Bateman equations for Xe poisoning

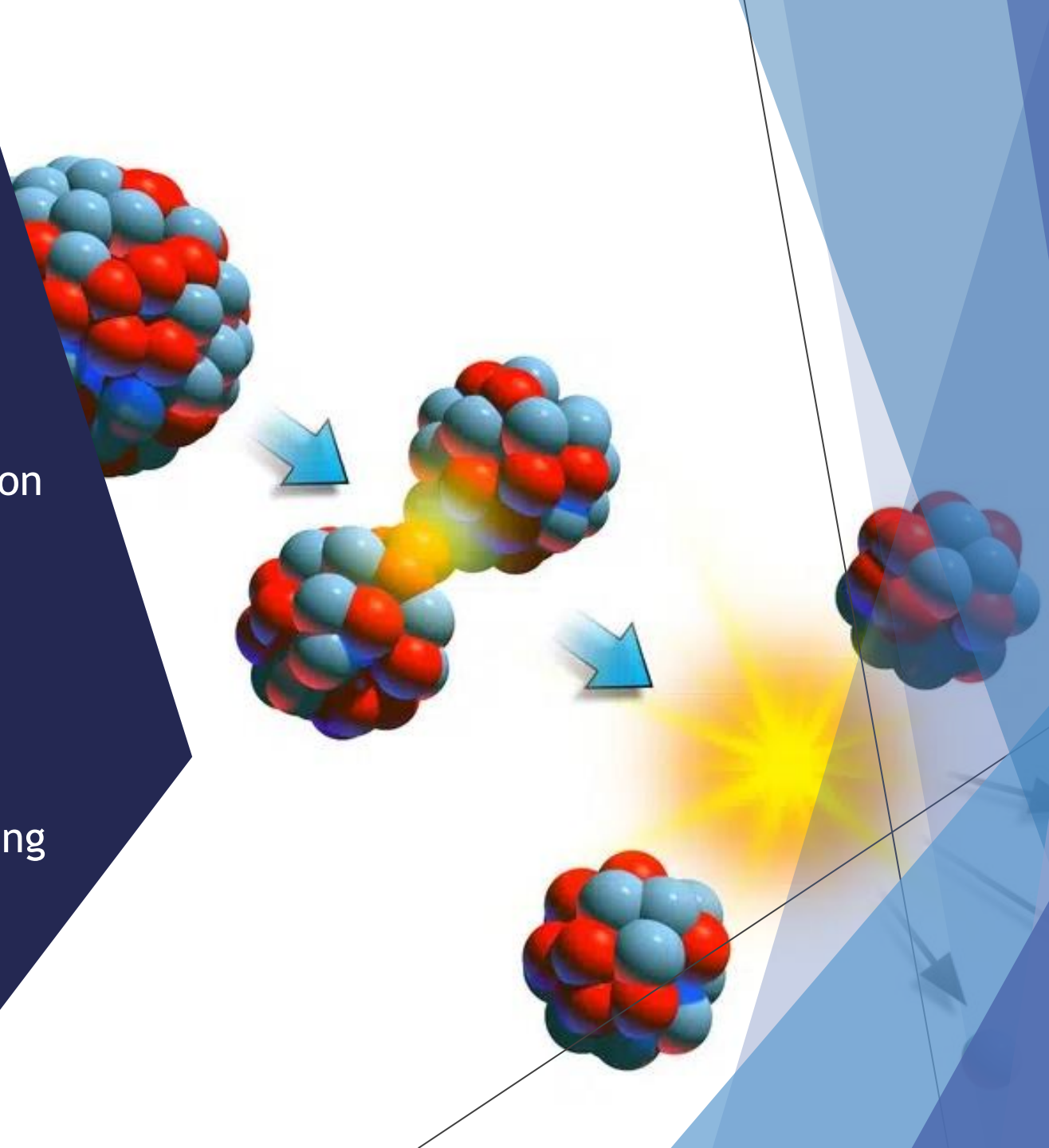
Matilde Dondi

Theoretical and Numerical aspects of Nuclear Physics

Xenon-135 poisoning in nuclear reactors

Xenon-135 :

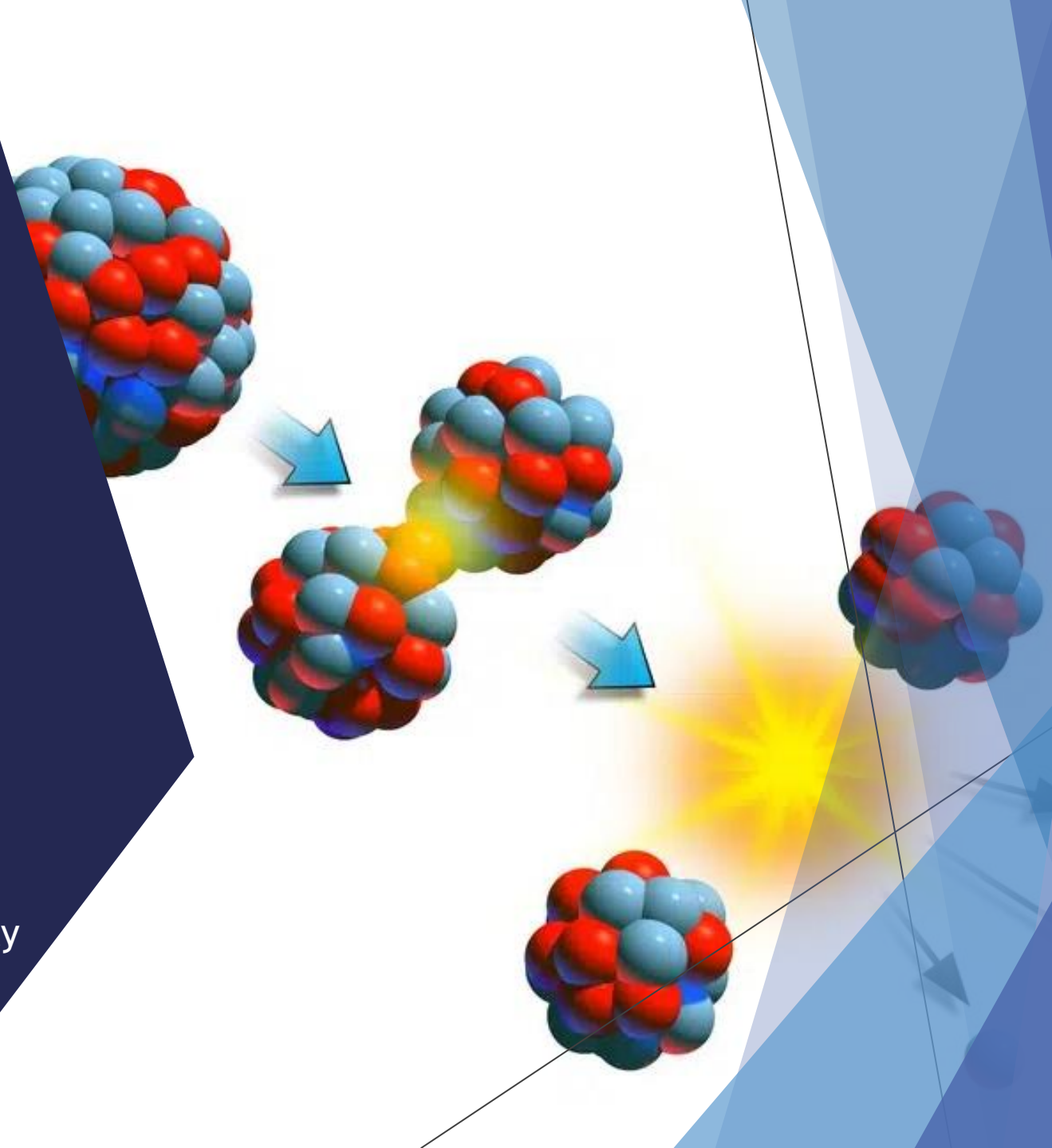
- Fission product with large absorption cross-section → ~ 2.75 Mbarns
- Comes from Iodine-135 decay
- Xe neutron absorption: reduces reactor reactivity → Xenon poisoning

