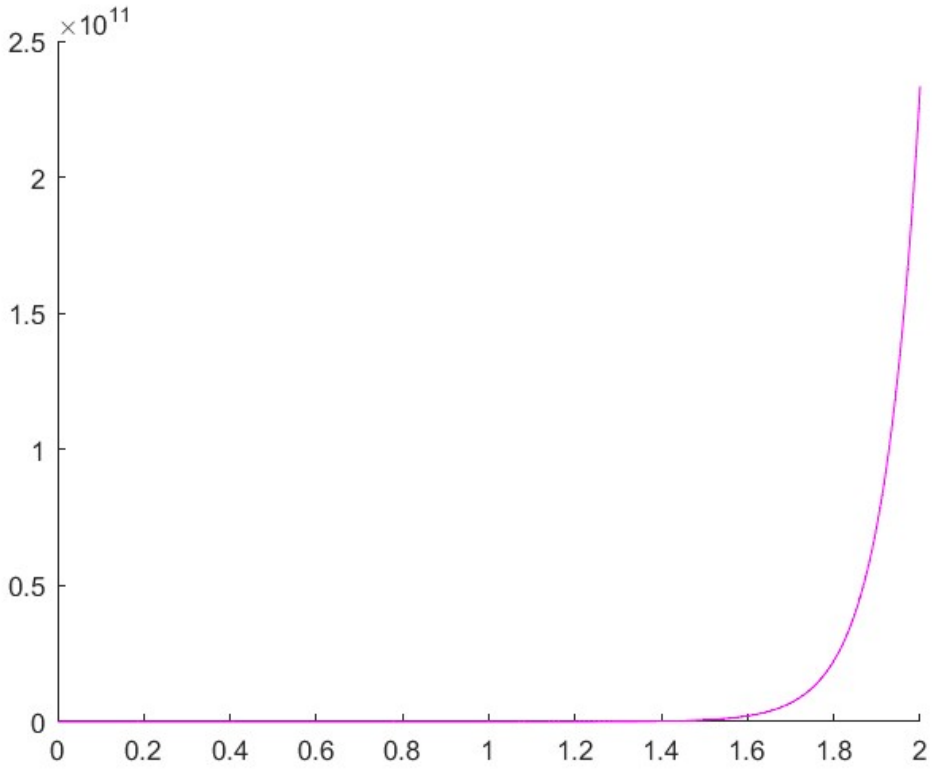

Figura 14: BEFD con $\rho = 0.007$, stadio critico.

Time (s)	Kinard e Allen	MacMahon	BEFD
t = 0.01	4.5088	4.5086	4.49007
t = 0.5	5.3459 $\times 10^3$	5.3447 $\times 10^3$	5.4673 $\times 10^3$
t = 2	2.0591 $\times 10^{11}$	2.0566 $\times 10^{11}$	2.33318 $\times 10^{11}$

