Neutron Diffusion

Elisa Medda

University of Bologna

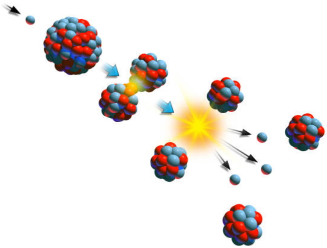
27/07/2022

Historical Introduction

In order to respond to the possible attack of Germany, in 1942, J. Robert Oppenheimer became the director of the Los Alamos National Laboratory, selected as the top-secret location for bomb design, collecting together some of the world's most famous scientists.

The Physics of a nuclear weapon

Now, we will focus on the diffusion of neutrons in a fissile material, where collisions between free neutrons and nuclei result in the release of secondary neutrons leading to a possible chain reaction.



As the fissile material increases in size, the radioactive material will become critical when the total density of neutrons increases exponentially.

The result is a runaway nuclear reaction that can lead to an intense explosion.

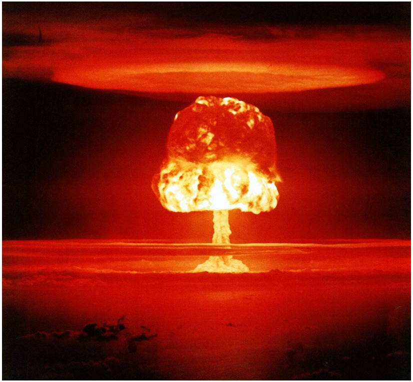


Figura 1: Photograph of the Trinity explosion, taken by Jack W. Aeby, 16th July, 1945.

Nuclear reaction

Suppose to take a domain Ω with boundary $\partial\Omega$, then we can write the diffusion equation, with a source, as:

$$\frac{\partial n}{\partial t} = \mu \nabla^2 n + \eta n \tag{1}$$

where

$$n = n(t, x), \qquad \mu, \eta > 0, \qquad 0 \leq t < \infty, \qquad x \in \mathbb{R}^s \tag{2}$$

and $s = 1, 2, 3$.
Where n is the neutron density, $\mu(m^2/s)$ the diffusion constant and $\eta(s^{-1})$ the neutron rate of formation.

1D case

Now we base our calculation upon simplified **Dirichlet boundary conditions**; thus, the neutron density is assumed to fall to zero at the edges of the core (i.e. no neutron escape).
Furthermore, we have:

$$n(t,0) = 0 \qquad\qquad n(t,L) = 0 \tag{3}$$

Now, for the one dimensional case, we have:

$$\frac{\partial n}{\partial t} = \mu \frac{\partial^2 n}{\partial x^2} + \eta n \tag{4}$$

Moreover, we can postulate a solution of the form:

$$n(t,x) = T(t)X(x) \tag{5}$$

Thus, we are able to obtain:

$$\frac{\partial T}{\partial t}X = \mu\frac{\partial^2 XT}{\partial x^2} + \eta TX \implies \frac{1}{T}\frac{\partial T}{\partial t} - \eta = \frac{\partial^2 X}{\partial x^2}\frac{1}{X}\mu = -\alpha \quad (6)$$

where α is the separation constant.
Now we have two ODEs:

$$\frac{dT}{dt} = (\eta - \alpha)T \qquad \frac{d^2 X}{dx^2} = -\frac{\alpha}{\mu}X \quad (7)$$

and the solutions are:

$$T = Ae^{(\eta-\alpha)t}, \qquad X = B_1\cos\left(\sqrt{\alpha/\mu}x\right) + B_2\sin\left(\sqrt{\alpha/\mu}x\right) \quad (8)$$

and

$$n = e^{(\eta-\alpha)t}\left[D\sin\left(\sqrt{\alpha/\mu}x\right)\right] \quad (9)$$

Since from the boundary conditions $n = 0$ at $x = 0$.

Furthermore, imposing $n = 0$ at $x = L$ we have:

$$\alpha = \mu\left(\frac{p\pi}{L}\right)^2 \tag{10}$$

where $p = 1, 2....$
Thus, from the superposition principle we can write:

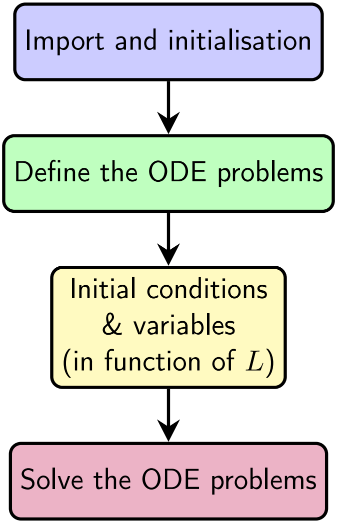
$$n = \sum_{p=1}^{\infty} a_p e^{(\eta-\alpha_p)t} \sin\left(\frac{p\pi}{L}x\right) \tag{11}$$

where the constants a_p are defined by the initial and/or the boundary conditions. Now, the critical condition for n, to increase unbounded, is that:

$$L > p\pi\sqrt{\mu/\eta} \tag{12}$$

and the worst scenario is when $p = 1$, i.e $L_{crit} = \pi\sqrt{\frac{\mu}{\eta}}$