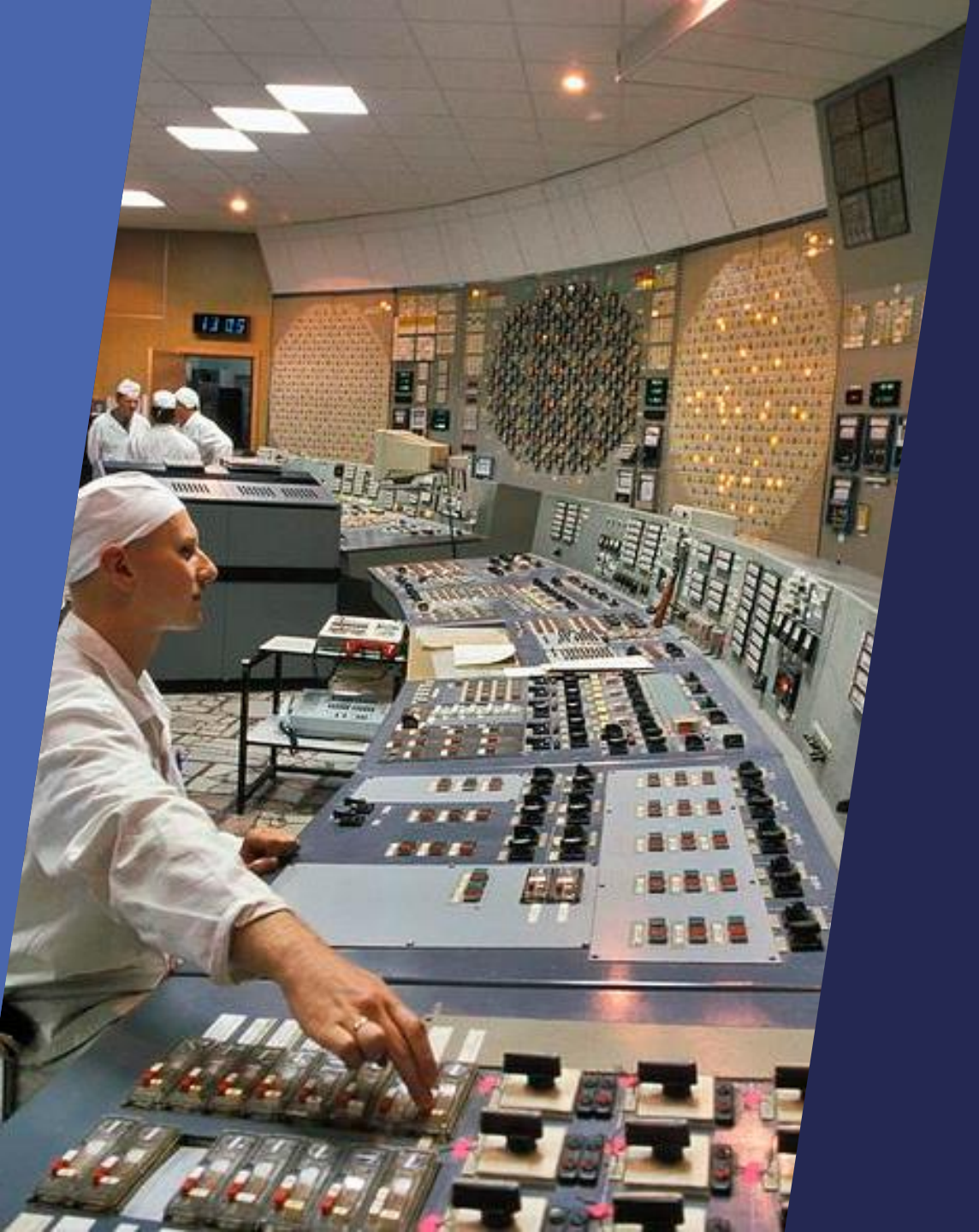


Xenon-135 poisoning in nuclear reactors

After Reaction Shutdown:

- Xe accumulates from I decay (6.7 h half life)
  → peak around 10-15 h
- Drop in reactivity: Iodine «pit»
  → difficult to restart the reactor
- Xe decay (9.2 h half-life): reactivity increase again
  → risk of criticality or supercriticality





Chernobyl disaster: the role of Xe

25 April 1986: Scheduled test on Reactor Shutdown

- 01:06: Gradual power reduction begins
- By day shift: Power at 50% of nominal 3,200 MW
- 14:00: Kiev grid requests halt due to peak demand
- 23:04: Shutdown resumes
- 00:05 (26 April): Power reduced to 720 MW

Chernobyl disaster: the role of Xe

Xe poisoning in the reactor

- Reactor was run at **low power** for a prolonged period → Xe-135 buildup → reactivity drop
- Other factors caused another power drop to near-shutdown state (≤ 30 MW)
- To increase power, operators withdrew **too many control rods** → increased reactivity, masked by Xe poisoning
- Around 1:00, power level reached 200 MW
- Very dangerous and unstable reactor conditions that later led to the accident

Chain reaction

$$^{135}_{52}\mathrm{Te} \rightarrow \ ^{135}_{53}\mathrm{I} \rightarrow \ ^{135}_{54}\mathrm{Xe} \rightarrow \ ^{135}_{55}\mathrm{Cs} \rightarrow \ ^{135}_{56}\mathrm{Ba}$$

$$\frac{\mathrm{d}}{\mathrm{dt}}\mathrm{I(t)} = \lambda_{\mathrm{T}}\mathrm{Te(t)} - \lambda_{\mathrm{I}}\mathrm{I(t)}$$

Te half-life ~ 11 s: short enough to consider I as **primary fission product**

$$\frac{\mathrm{d}}{\mathrm{dt}}\mathrm{Xe(t)} = \lambda_{\mathrm{I}}\mathrm{I(t)} - \lambda_{\mathrm{Xe}}\mathrm{Xe(t)}$$