Tabella 15: BEFD per $\rho = 0.007$ e $h = 0.01$

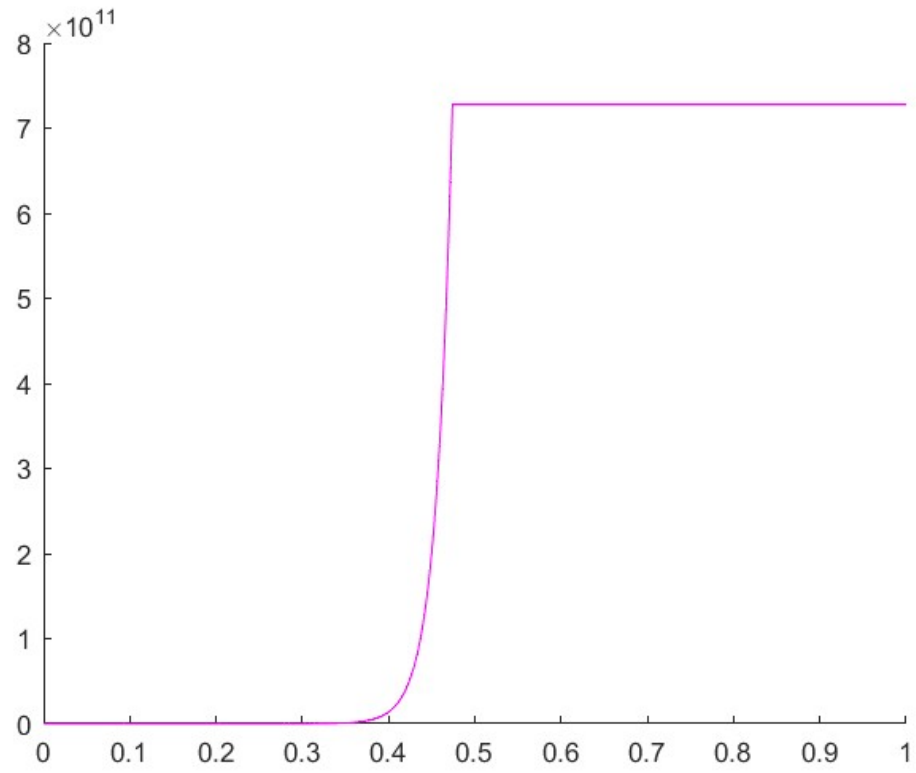

Figura 15: BEFD con $\rho = 0.008$, stadio supercritico

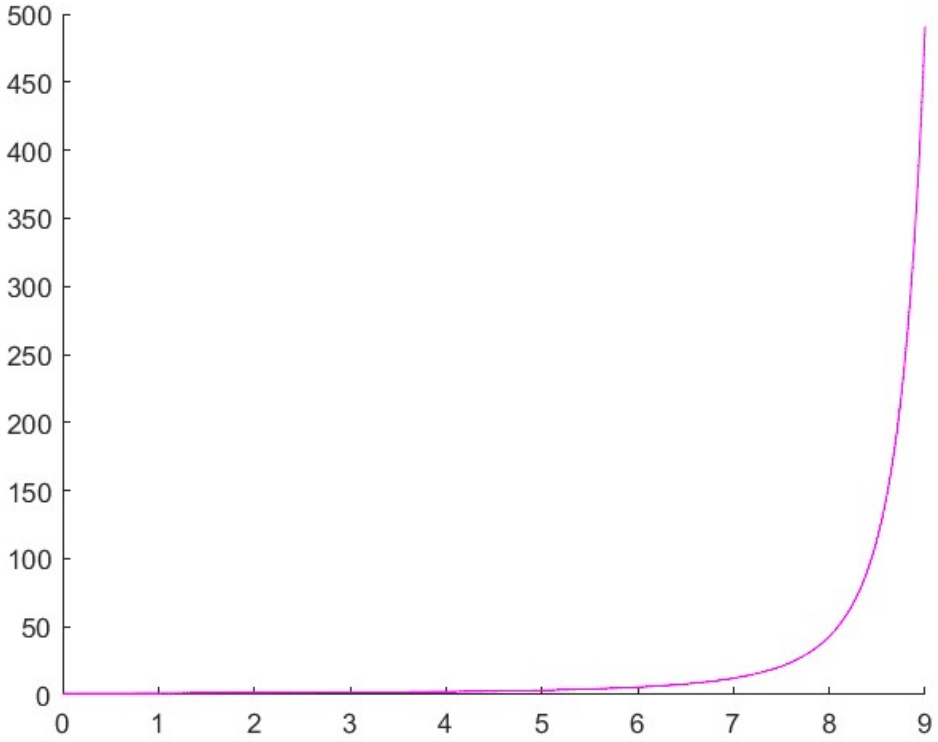

Figura 16: BEFD con $\rho = 0.0007t$, rampa.

Benchmark e Risultati

Prima di passare al benchmark con tutti i modelli soffermiamoci su un caso particolare di reattività sinusoidale: $\rho = \rho_0 \sin\left(\frac{\pi t}{T}\right)$ con $\rho_0 = 0.005333, T = 50s$, inoltre: $\Lambda = 10^{-7}$, che corrisponde al tempo di generazione dei neutroni, un lambda molto piccolo, come in questo caso corrisponde ad un reattore molto "rapido", al contrario delle simulazioni effettuate precedentemente.