1D, I Method

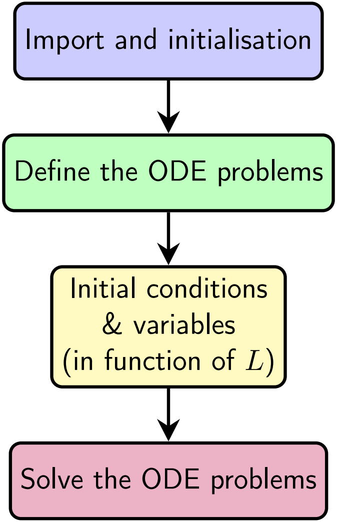


1. Import necessary libraries (for Julia):
    - *Plots*
    - *DifferentialEquations*: ODEProblem, solve
    - *ForwardDiff*: derivative
    - *DiffEqOperators*: CenteredDifference,Dirichlet0BC, to discretize the differential operators.
    - *LinearAlgebra*: eigen
    - *NumericalIntegration*: integrate
2. Initialization and definition of constants

   For ^{235}U and for ^{239}Pu

```julia
using Plots
using DifferentialEquations: ODEProblem, solve
using ForwardDiff: derivative
using DiffEqOperators: CenteredDifference,DirichletOBC
using LinearAlgebra: eigen
using NumericalIntegration: integrate




#diffusion constant
const μ_U = 2.3446e5 #m^2/s
const μ_P = 2.6786e5 #m^2/s

#neutron rate of formation
const η_U = 1.8958e8 #s^-1
const η_P = 3.0055e8 #s^-1
```

We import the necessary libraries in Julia and define the physical relevant constants.



3. We have to define the ODE problems:

In 1D we take $\Omega = [0, L]$, and we postulate a solution like:

$$n(t,x) = T(t)X(x) \tag{13}$$

And from here we have two ODEs to solve:

$$\frac{dT}{dt} = (\eta - \alpha)T \tag{14}$$

$$\frac{d^2X}{dx^2} = -\frac{\alpha}{\mu}X \tag{15}$$

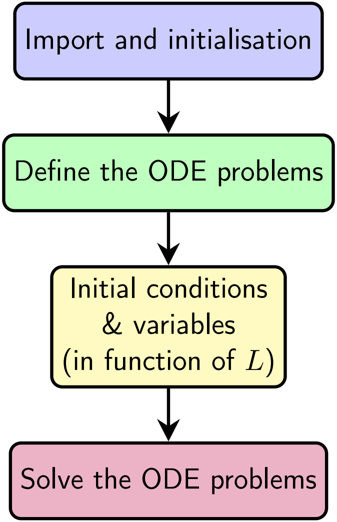
Which means that our ODEs depend on α, that depends on L

```julia
#For the T
function diffusion_t(T::Float64,
p::Vector{Float64},t::Float64)
    η,α =p
    return (η-α)*T
 end
```

We create the function for the ODE.

In the case of X we have a 2nd order ODE, while the T one is a 1st order ODE, and both of them will depend on some parameters (p).

Important is the dependence on α, thus in L, that will be our unknown in the upcoming analysis.
We didn't implement a function for the ODE of X, since we will discretize it and solve for the eigenvalues.



4. Define the initial conditions:
From the solution of T:

$$T = Ae^{(\eta-\alpha)t} \tag{16}$$

We can set:

$$T_0 = T(0) = 1$$

```julia
Δx(L,nx)= L/(nx+1) #discretization step
const ord_deriv = 2; #order of derivative
const ord_approx = 2; #order of approximation
# -μΔX = aX, where Δ is the discretization of the differential operator:
Δ(L::Float64,nx::Int64,μ::Float64)= -μ*
CenteredDifference(ord_deriv,ord_approx,Δx(L,nx),nx)*
DirichletOBC(Float64)
```

In this part of the code starting from:

$$-\mu \frac{\mathrm{d}^2 X}{\mathrm{d}x^2} = \alpha X \tag{17}$$

we discretize the following objects:

$$X(x) \longrightarrow X_i \quad i = 1, \ldots, n_X, \qquad -\mu \frac{\mathrm{d}^2}{\mathrm{d}x^2} \longrightarrow \Delta$$

$$\text{s.t.} \quad \sum_{j=1}^{n_X} \Delta_{ij} X_j = \alpha X_i \tag{18}$$

```julia
#Function to calculate eigenvalues and eigenvectors
function α_eigen(L::Float64,nx::Int64,μ::Float64)
    #=Matrix of Δ, where we focus on the interior points, since the BC set to zero the extremal points =#
    Δ_matrix = reduce(hcat,Array(Δ(L,nx,μ)))[:,1:nx]
    return eigen(Δ_matrix)
end
α_eigenvalues(L::Float64,nx::Int64,μ::Float64)= α_eigen(L, nx, μ).values
α_eigenvectors(L::Float64,nx::Int64,μ::Float64)= α_eigen(L, nx, μ).vectors
```

Here we define a function in order to be able to calculate the eigenvalues and the eigenvectors of the discretize diff. operator.

```
#q-th eigenvalue
α(L::Float64,nx::Int64,μ::Float64,q::Int64)=
↪  α_eigenvalues(L, nx, μ)[q]
#discretization of the X domain
L_range(L::Float64,nx::Int64) =
↪  Δx(L,nx):Δx(L,nx):L-Δx(L,nx)
#conditions for T
T0=1. #initial condition
p_T(L::Float64,nx::Int64,μ::Float64,η::Float64,q::Int64)=
↪  [η, α(L, nx, μ, q)] #parameters
t_step = 1e-9 #step of t range
t_span = (0.0, 1e-7) #boundary of the time domain
#discretized domain:
t_range= t_span[1]:t_step:t_span[2]
```