Bateman equation system

$$\frac{d}{dt}I(t) = \gamma_I \Sigma_f \phi - \lambda_I I(t)$$

$$\frac{d}{dt}Xe(t) = \gamma_{Xe} \Sigma_f \phi + \lambda_I I(t) - \lambda_{Xe} Xe(t) - \sigma_{aXe} \phi Xe(t)$$

- λ_I and λ_{Xe} decay constants
- Fission contribution:

  γ_I and γ_{Xe} fission yields, $\Sigma_f \phi$ fission rate (ϕ neutron flux)
- Thermal neutron absorption:

  σ_{Xe} microscopic absorption cross section (too small for I)

Ordinary differential equations (ODE)
Initial value problem

$$\frac{d}{dt}I(t) = \gamma_I \Sigma_f \phi - \lambda_I I(t)$$

$$\frac{d}{dt}Xe(t) = \gamma_{Xe} \Sigma_f \phi + \lambda_I I(t) - \lambda_{Xe} Xe(t) - \sigma_{aXe} \phi Xe(t)$$

Long-running reactors at equilibrium

$$\frac{d}{dt}I(t) = 0$$

$$\frac{d}{dt}Xe(t) = 0$$

Initial conditions:

$$I_0 = \frac{\gamma_I \Sigma_f \phi}{\lambda_I}$$

$$Xe_0 = \frac{\Sigma_f \phi (\gamma_I + \gamma_{Xe})}{\lambda_{Xe} + \sigma_{aXe} \phi}$$

Ordinary differential equations (ODE)
Initial value problem

$$\frac{\mathrm{d}}{\mathrm{dt}}\mathrm{I(t)} = \gamma_I \Sigma_f \phi - \lambda_\mathrm{I}\mathrm{I(t)}$$

$$\frac{\mathrm{d}}{\mathrm{dt}}\mathrm{Xe(t)} = \gamma_{Xe} \Sigma_f \phi + \lambda_\mathrm{I}\mathrm{I(t)} - \lambda_\mathrm{Xe}\mathrm{Xe(t)} - \sigma_\mathrm{aXe}\phi\mathrm{Xe(t)}$$

Reactor shutdown:

$$\phi = 0$$

Ordinary differential equations (ODE)
Initial value problem

Initial conditions:

$$\mathrm{I}_0 = \frac{\gamma_I \Sigma_f \phi}{\lambda_I}$$

$$\mathrm{Xe}_0 = \frac{\Sigma_f \phi(\gamma_I + \gamma_{Xe})}{\lambda_{Xe} + \sigma_{aXe}\phi}$$

$$\frac{\mathrm{d}}{\mathrm{dt}}\mathrm{I(t)} = -\lambda_\mathrm{I}\mathrm{I(t)}$$

$$\frac{\mathrm{d}}{\mathrm{dt}}\mathrm{Xe(t)} = \lambda_\mathrm{I}\mathrm{I(t)} - \lambda_\mathrm{Xe}\mathrm{Xe(t)}$$

Analytical solution

$$\frac{d}{dt}I(t) = -\lambda_I I(t)$$

$$\frac{d}{dt}Xe(t) = \lambda_I I(t) - \lambda_{Xe} Xe(t)$$

$$\downarrow$$

$$I(t) = I_0 e^{-\lambda_I t} \longrightarrow \frac{d}{dt}Xe(t) = \lambda_I I_0 e^{-\lambda_I t} - \lambda_{Xe} Xe(t)$$

Analytical solution
Integrating factor technique

$$\frac{\mathrm{d}}{\mathrm{dt}}\mathrm{Xe(t)} + \lambda_{\mathrm{Xe}} Xe(t) = \lambda_{\mathrm{I}} \mathrm{I}_0 \mathrm{e}^{-\lambda_I \mathrm{t}} \longrightarrow \text{Of the form} \quad \frac{\mathrm{dy}}{\mathrm{dx}} + \mathrm{Py} = \mathrm{Q}$$

Multiply for the integrating factor:

$$\mu = e^{\int P dx} = e^{\lambda_{Xe} t}$$

$$\downarrow$$

$$e^{\lambda_{Xe} t} \frac{\mathrm{d}}{\mathrm{dt}}\mathrm{Xe(t)} + e^{\lambda_{Xe} t} \lambda_{\mathrm{Xe}} Xe(t) = \lambda_{\mathrm{I}} \mathrm{I}_0 \mathrm{e}^{(\lambda_{Xe} - \lambda_I)\mathrm{t}} \longrightarrow \frac{\mathrm{d}}{\mathrm{dt}}(\mathrm{Xe(t)} e^{\lambda_{Xe} t}) = \lambda_{\mathrm{I}} \mathrm{I}_0 \mathrm{e}^{(\lambda_{Xe} - \lambda_I)\mathrm{t}}$$

Analytical solution
Integrating factor technique

$$\frac{d}{dt}\left(Xe(t)e^{\lambda_{Xe}t}\right) = \lambda_I I_0 e^{(\lambda_{Xe}-\lambda_I)t}$$

Integrating

$$Xe(t)e^{\lambda_{Xe}t} = \frac{\lambda_I}{\lambda_{Xe}-\lambda_I} I_0 e^{(\lambda_{Xe}-\lambda_I)t} + C \longrightarrow Xe(t) = \frac{\lambda_I}{\lambda_{Xe}-\lambda_I} I_0 e^{-\lambda_I t} + Ce^{-\lambda_{Xe}t}$$

With

$$Xe_0 = \frac{\Sigma_f \phi(\gamma_I + \gamma_{Xe})}{\lambda_{Xe} + \sigma_{aXe}\phi}$$

Analytical solution

Final solution:

$$\mathrm{I(t)} = \frac{\gamma_I \Sigma_f \phi}{\lambda_I} \mathrm{e}^{-\lambda_I t}$$

$$\mathrm{Xe(t)} = \Sigma_f \phi \left[\frac{\gamma_I + \gamma_{Xe}}{\lambda_{Xe} + \sigma_{aXe}\phi} e^{-\lambda_{Xe} t} + \frac{\gamma_I}{\lambda_I - \lambda_{Xe}} (e^{-\lambda_{Xe} t} + e^{-\lambda_I t}) \right]$$

Costants and poisoning

Symbol	Name	Value
γ_I	Fission yield of I-135	0.061
γ_X	Fission yield of Xe-135	0.003
λ_I	Decay constant of I-135 (s^{-1})	2.874×10^{-5}
λ_X	Decay constant of Xe-135 (s^{-1})	2.027×10^{-5}
σ_{aX}	Microscopic absorption cross-section of Xe-135 (cm^2)	2.75×10^{-18}
ν	# of neutrons released per fission	2.3
ϕ	Thermal neutron flux $(\mathrm{cm}^{-2}\mathrm{s}^{-1})$	4.42×10^{20}
Σ_f	Macroscopic absorption cross-section (cm^2)	0.008

$$\rho_{Xe}(t) = -\frac{\sigma_{aXe} Xe(t)}{\nu \Sigma_f}$$

Zechuan Ding, Solving Bateman Equation for Xenon Transient Analysis Using Numerical Methods