Nella figura possiamo osservare che tutti i modelli, con un adeguato passo, riescono a simulare accuratamente la densità di neutroni, meno l'algoritmo basato sull'espansione di Taylor del sistema. Tale risultato è dovuto alla velocità di generazione del reattore, infatti, il passo per questi algoritmi deve essere dell'ordine di grandezza di Λ e ciò fa lievitare enormemente i tempi di computazione, rendendo l'algoritmo inutilizzabile per reattori "rapidi".

Passo: $\log_{10}(h)$	Tempo (s)	Errore: $log_{10}(\|err\|)$
0	0.056725	303
-1	0.129831	299
-2	0.973832	303
-3	26.223967	303
-4	N.A.	303

Tabella 16: Errore metodo Taylor per reattività sinusoidale

Gli altri algoritmi invece riescono a simulare anche questo tipo di reattori:

	PCA	CORE	BEFD
Tempo computazione (s)	0.176034	0.191041	258.0588
t = 10s	0.08%	0.133%	0.136%
t=40s	0.022%	0.019%	0.233%
t=100s	0.022%	0.043%	0.275%

Tabella 17: Errore relativo e tempo computazionale per sinusoide, h = 0.01

Osserviamo quindi che l'algoritmo BEFD, che utilizza una modifica di un metodo di Eulero indietro, risulta essere il meno preciso. Tuttavia questo algoritmo dovrebbe risultare migliore, dal momento che implementa anche una estrapolazione di Richardson per miglirare la precisione, questa differenza potrebbe essere legata al linguaggio in cui è stato scritto il codice originale, probabilmente FORTRAN.