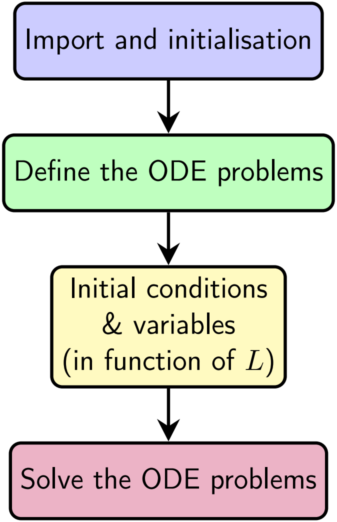
Where we choose a time span of 10^{-7}, since $\eta \sim \mathcal{O}\left(10^8 s^{-1}\right)$. Thus, in this way T is big enough to see clearly the evolution graphically.

```julia
#function for the ODE of T
prob_T(L::Float64,nx::Int64,μ::Float64,η::Float64,q::Int64) = ODEProblem(diffusion_t,T0,t_span,p_T(L, nx, μ,η, q))
sol_T(L::Float64,nx::Int64,μ::Float64,η::Float64,q::Int64)= solve(prob_T(L, nx, μ,η, q))

#q-th eigenvector solution for X
sol_X_eigen(L::Float64,nx::Int64,μ::Float64,q::Int64)= α_eigenvectors(L, nx, μ)[:,q]
```

Here we use *ODEProblem* function to set up the problem, which takes as arguments:

- The function for the ODE
- The initial conditions
- The interval of integration
- The parameters

Then we can call the *solve* function which give us the solution. Here we didn't choose any particular solving algorithm.



5. Now we solve the ODE problems.
   We choose different values of L, starting from zero (not included) to study the behaviour of the solutions of the ODEs.

1st approach

```julia
function find_L_critical(μ::Float64,η::Float64, ε::Float64,
    nx::Int64,L_in::Float64,ΔL::Float64,q::Int64, t_range::StepRangeLen)
    L_loop,L_crit=L_in,0 #initialization
    nt=length(t_range)#n.of time points
    weight=Array(range(0.9,1.1,nt)) #weight of the points
    #ask derivative>0, iterating until the required accuracy
    while ΔL >= ε
        L_crit=L_loop+ΔL #increasing L
        sol_T_loop=sol_T(L_crit,nx,μ,η,q) #solve ODE
        #differentiate the solution
        partial_sol_T(t)=derivative(sol_T_loop, t)
        derivative_check=partial_sol_T.(t_range)
        #weighted mean
        if (derivative_check'*weight)/sum(weight)>0
            ΔL=ΔL/10  #L (over-)critical: finer step
        else
            L_loop=L_crit #L sub-critical: new starting L
        end
    end
    return L_crit
end
```

In this part of the code we create a function to be able, once we put the parameters that we want, to find the value of the critical L.

```
function find_L_critical(μ::Float64,η::Float64,ε::Float64,nx::Int64,L_in::Float64,ΔL::Float64,q::Int64, t_range::StepRangeLen)
    L_loop,L_crit=L_in,0. #initialization
    nt=length(t_range)#n.of time points
    weight=Array(range(0.9,1.1,nt)) #weight of the points
```

We create the weight to assign at each point of the derivative, since our purpose is to find the critical L when the derivative of the solution of T is positive, and we want to take in to account any possible numerical fluctuations.

```
#ask derivative>0, iterating until the required
↪ accuracy
 while ΔL >= ε
    L_crit=L_loop+ΔL #we increase L
    sol_T_loop=sol_T(L_crit,nx,μ,η,q) #solve ODE
    #differentiate the solution
    partial_sol_T(t)=derivative(sol_T_loop, t)
    derivative_check=partial_sol_T.(t_range)
```

Here first we calculate the partial derivative of the solution of the ODE of T, then, we evaluate the derivative at a given t range.

```
#weighted mean
        if (derivative_check'*weight)/sum(weight)>0
            ΔL=ΔL/10  #L (over-)critical: finer step
        else
            L_loop=L_crit #L sub-critical: new starting
             ↪  L
        end
    end
    return L_crit
end
```

Now, we make a weighted mean and we ask when this is positive, since we know that in this scenario we have a runway nuclear reaction, and so an explosion.

Now we proceed with the following parameters to find L:

```
#parameters to find the L
ε= 1e-4; #accuracy
L_in=0.; #starting L
ΔL = 1e-2; #starting step (cm)
q_choose =1; #worst case eigenvalue
nx= 100; #n. of points for the discretization
```

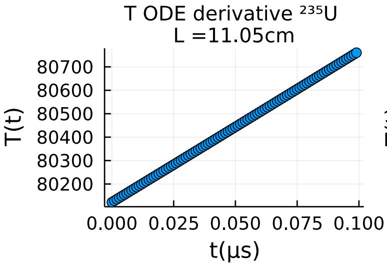
In this part of the code we choose an accuracy $\epsilon = 10^{-4}$, since the result that we expect is of $\mathcal{O}\left(10^{-2}m\right)$.Thus, we initialize L_{in} and L_{loop} to store the value of the L.
Then, for the study of L, we choose to start from $1\,cm$, from dimensional analysis: $L \sim \sqrt{\mu/\eta} \sim \mathcal{O}(3cm)$ for ${}^{235}U$ and ${}^{239}Pu$.

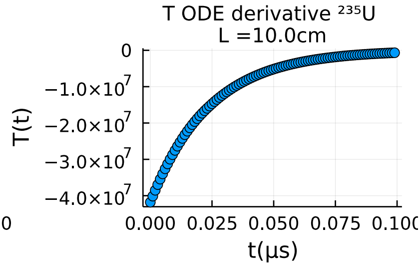
```
#calculation of the critical L for U-235 and Pu-239
L_crit_U = find_L_critical(μ_U, η_U, ε, nx, L_in, ΔL,
↪  q_choose, t_range) #m
L_crit_P = find_L_critical(μ_P, η_P, ε, nx, L_in, ΔL,
↪  q_choose, t_range) #m
```

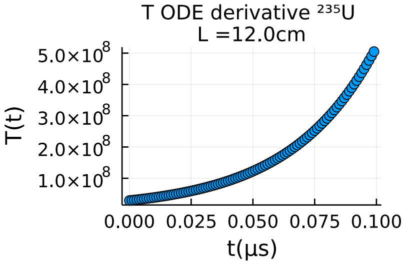
And from this two lines we find $L_{crit,U} = 11.05\,cm$ for ^{235}U, and $L_{crit,P} = 9.38\,cm$ for ^{239}Pu as we expected from the analysis of Graham W Griffiths (see [1]).