Numerical solution: fourth – order Runge Kutta (RK4)

- RK4 method: numerical method to solve ODE with initial value

$$\frac{dy(t)}{dt} = f(t, y), \qquad y(t_0) = y_0$$

- Based on approximating the solution using Taylor series expansion, but without calculating high-order derivatives:
  - y_{n+1} is approximated by y_n plus the weighted average of four increments

$$\downarrow$$

  - Each increment is the product of an interval h and an estimated slope specified by function f at different points of the interval $[t_n, t_n + h]$

- Error of the order $O(h^4)$

Numerical solution: fourth – order Runge Kutta (RK4)

Increments:

$$\Delta y_1 = hf(x_n, y_n)$$

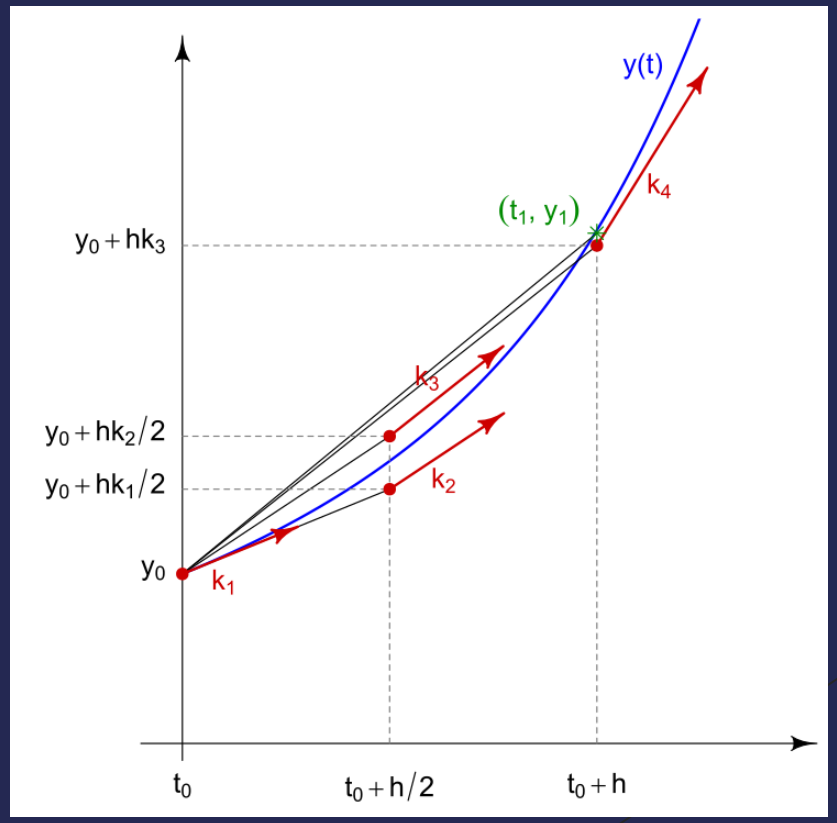
$$\Delta y_2 = hf\left(x_n + \frac{h}{2}, y_n + \frac{\Delta y_1}{2}\right)$$

$$\Delta y_3 = hf\left(x_n + \frac{h}{2}, y_n + \frac{\Delta y_2}{2}\right)$$

$$\Delta y_4 = hf(x_n + h, y_n + \Delta y_3)$$

$$\downarrow$$

$$y_{n+1} = y_n + \frac{1}{6}(\Delta y_1 + 2\Delta y_2 + 2\Delta y_3 + \Delta y_4)$$



Numerical solution: fourth – order Runge Kutta (RK4)

Increments:

$$\Delta y_1 = hf(x_n, y_n)$$

$$\Delta y_2 = hf\left(x_n + \frac{h}{2}, y_n + \frac{\Delta y_1}{2}\right)$$

$$\Delta y_3 = hf\left(x_n + \frac{h}{2}, y_n + \frac{\Delta y_2}{2}\right)$$

$$\Delta y_4 = hf(x_n + h, y_n + \Delta y_3)$$

$$\downarrow$$

$$y_{n+1} = y_n + \frac{1}{6}(\Delta y_1 + 2\Delta y_2 + 2\Delta y_3 + \Delta y_4)$$

For Xe poisoning case:

$$y = I(t)\,, f(t, y) = -\lambda_I I(t)$$

$$y = Xe(t)\,, f(t, y) = \lambda_I I(t) - \lambda_{Xe} Xe(t)$$

RK4: code implementation

Initial conditions

Increments calculation

New state

```python
def runge_kutta(f, y0, t0, tf, dt):
    """
    Solves a system of ordinary differential equations using the Runge-Kutta method.

    Args:
        f:  Equations to solve.
        y0: Initial state of the system.
        t0: Initial time.
        tf: Final time.
        dt: Time step for the integration.

    Returns:
        A list of tuples where each tuple contains the current time and the corresponding state of the system.
    """
    # Calculate the number of steps based on the total time and time step
    n = int((tf - t0) / dt)
    # Initialize the time and state variables
    t = t0
    y = y0[:]


    results = [(t, y)]

    # Perform the Runge-Kutta integration
    for _ in range(n):

        delta_y1 = [dt * dy for dy in f(y)]

        y2_app = [y[j] + delta_y1[j] / 2 for j in range(len(y))]
        delta_y2 = [dt * dy for dy in f(y2_app)]

        y3_app = [y[j] + delta_y2[j] / 2 for j in range(len(y))]
        delta_y3 = [dt * dy for dy in f(y3_app)]

        y4_app = [y[j] + delta_y3[j] for j in range(len(y))]
        delta_y4 = [dt * dy for dy in f(y4_app)]

        # Update the state
        y = [y[j] + (delta_y1[j] + 2 * delta_y2[j] + 2 * delta_y3[j] + delta_y4[j]) / 6 for j in range(len(y))]
        t += dt

        results.append((t, y))

    return results
```

Numerical solution: matrix exponential

- Matrix exponential method: ODE system

$$\frac{d\,\vec{y}(t)}{dt} = A\vec{y}, \qquad \vec{y}(t_0) = \vec{y}_0$$

- The solution is given by:

$$\vec{y}(t) = e^{(t-t_0)A}\vec{y}_0$$

- Each step is:

$$\begin{bmatrix} I_{n+1} \\ Xe_{n+1} \end{bmatrix} = e^{\Delta t A} \begin{bmatrix} I_n \\ Xe_n \end{bmatrix}, \qquad A = \begin{bmatrix} -\lambda_I & 0 \\ \lambda_I & -\lambda_{Xe} \end{bmatrix}$$