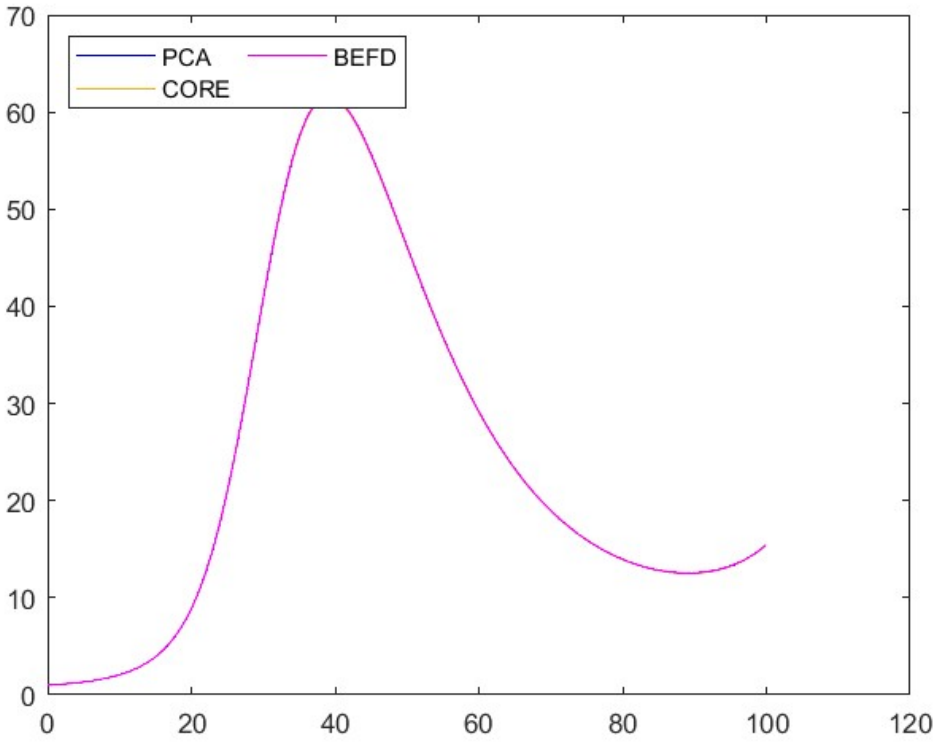

Figura 17: Reattività sinusoidale per tutti gli algoritmi

Come Benchmark finale considereremo un caso più realistico di reattore nucleare, ossia nel quale la reattività non sia indipendente ma sia funzione del numero di Neutroni nel sistema, tale tipologia si sistema viene chiamato feedback. La scelta di utilizzare questo benchmark deriva dall'assenza, nella letteratura di uno studio delle soluzioni delle Point Kinetics Equations in tale caso, ad eccezione dell'algoritmo BEFD sviluppato in [6], su cui ci baseremo per adattare anche gli altri algoritmi trattati precedentemente.

Per l'algoritmo ci rifacciamo al paper [7], dove viene illustrato il benchmark di feedback adiabatico, per prima cosa, utilizzeremo i seguenti λ e β:

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

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

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

Inoltre, il feedback adiabatico sarà rappresentato dall'equazione differenziale:

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

Nel quale $\rho_0(t) = 0.1t$ è una rampa e $\alpha = 10^{-11}$. Rifacendoci ai risultati di [7], possiamo ottenere i seguenti risultati:

Tempo	PCA	Taylor	Taylor + Mod	CORE	BEFD	Reference
0.001s	N.A.	1.00096	1.00097	1.00114	1.00098	1.0009554
0.010s	1.12654	1.06888	1.06891	1.06902	1.06897	1.06882056
0.100s	2.58882e01	1.81028e01	1.81065e01	1.81189e01	1.81227e01	1.81008431e01
0.150s	1.1409e04	4.99749e03	5.01479e03	5.02283e03	5.04028e03	5.01107924e03
0.200s	6.92018e08	1.82275e08	1.84972e08	1.85455e08	1.88209e08	1.84748246e08
0.250s	1.13922e09	3.18474e09	3.13909e09	3.13212e09	3.10963e09	3.17939153e09

Tabella 18: Comparazione dell'accuratezza dei metodi discussi finora, passo $h = 10^{-5}$ (PCA solo con h=0.005).

PCA	Taylor	Taylor + Modifiche	CORE	BEFD
0.106s	4.89s	4.89s	3.17s	1244s

Tabella 19: Tempo computazionale, passo h=0.001.

Riferimenti bibliografici

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

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

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

[4] A. Pierson D. McMahon. A taylor series solution of the reactor point kinetics equations.

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