[1] https://www.researchgate.net/publication/323035158_Neutron_diffusion

In the following plots we can see that we have to *weight* more the final points, since they tell us if our derivative is truly increasing or not at late time.







```
#the function of the eigenvectors at a given time
function time_eigenvectors(L::Float64, nx::Int64,
    μ::Float64,η::Float64, t::Float64)
    X= α_eigenvectors(L,nx,μ)#full set eigenvectors
    for q in [1,nx]
        # n_q(x,t) = X_q(x)*T_q(t)
        X[:,q]=X[:,q]*sol_T(L,nx,μ,η,q)(t)
    end
    return X
end
```

Each eigenvector evolves in time accordingly with:

$$n_q(x,t) = X_q(x)\,T_q(t) \tag{19}$$

```julia
#we normalize the eigenvectors
normalization(x::Vector{Float64}, F::Vector{Float64})= 1/sqrt(integrate(x, F.*F))
normalization(x::StepRangeLen, F::Vector{Float64})= normalization(Array(x), F)
#function to calculate the coefficient of the expansion
function series_coef(f::Function, L::Float64,nx::Int64, μ::Float64)
    X = α_eigenvectors(L,nx,μ)
    x = range(0.,L,nx)
    F = f.(x) #discretize the initial function
    a_vector =zeros(Float64,nx) #initialization
    for q in 1:nx
        X_q= X[:,q]
        #coeff. calculation
        a_vector[q]=normalization(x,X_q)^2*integrate(x,F.*X_q)
    end
    return a_vector
end
```

In this part of the code we normalize the coefficients and create the function coefficients in order to solve the series:

$$a_p = (N_q)^2 \int_0^L \mathrm{d}x \, X_q(x) \, f(x) \tag{20}$$

where N_q is the normalization.
In fact, the normalization is defined as:

$$(N_q)^2 = \left(\int \mathrm{d}x \, (X_q)^2 \right)^{-1} \tag{21}$$

```
#creation of the function for the series expansion
function series_exp(f::Function,
    L::Float64,nx::Int64, μ::Float64, η::Float64,
     ↪  t::Float64)
    #n =  a_q*n_q(x,t)
    return time_eigenvectors(L,nx,μ,η,t)*
    series_coef(f,L,nx,μ)
end
```

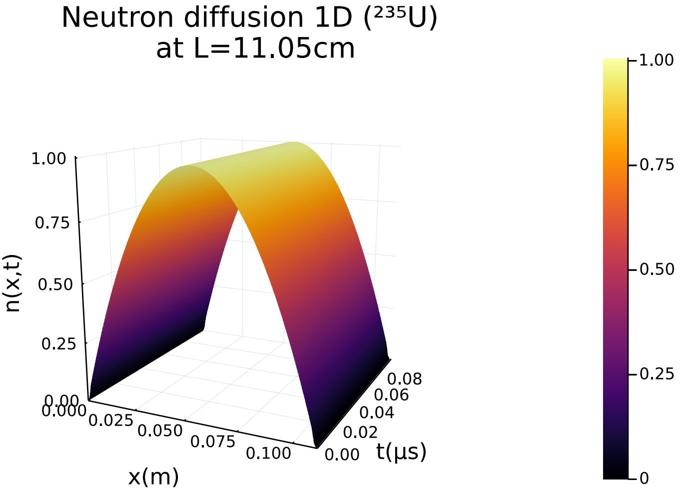
Here we write the expansion of a function n:

$$n(x,t) = \sum_{q=1}^{\infty} a_p \, n_q(x,t) \tag{22}$$

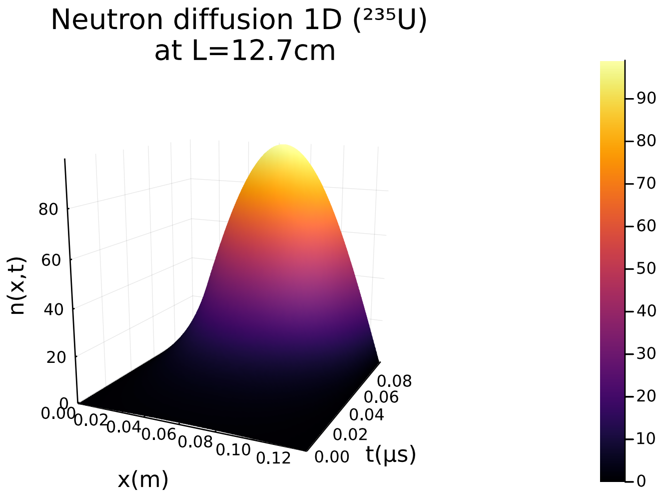
with $n(x,0) = f(x)$.