Matrix exponential: code implementation

Initial conditions

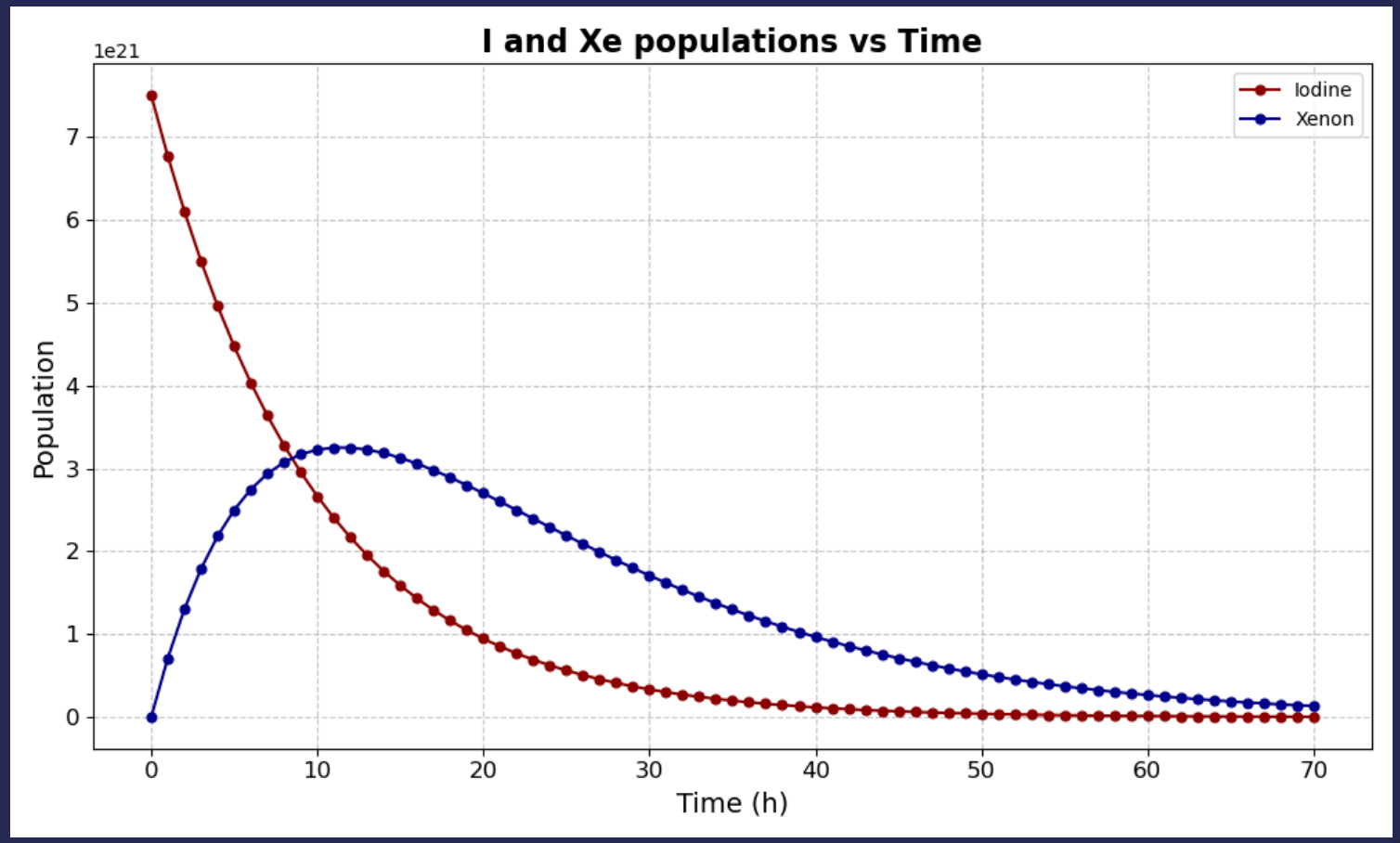
Matrix exponential calculation

New state

```python
import numpy as np
from scipy.linalg import expm
from parameters import par

def matrix_method(y0, t0, tf, dt):
    """
    Solves a system of ordinary differential equations using the matrix exponential method.

    Args:
        y0: Initial state of the system.
        t0: Initial time.
        tf: Final time.
        dt: Time step for the integration.

    Returns:
        A list of tuples where each tuple contains the current time and the corresponding state of the system.
    """
    # Calculate the number of steps based on the total time and time step
    n = int((tf - t0) / dt)
    # Initialize the time and state variables
    t = t0
    y = y0

    results = [(t, y)]

    # Define the matrix based on the differential equations and compute the matrix exponential
    A = np.array([
        [-par.LAMBDA_I, 0],
        [par.LAMBDA_I, -par.LAMBDA_XE]
    ])
    matrix_exp = expm(dt * A)

    # Perform the matrix exponential integration
    for _ in range(n):
        y = np.dot(matrix_exp, y)
        t += dt

        results.append((t, y))

    return results
```

Numerical solutions: results

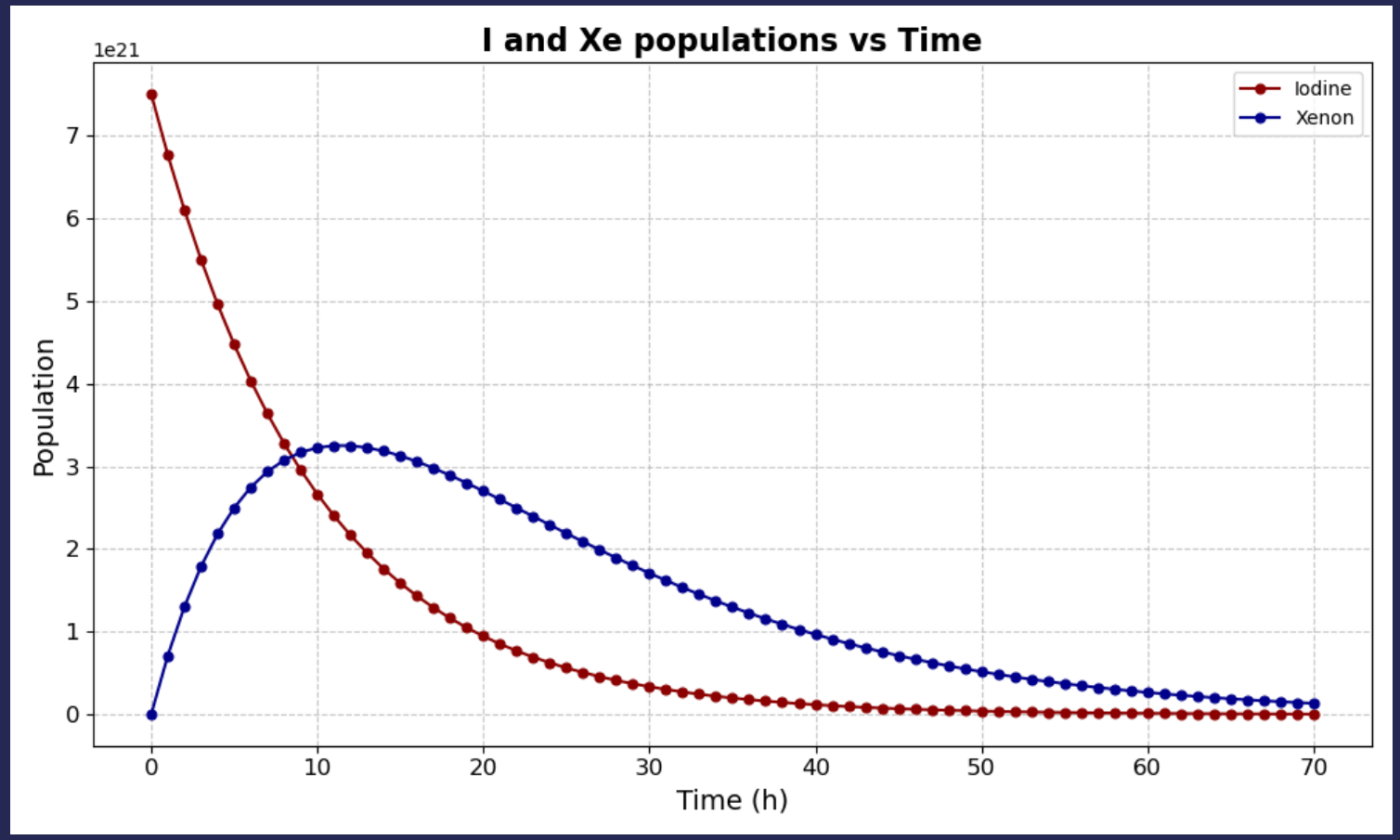
Time step of 3600 s, up to 252000 s (= 70 h)