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

[7] R. Furfaro 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.

Codici MATLAB

```
function elapsedTime = constTest()
    clc;clear all; %close all;
    addpath('C:\Users\alewi\Desktop\Tesi\MATLAB\BDEF - Ganapol')
    addpath('C:\Users\alewi\Desktop\Tesi\MATLAB\CORE - Quintero Leyva')
    addpath('C:\Users\alewi\Desktop\Tesi\MATLAB\PCA - Kinard')
    addpath('C:\Users\alewi\Desktop\Tesi\MATLAB\TS - McMahon')

    %{
    n = 6;
    tic
    [T_pca,N_pca,color_pca] = initPCA(n);
    toc
    tt(1) = toc;
    tic
    [T_ts, N_ts, color_ts, N1_ts, color1_ts] = initTS(n);
    toc
    tt(2) = toc;
    tic
    [T_core, N_core, color_core]=initCORE(n);
    toc
    tt(3) = toc;
    tic
    [T_befd, N_befd, color_befd] = initBEFD(n);
    toc
    tt(4) = toc;

    xlabel('Time')
    ylabel('Neutron density')
    loglog(T_pca,N_pca, Color =color_pca)
    hold on
    loglog(T_befd,N_befd, Color = color_befd)
    loglog(T_ts,N_ts, Color = color_ts)
    loglog(T_ts,N1_ts, Color = color1_ts)
```

```
34      loglog(T_core, N_core, Color = color_core)
35      legend({'PCA', 'BEFD', 'TS', 'TS + Modifiche', 'CORE
           '}, 'Location','southeast','NumColumns',2)
36      hold off
37
38      %}
39      n = 0;
40      h = 1e-5;
41      elapsedTime = [];
42      tic
43      [T_pca,N_pca,color_pca] = initPCA(n,h*500);
44      toc
45      elapsedTime(1) = toc;
46      tic
47      [T_ts, N_ts, color_ts, N1_ts, color1_ts] = initTS(n,
           h);
48      toc
49      elapsedTime(2) = toc;
50      tic
51      [T_core, N_core, color_core]=initCORE(n,h);
52      toc
53      elapsedTime(3) = toc;
54      tic
55      [T_befd, N_befd, color_befd] = initBEFD(n,h);
56      toc
57      elapsedTime(4) = toc;
58
59
60      xlabel('Time')
61      ylabel('Neutron density')
62      semilogy(T_pca,N_pca, Color =color_pca)
63      hold on
64      semilogy(T_befd,N_befd, Color = color_befd)
65      semilogy(T_ts,N_ts, Color = color_ts)
66      semilogy(T_ts,N1_ts, Color = color1_ts)
67      semilogy(T_core, N_core, Color = color_core)
68      legend({'PCA', 'BEFD', 'TS', 'TS + Modifiche', 'CORE
           '}, 'Location','southeast','NumColumns',2)
69  end
```