1st approach

```
#initial function that respects the BC
f_initial(x::Float64) = sin(x*π/L_crit_U);

n(t::Float64)= series_exp(f_initial,L_crit_U,nx,μ_U,η_U,t);

t_range_plot= 0.0:1e-9:1e-7 #time range for the plot

n_t = length(t_range_plot) #n. time points

nMatrix=reduce(hcat, n.(t_range_plot))';# matrix of n
BC1 = zeros(Float64,n_t) #putting back boundary
newMatrix = hcat(BC1,nMatrix,BC1)

newL_range = [0.0;L_range(L_crit_U,nx);L_crit_U]

#plot of the diffusion at exactly the critical L
Plot=plot(newL_range, t_range_plot*10^6, newMatrix, st=:surface, xlabel="x(m)", ylabel="t(μs)", zlabel="n(x,t)",title="Neutron diffusion 1D (²³U)\n at
L=$(L_crit_U*10^2)cm",camera=(25,14),dpi=1000)
```

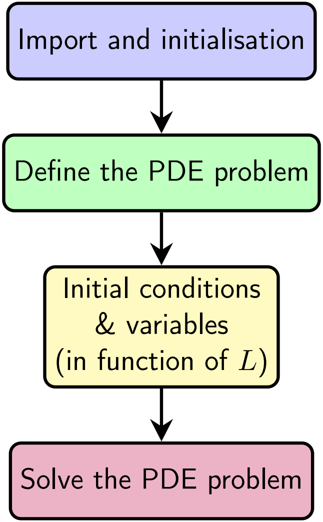


The same plot is obtained for the ^{239}Pu with a different L.



Here we can see the diffusion in the case of a sovra critical L.

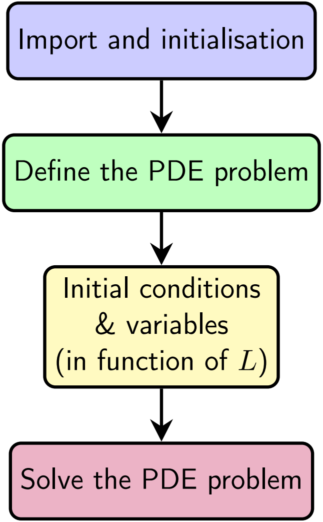
1D, II Method



Now we will present a different method, with some minor changes, in respect to the first one.

1. "Import and initialisation":

```
using Plots
using DifferentialEquations: ODEProblem, solve
using ForwardDiff: derivative
using DiffEqOperators: CenteredDifference,
DirichletOBC,DerivativeOperator,
RobinBC
using Statistics: mean
```



❷ PDE problem:

For this point we have to solve this equation:

$$\frac{\partial n}{\partial t} = \mu \frac{\partial^2 n}{\partial x^2} + \eta n \qquad (23)$$

In order to do so, we discretize:

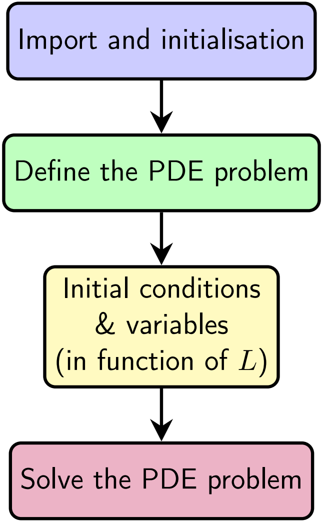
$$n(x,t) \longrightarrow n_i(t) \quad i = 1, \ldots, n_X,$$
$$\frac{\mathrm{d}^2}{\mathrm{d}x^2} \longrightarrow \Delta_{ij}$$

s.t. we have a set of ODEs:

$$\frac{\mathrm{d}n_i(t)}{\mathrm{d}t} = \mu \sum_{j=1}^{n_X} \Delta_{ij} n_j(t) + \eta n_i(t) \quad (24)$$

```julia
#definition of the PDE
function diffusionPDE(u::Vector{Float64},p,
    t::Float64)
    μ, η, Δ, bc=p  #parameters
    μ::Float64,η::Float64, bc::RobinBC, Δ::DerivativeOperator
    #Δ is the matrix, bc are the boundary condition
    return μ*Δ*bc*u+η*u
end
```

Where bc are the Boundary Conditions (BC)



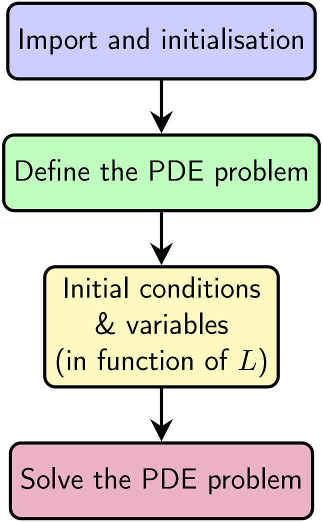
3. Define the initial conditions:
Here we will follow the same steps of the first method.

```
nx_PDE = 100; #n. of points
bc = Dirichlet0BC(Float64);#boundary conditions
#differential operator
Δ(L::Float64)= CenteredDifference(ord_deriv,
ord_approx,Δx(L),nx_PDE);
```

Here we define the same quantities, in function of the length of the domain, as we did in the first method.
The only difference is the definition of the differential operator, Δ, that now is without the numerical factor in front and the boundary conditions, that are defined separately as bc.
Moreover, notice that now we fixed the number of points of the discretization.

```
#PDE
prob_PDE(n0::Vector{Float64},L::Float64,μ::Float64,
η::Float64,bc::RobinBC)=
ODEProblem(diffusionPDE,n0,t_span,[μ, η, Δ(L), bc])

sol_PDE(n0::Vector{Float64},L::Float64,μ::Float64,
η::Float64,bc::RobinBC
) =  solve(prob_PDE(n0,L,μ,η,bc))
```

In this part of the code we define the function to instantiate the problem and the one that will solve it.