Fourth order Runge-Kutta

Numerical solutions: results

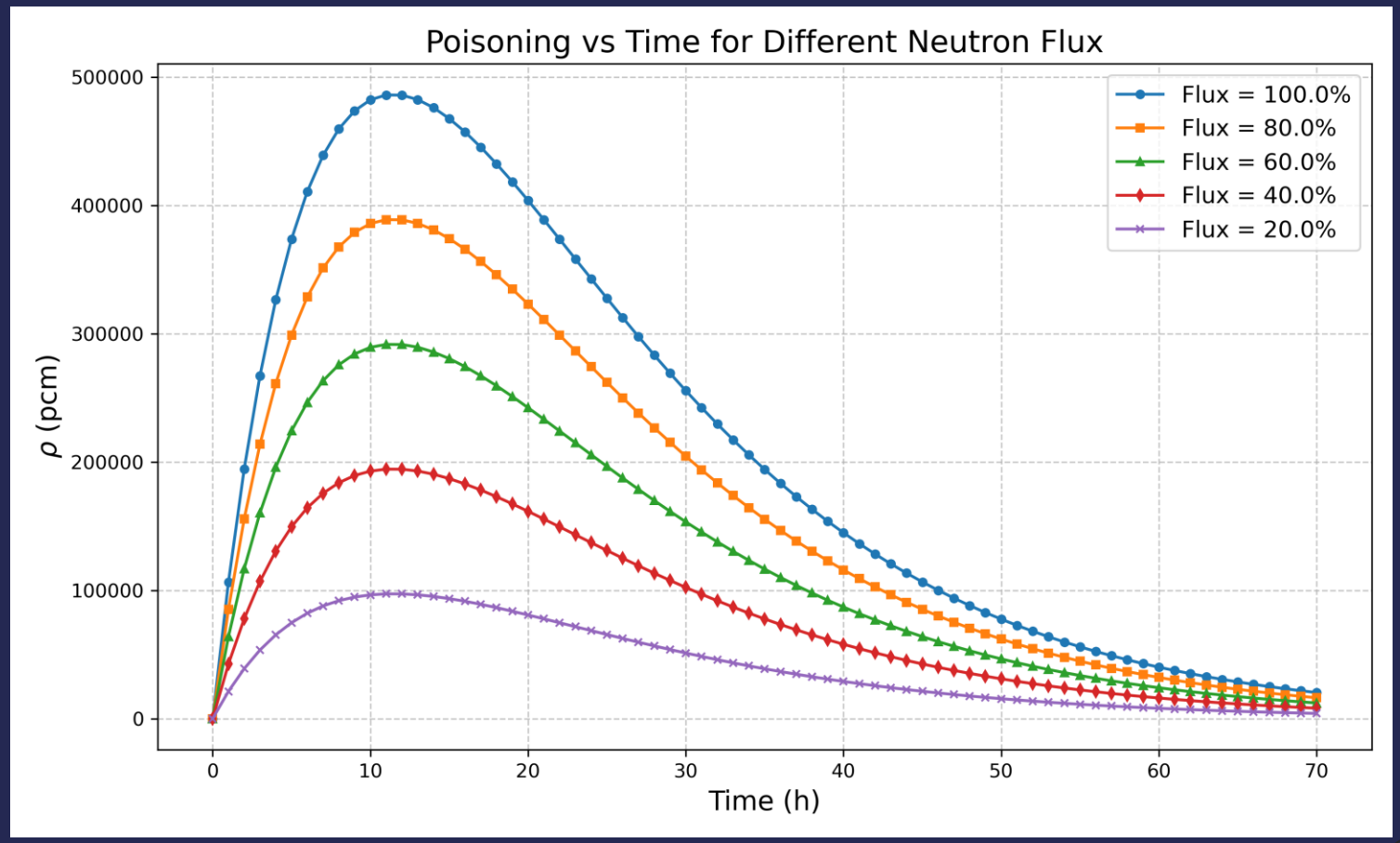
Time step of 3600 s, up to 252000 s (= 70 h)



Matrix exponential

Numerical solutions: results

Time step of 3600 s, up to 252000 s (= 70 h)