PCA

L'intero codice per il metodo PCA, è stato scritto da Kinard e Allen in [2]; mi limiterò ad aggiungere solo il codice che ho modificato personalmente.

```
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 = 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);
```

```
45
46      % Construct matrices for computation
47      for k = 1:m
48          d_hat(k,k) = exp(d(k)*h);
49          big_d(k,k) = d(k);
50      end
51
52      f_hat(1) = source;
53      big_d_inv = zeros(m,m);
54      for k =1:m
55          big_d_inv(k,k) = 1/big_d(k,k);
56      end
57
58      % Compute next time step
59      x = (Y * d_hat * Y_inv)*(x + (Y*big_d_inv*Y_inv*
            f_hat)) - (Y*big_d_inv*Y_inv*f_hat);
60
61      % Store results in a matrix
62      for j = 1:m
63          result(1,i) = time;
64          result(j+1,i) = x(j);
65      end
66      i/iterations*100
67      % Update counters
68      i = i + 1;
69  end
70
71  y=result;
```

Taylor Series Expansion

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

```
6
7       addpath('C:/Users/alewi/Desktop/Tesi/MATLAB/PCA -
           Kinard')
8       addpath('C:/Users/alewi/Desktop/Tesi/MATLAB/BDEF -
           Ganapol')
9
10      t0 = 0;
11
12      %Parametri propri del reattore considerato
13      [lambda, beta, LAMBDA, rho_0, alpha, q, t_max] =
           cases(n);
14
15      sum(beta)
16      %Valori iniziali per risolvere il problema di Cauchy
            associato al sistema
17      %di EDO.
18      N0 = 1;
19      C0 = beta./(LAMBDA*lambda);
20      y0 = [N0; C0];
21      tic
22      [t, N, C, N_1, C_1] = taylorPkeSolver(LAMBDA, lambda
           , beta, y0, rho_0, t0,t_max, h, alpha);
23      toc
24      color = 'r';
25      color1 = 'g';
26
27      plot(t,N_1,'r')
28  end
```

```
1   function [t,N, C,N_1, C_1] = taylorPkeSolver(LAMBDA,
       lambda, beta, y0, rho_0, t0,t_max, h, alpha)
2       n = ceil((t_max-t0)/h);
3
4       BETA = sum(beta);
5
6       N = [y0(1), zeros(1, n-1)];
7       C = [y0(2:end), zeros(length(y0)-1, n-1)];
8
9       N_1 = [y0(1), zeros(1, n-1)];
10      C_1 = [y0(2:end), zeros(length(y0)-1, n-1)];
11
```

```
12      u_0 = y0;
13
14      t(1) = t0;
15      figure()
16
17      %plot((1:n)*h, rho((1:n)*h))
18      for i=2:n
19          t(i) = i*h;
20
21          rho = @(t) rho_0(t) + alpha*h*sum(N);
22          rho1 = @(t) rho_0(t) + alpha*h*sum(N_1);
23
24          D = diag([-lambda]);
25          v = [lambda'];
26          w = [beta/LAMBDA];
27          A = @(h) [(rho(t(i)+h)-BETA)/LAMBDA, v; w, D];
28
29          u_0 = [N(i-1);C(:,i-1)];
30
31          u_01 = [N_1(i-1);C_1(:,i-1)];
32
33          myRichardson(A, zeros(length(y0),1),[N_1(i-1);
               C_1(:,i-1)], h, 5, 1e-9);
34
35          N(i) = (1+h*(rho(i*h)-BETA)/LAMBDA)*u_0(1) + h*
               sum(lambda.*u_0(2:end));
36          C(:,i) = (h/LAMBDA*beta)*u_0(1) + (1-h*lambda).*
               u_0(2:end);
37
38
39          N_1(i) = (1+h*(rho1(i*h)-BETA)/LAMBDA +h^2/2*(((
               rho1(i*h)-BETA)/LAMBDA)^2 + ...
40                      sum(lambda.*beta/LAMBDA)))*u_01(1) +
                           h*sum(lambda.*u_01(2:end)) + ...
41                      h^2/2*((rho1(i*h)-BETA)/LAMBDA)*sum(
                           lambda.*u_01(2:end)) - h^2/2*sum(
                           lambda.^2.*u_01(2:end));
42
43          C_1(:,i) = (h/LAMBDA*beta+h^2/2*((beta*(rho1(i*h
               )-BETA)/LAMBDA^2)-lambda.*beta/LAMBDA))*u_01
```

```
            (1)+...
44                  (1-h*lambda-(h*lambda).^2/2).*u_01(2:
                       end) + h^2/2*beta/LAMBDA*sum(lambda
                       .*u_01(2:end));
45
46          round(i/n*100,2)
47      end
48
49  end
```

```
1   function y = myRichardson(A, q, u_0, h, n, toll)
2       R(:, 1, 1) = (1+h*A(h))*u_0;
3       for i=1:100
4           h = h/2;
5
6           R(:, i + 1, 1) = (1+h*A(h))*u_0;
7
8           for j=1:i
9               R(:,i + 1, j + 1) = (2^j*R(:,i + 1, j) - R(:,i
                    , j))/(2^j - 1);
10          end
11
12          if ( norm( R(:,i + 1, i + 1) - R(:,i, i) ) < toll
                )
13              break;
14          elseif ( i == 100 )
15              error( 'Richardson extrapolation failed to
                    converge' );
16          end
17
18      end
19      y = R(:,i+1, i+1);
```

CORE

```
1   function [t, N, color]=initCORE(n,h)
2   %In questo codice MATLAB verra costruito l'algoritmo
        CORE come descritto
3   %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);
```

```
11
12      t = 0:h:t_max;
13
14      for i = 2:(n+1)
15
16          rho = @(t) rho_0(t) + alpha*h*sum(N);
17
18          C_dot(i,:) = (beta./LAMBDA)'*N(i-1) - lambda'.*C
                (i-1,:);
19
20          S_k = inhour(lambda, LAMBDA, beta, rho(i*h),
                zeros(length(beta)+1,1));
21
22          R_k = R_calculation(N(i-1), C_dot(i,:), S_k,
                beta, lambda, LAMBDA);
23          R_k = R_k';
24
25          N(i) = sum(R_k.*exp(S_k*h));
26
27          C(i,:) = C_calculation(R_k, S_k, C(i-1,:), h,
                beta, lambda, LAMBDA);
28
29          ceil(i/n*100)
30
31      end
32  end
```

```
1   function R_k = R_calculation(N, C_dot, S_k, beta, lambda
        , LAMBDA)
2       R_k = zeros(1, length(S_k));
3       for i = 1:length(S_k)
4           s1 = 0;
5           for j=1:length(beta)
6               add = beta(j)/(S_k(i) + lambda(j));
7               s1 = s1 + add;
8           end
9           s1 = s1/LAMBDA;
10          s1 = s1+1;
11
12
13          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
```