4. Now we solve the PDE problem.

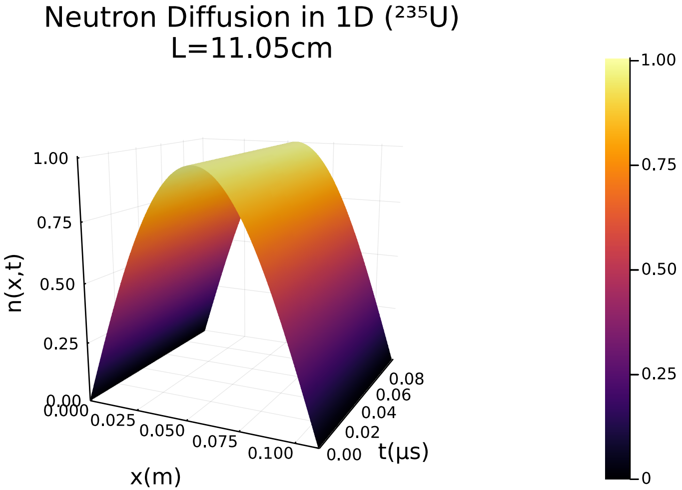
   We choose different values of L, starting from zero (not included) to study the behaviour of the solutions of the PDE.

```
function find_L_crit(μ::Float64,η::Float64,ε::Float64,L_in::Float64,
    ΔL::Float64, t_range::StepRangeLen)
    L_loop, L_crit = L_in,0. #initialization
    nt=length(t_range)#n. of time points
    weight = Array(range(0.9,1.1, nt)) #weight of the points
    points =[30, 50, 70];#points for the mean
    while ΔL >= ε
        L_crit = L_loop+ΔL #all the different L
        L_range_loop = L_range(L_crit)
        n0_loop= f.(L_range_loop, L_crit) #PDE
        sol_PDE_loop = sol_PDE(n0_loop,L_crit,μ, η,bc)
        sol_PDE_x(t)= mean(sol_PDE_loop(t)[points])
        partial_sol_x(t) = derivative(sol_PDE_x,t)
        derivative_check = partial_sol_x.(t_range)
        if (derivative_check'*weight)/sum( weight)>0
            ΔL = ΔL/10
        else
            L_loop= L_crit
        end
    end
    return L_crit
```

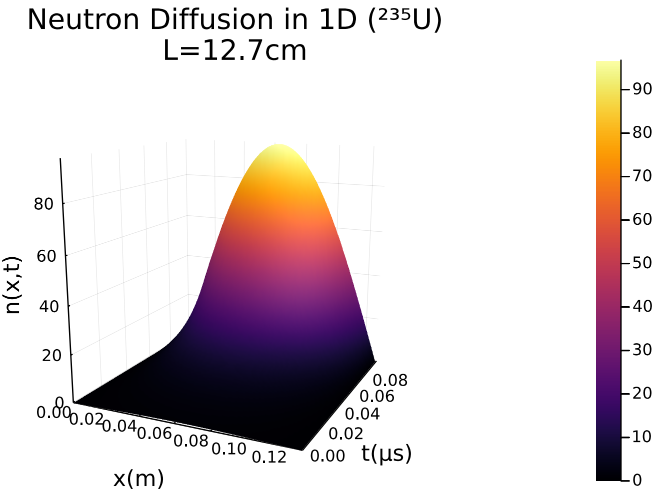
Here, as in the first method, we ask when the derivative is positive to find the correct L. From here we find $L_{critU} = 11.05\,cm$ for ^{235}U, and $L_{critP} = 9.38\,cm$ for ^{239}Pu.

Then, as in the previous case we make a plot.

```
g(x)= sin(x*π/L_crit_U);#initial function
t_range_plot = 0:t_step:1e-7;#range plot
n_t=length(t_range_plot)
L_plot =L_range(L_crit_U)#L range at L critical
n0_plot_U = g_U.(L_plot_U);
sol_plot =sol_PDE(n0_plot_U,L_crit_U,μ_U,η_U,bc).(t_range_plot)
nMatrix=reduce(hcat,sol_plot)' #matrix solution
#we need the boundary
BC1 = zeros(Float64,n_t)
newMatrix = hcat(BC1,nMatrix,BC1)
newL_range = [0.0;L_plot;L_crit_U]
Plot=plot(newL_range, t_range*10^6, newMatrix, st=:surface,
    xlabel="x(m)", ylabel="t(μs)", zlabel="n(x,t)",
    title ="Neutron Diffusion in 1D
    ↪  (U²³)\nL=$(L_crit_U*10^2)cm",camera=(25,14),dpi=1000)
```



The same plot is obtained for the ^{239}Pu with a different L.



Here we can see the diffusion in the case of a sovra critical L.

3D in Cartesian

Now we will discuss the three dimensional case. Our domain is $\Omega = [0, L_x] \times [0, L_y] \times [0, L_z]$, and the problem is:

$$\frac{\partial n}{\partial t} = \mu\left(\frac{\partial^2 n}{\partial x^2} + \frac{\partial^2 n}{\partial y^2} + \frac{\partial^2 n}{\partial z^2}\right) + \eta n \tag{25}$$

we can postulate a solution of the form:

$$n(t, x, y, z) = T(t)X(x)Y(y)Z(z). \tag{26}$$

In this way we have to solve four ODEs:

$$\frac{dT}{dt} = (\eta - \alpha)T, \qquad \frac{d^2X}{dx^2} = -\frac{\alpha_x}{\mu}X,$$
$$\frac{d^2Y}{dy^2} = -\frac{\alpha_y}{\mu}Y, \qquad \frac{d^2Z}{dz^2} = -\frac{\alpha_z}{\mu}Z$$