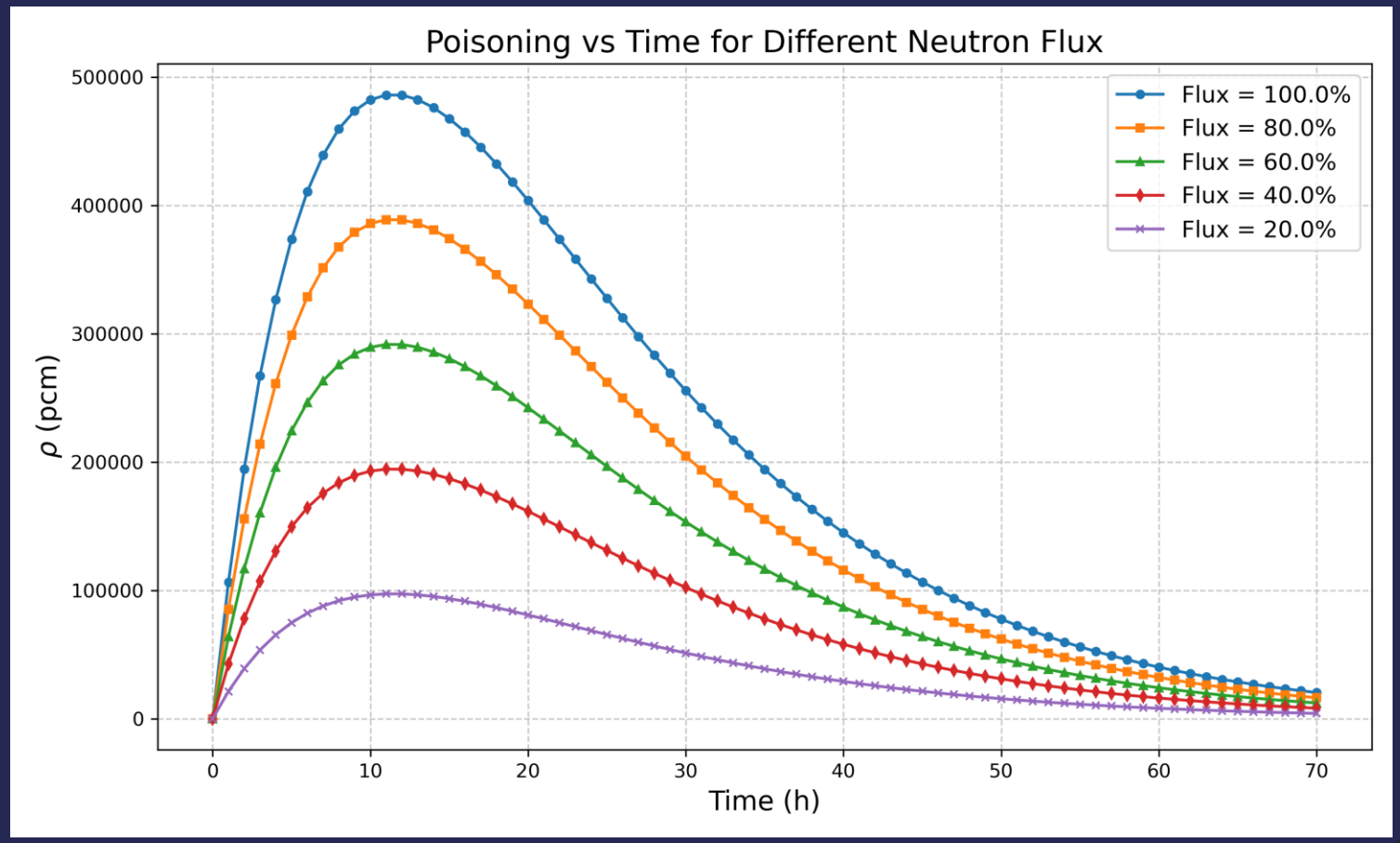


Poisoning: Fourth-order Runge-Kutta

Neutron flux = $4.42 \times 10^{20}\ \mathrm{cm}^{-2}\mathrm{s}^{-1}$

Numerical solutions: results

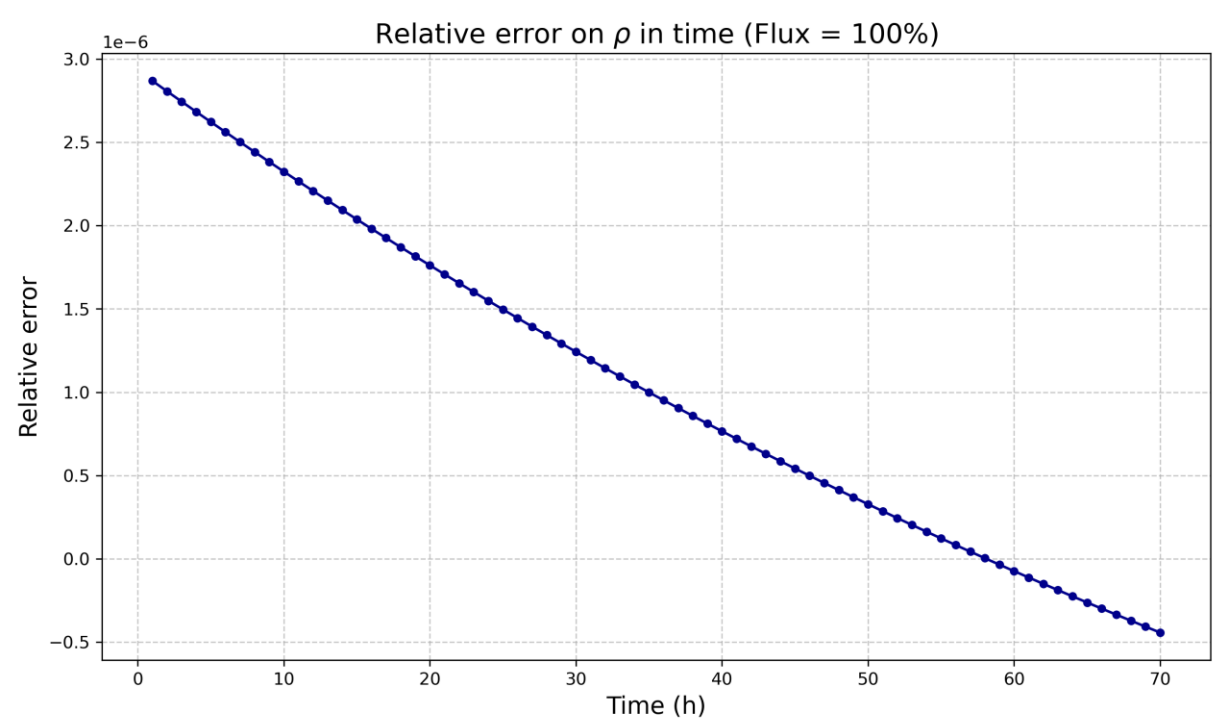
Time step of 3600 s, up to 252000 s (= 70 h)