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];
```

```
7
8       tmin = 0;
9
10      n=ceil((tmax-tmin)/h);
11
12      toll = 1e-10;
13      T(1) = h;
14      u(:,1) = u_0;
15      R = [];
16
17      for i = 2:(n)
18
19          T(i) = (i)*h;
20
21          rho = @(n, t) rho_0(t)+alpha*(sum(u(1,:)*h)+n*h)
                ;
22
23          D = diag([-lambda; 0]);
24          v = [lambda', 0];
25          w = [beta/L; alpha];
26          A = @(n,t) [(rho(n,t)-sum(beta))/L, v; w, D];
27
28
29          F = @(y) -(eye(length(u_0)) - h*A(y(1),(i+1)*h))
                *y + u(:,i-1) + h*q;
30          options = optimoptions('fsolve','TolFun', toll,'
                Display','off');
31
32          %Richardson Extrapolation
33          u_0 = myRichardson_Benchmark_fun(A, T(i), q, u
                (:,i-1), h, 10, toll);
34          u(:,i) = fsolve(F, u_0, options);
35
36          R(i) = rho(u(1,i),i*h);
37
38          round(i/n*100,2)
39
40      end
41
42      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 = 0.25;

    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;

        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;
```

```
35
36          rho_init = 0.007;
37          rho_0 = @(t) rho_init;
38
39          alpha = 0;
40          q = zeros(8,1);
41          tmax=2;
42
43      elseif(n==3)
44          %Step Insertion of rho = 0.008
45          beta = [0.000266, 0.001491, 0.001316, 0.002849,
                0.000896, 0.000182]';
46          lambda = [0.0127, 0.0317, 0.115, 0.311, 1.4,
                3.87]';
47          L = 2e-5;
48
49          rho_init = 0.008;
50          rho_0 = @(t) rho_init;
51
52          alpha = 0;
53          q = zeros(8,1);
54          tmax=1;
55
56
57      elseif(n==4)
58          %Ramp Insertion of 0.1$
59          beta = [0.000266, 0.001491, 0.001316, 0.002849,
                0.000896, 0.000182]';
60          lambda = [0.0127, 0.0317, 0.115, 0.311, 1.4,
                3.87]';
61          L = 2e-5;
62
63          rho_init = 0.007*0.1;
64          rho_0 = @(t) rho_init*t;
65
66          alpha = 0;
67          q = zeros(8,1);
68          tmax=9;
69
70      elseif(n==5)
```

```
71          %Sinusoidal Insertion T=50, rho_0 = 0.0053333
72          beta = 0.0079;
73          lambda = 0.077;
74          L = 1e-8;
75
76          rho_init = 0.0053333;
77          T=50;
78          rho_0 = @(t) rho_init*sin(pi*t/T);
79
80          alpha = 0;
81          q = zeros(3,1);
82          tmax=100;
83
84
85      elseif(n==6)
86          %Step Insertion of rho = 0.007 with feedback
87          lambda = [1.24e-2; 3.05e-2; 1.11e-1; 3.01e-1;
                1.13; 3];
88          beta = [2.1e-4; 1.41e-3; 1.27e-3; 2.55e-3; 7.4e
                -4; 2.7e-4];
89          L = 5e-5;
90
91          rho_init = sum(beta)*1.5;
92          rho_0 = @(t) rho_init;
93
94          alpha = -2.5e-6;
95          q = zeros(8,1);
96          tmax=100;
97
98
99      elseif(n==7)
100         %Ramp Insertion of 0.1$ with feedback
101         lambda = [1.24e-3; 3.05e-2; 1.11e-1; 3.01e-1;
                1.13; 3];
102         beta = [2.1e-4; 1.41e-3; 1.27e-3; 2.55e-3; 7.4e
                -4; 2.7e-4];
103         L = 5e-5;
104
105         ramp_rate = 0.1;
106         rho_0 = @(t) ramp_rate*t;
```

```
107
108         alpha = -1e-11;
109         q = zeros(8,1);
110         tmax=10;
111
112     end
113 end
```