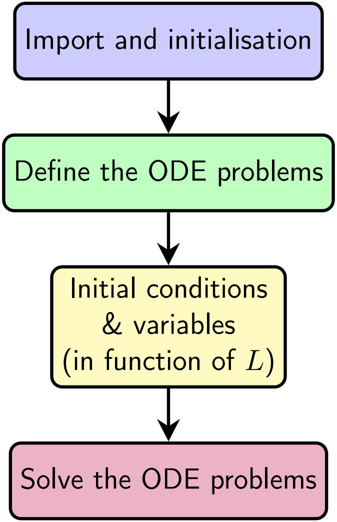
Where we have that $\alpha = \alpha_x + \alpha_y + \alpha_z$.
Thus, we have the expansion in eigenvectors:

$$n(t,x,y,z) = \sum_{q_x,q_y,q_z}^{\infty} a_{q_x q_y q_z} n_{q_x q_y q_z}(t,x,y,z) \tag{27}$$

where the coefficients are given by:

$$a_{q_x q_y q_z} = (N_{q_x} N_{q_y} N_{q_z})^2 \int_0^{L_x} \int_0^{L_y} \int_0^{L_z} X_{q_x} Y_{q_y} Z_{q_z} f(x,y,z) \tag{28}$$

where N_{q_i} is the normalization and we have
$n(0,x,y,z) = f(x,y,z)$.



Now we generalize the first $1D$ method in 3 dimensions.

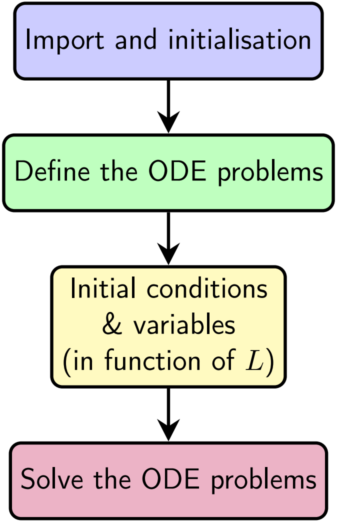
1. "Import and initialisation"

The only additional import is

```
using Einsum #Einstein summation
```

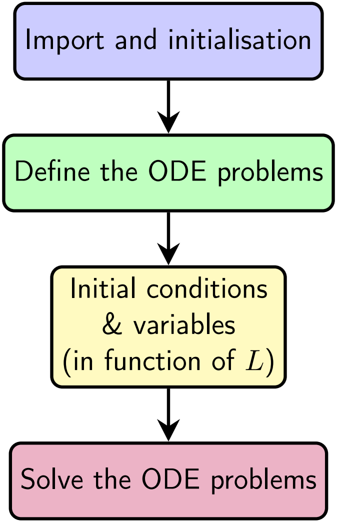
to handle multi-index sums. Additionally, we have to introduce the density of the ^{235}U and ^{239}Pu, in order to be able to calculate the critical mass.

```
#density
const ρ_U = 18.71e3 #kg/m^3
const ρ_P = 15.60e3 #kg/m^3
```



❷ ODE problems:

The definition of the quantities of this section is completely equal to the $1D$ analogue. Thus, we will omit it.



3 Define the initial conditions.

Now, differently from the $1D$ case, we compute the sum of the q_x-th, q_y-th, q_z-th eigenvalues, of their respective discretized differential operator

```
function α_sum(L::Vector{Float64},
    N::Vector{Int64},
    μ::Float64,
    Q::Vector{Int64})
    Lx, Ly, Lz = L
    nx, ny, nz = N
    qx, qy, qz = Q
  return α(Lx,nx,μ,qx)
  +α(Ly,ny,μ,qy)+α(Lz,nz,μ,qz)
end
```

Then we have that now the parameters are a function of L a vector equal to (L_x, L_y, L_z) and $Q = (q_x, q_y, q_z)$ index of the eigenvalues/eigenvectors.

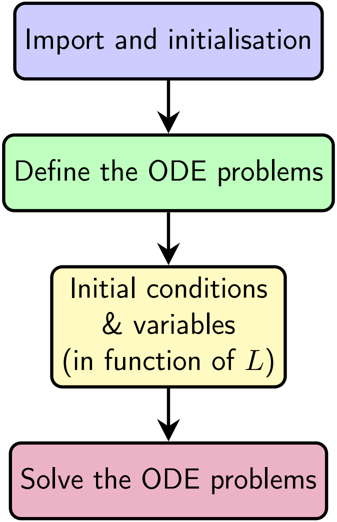
```
p_T(L::Vector{Float64},
    N::Vector{Int64},
    μ::Float64,
    η::Float64,
    Q::Vector{Int64}) = [η, α_sum(L, N, μ, Q)];
```

and this function returns the parameters of the time ODE:

$$\frac{dT}{dt} = (\eta - \alpha)t \tag{29}$$

Then we can write also the function to instantiate the problem and the one that will solve it, in function of L and Q.

```
#function for the ODE of T
prob_T(L::Vector{Float64},
    N::Vector{Int64},
    μ::Float64,
    η::Float64,
    Q::Vector{Int64})= ODEProblem(diffusion_t, T0, t_span, p_T(L, N, μ, η, Q))
sol_T(L::Vector{Float64},
    N::Vector{Int64},
    μ::Float64,
    η::Float64,
    Q::Vector{Int64})= solve(prob_T(L, N, μ, η, Q))
#q-th eigenvector solution for X
sol_X_eigen(L::Float64,
    nx::Int64,
    μ::Float64,
    q::Int64)= α_eigenvectors(L, nx, μ)[:,q]
```



4. Now we solve the ODE problems.
In order to solve the ODEs we make the same study that we made in the one dimensional case, fixing $L_x = L_y = L_z$, i.e. a cubic domain.