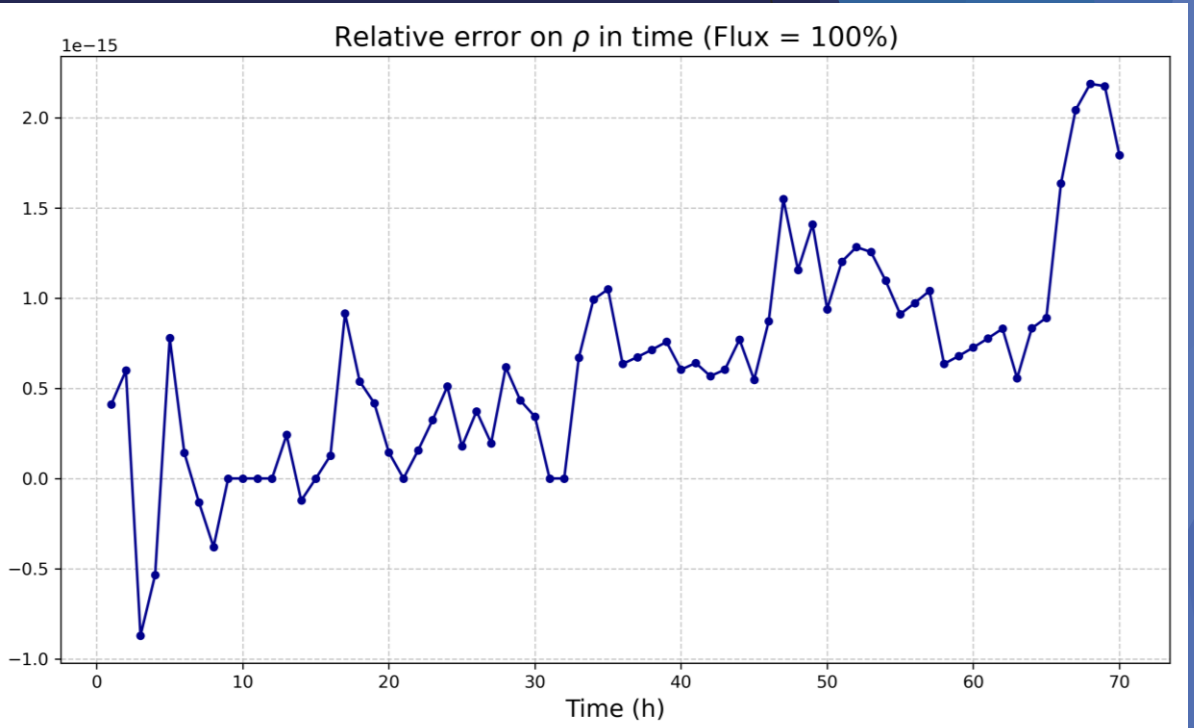


Poisoning: Matrix exponential

Neutron flux = $4.42 \times 10^{20}\ \mathrm{cm}^{-2}\mathrm{s}^{-1}$

Numerical solution:
Relative error on poisoning





Fourth order Runge-Kutta $\sim 10^{-4}\%$

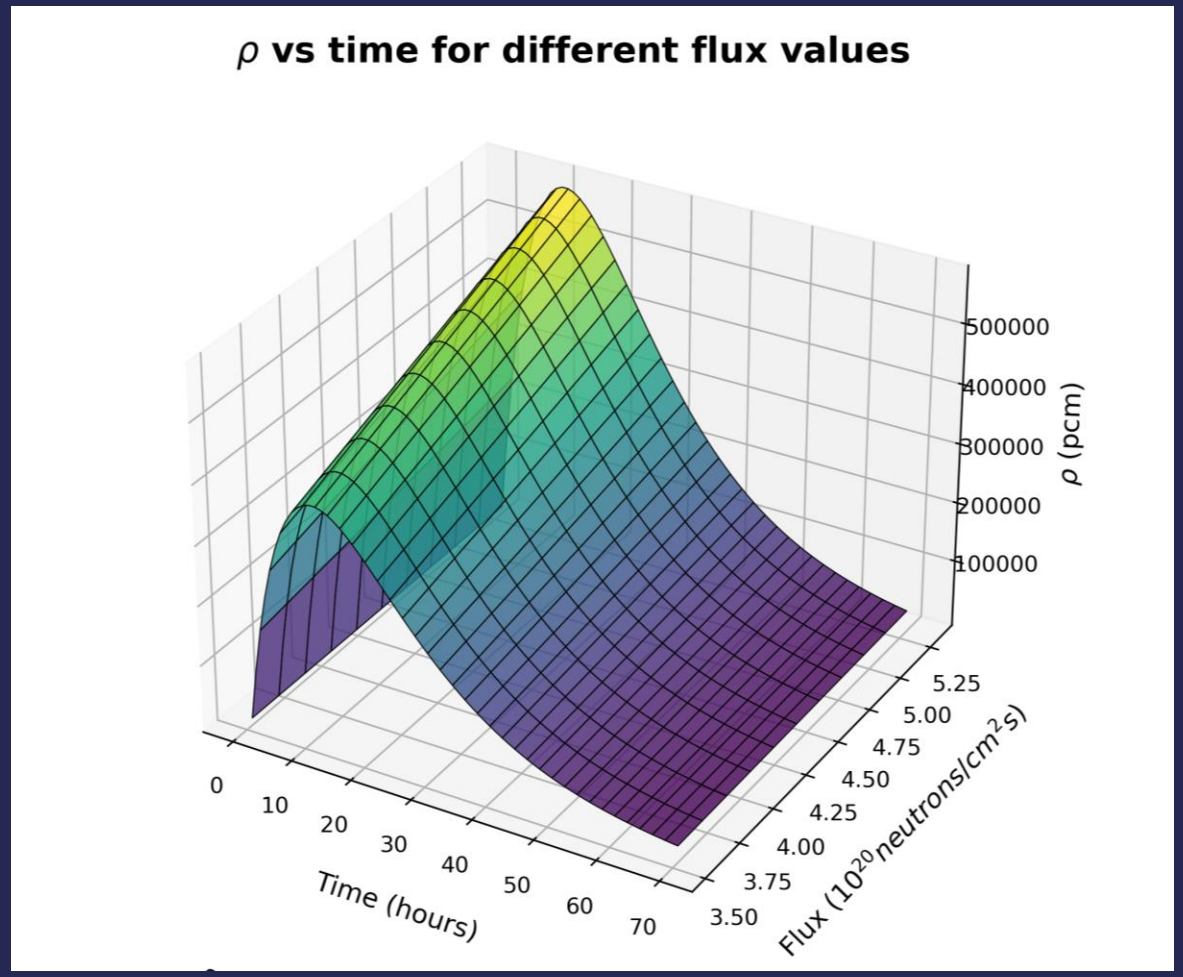
Matrix exponential $\sim 10^{-13}\%$

Numerical solution: Time performance

```python
import time

def measure_execution_time(method, *args):
    """
    Measures the average execution time of a given method over multiple runs.

    Args:
        method: The method to be tested
        *args: The arguments to pass to the method.

    Returns:
        The average execution time of the method in milliseconds.
    """
    times = []

    for _ in range(300):  # Run the method 300 times
        start_time = time.perf_counter()  # Record the start time
        method(*args)  # Execute the method
        end_time = time.perf_counter()  # Record the end time
        times.append(end_time - start_time)

    average_time = sum(times) / len(times)
    return average_time * 1000  # time in ms
```

- Mean value over 300 execution times: $\sim 10^{-1}$ ms for both methods
- Slightly better for matrix method

Xenon transient surface



- Matrix exponential method used
- Xenon transient surface used to predict Xe population

Code structure

- **parameters.py**: Defines the parameters used throughout the calculations

- **bateman_eq.py**: Defines the Bateman equations, the function to compute initial conditions, and the function to compute reactor poisoning ρ. It contains also the analytical solution of the Bateman equations for Xe and computes the relative error of poisoning of the numerical solution with respect to the analytical one.

- **runge_kutta.py**: Implements the fourth-order Runge-Kutta method for solving differential equations.

- **matrix_method.py**: Implements the matrix exponential method for solving differential equations.

Code structure

- **plot_results.py**: Contains functions for plotting results, including Iodine and Xenon populations, reactor poisoning, relative errors, and a 3D plot of the xenon transient surface.

- **compute_solutions.py**: Compute solutions of the Bateman equations using both matrix method and Runge-Kutta method, for different neutron flux values.

- **estimate_ex_time.py**: Contains the function that measures the execution time of a method.

- **main.py**: Main function that calls all the others

Thank you!