Now, introducing an analogous function to find the critical L, as we did in the $1D$ case, we find:

$$L_{crit,U} = 19.14\,cm \qquad L_{crit,P} = 16.25\,cm$$

confirming the result of Graham W Griffiths (see [1]).
Knowing the critical L is now possible to calculate the critical mass, as $m_{crit} = \rho\,(L_{crit})^3$:

$$m_{crit_U} = 131.18\,kg \qquad m_{crit_P} = 66.94\,kg$$

for ^{235}U and ^{239}Pu respectively.

[1] https://www.researchgate.net/publication/323035158_Neutron_diffusion

In order to find the L we had to define the Q vector of eigenvalues, set to $(1,1,1)$, i.e. the worst scenario.
Thus, we again define a function to calculate the series expansion of n, where now we have an additional argument Q_{max}, representing a cutoff:

$$n = \sum_{q_x, q_y, q_z} a_{q_x q_y q_z} n_{q_x q_y q_z}, \qquad q_x + q_y + q_z < Q_{max} \tag{30}$$

since in $3D$ we are dealing with $N \times N \times N$ tensors, and the computational time is heavily impacted by the number of operations, justifying the cutoff.

```julia
#function to calculate the series expansion
function series(f::Function,L::Vector{Float64},
N::Vector{Int64},μ::Float64,η::Float64,
t::Float64,Q_max::Int64)
    L1, L2, L3 = L #L for X,Y,Z
    nx,ny,nz = N
    X = α_eigenvectors(L1,nx,μ)
    Y = α_eigenvectors(L2,ny,μ)
    Z = α_eigenvectors(L3,nz,μ)
    x_range = range(0.,L1,nx) #the range
    y_range = range(0.,L2,ny)
    z_range = range(0.,L3,nz)
    #discretize the initial function
    @einsum F[i,j,k] := f(x_range[i], y_range[j], z_range[k])
    f_series = zeros(Float64,(nx,ny,nz)) #initialization
    for q1 in 1:nx, q2 in 1:ny, q3 in 1:nz
        if q1+q2+q3 == Q_max #cut off of the expansion
            return f_series
        end
```

```
            X_q= X[:,q1]
            Y_q= Y[:,q2]
            Z_q= Z[:,q3]
            @einsum R[i,j,k] := X_q[i]*Y_q[j]*Z_q[k]
            #coeff. calculation
            norm = normalization(x_range,X_q)*
            normalization(y_range,Y_q)*
            normalization(z_range,Z_q)
            a= norm^2*integrate((x_range,y_range,z_range),F.*R)
            f_series .+= a.*R.*sol_T(L,N,μ,η,[q1,q2,q3])(t)
        end
        return f_series
    end
```

This function computes the expansion in eigenvectors of the ODEs, with their time evolution satisfying the time ODE, of the function f(x,y,z) in a given domain.

Now, we choose the initial function and important quantities:

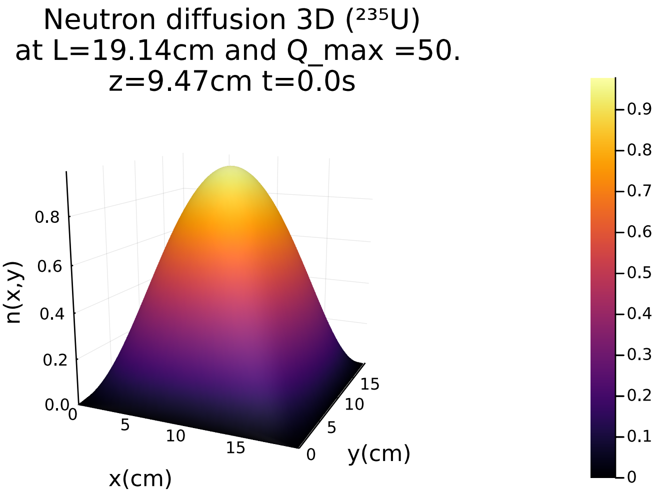
```
#initial function
f_initial(x::Float64,y::Float64,z::Float64) =
↪  sin(π*x/L_crit_U)*sin(π*y/L_crit_U)*sin(π*z/L_crit_U);
Q_max =50 #terms of the series
nx=100;#n. of points
N=[nx,nx,nx]; #n. of points for X,Y,Z
L = [L_crit_U, L_crit_U, L_crit_U] #L Vector
#calculation of n
n(t::Float64)= series(f_initial,L,N,μ_U, η_U,t,Q_max)
n_t = n(0.) #n at t=0
```

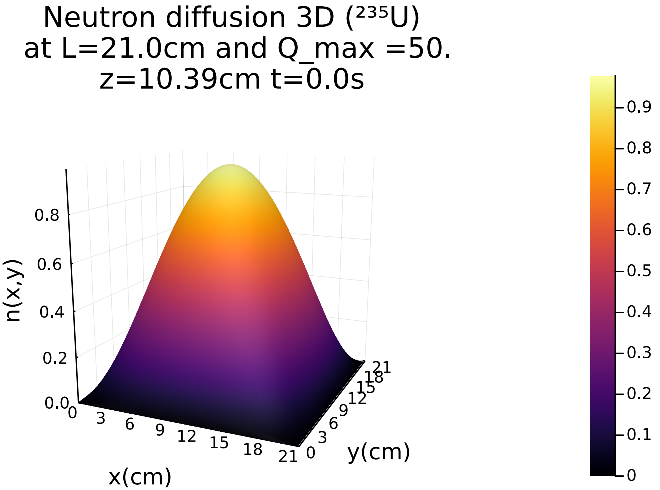
As initial function we choose the same of the $1D$ case but adapted to the $3D$:

$$f(x,y,z) = \sin\left(\pi \frac{x}{L_{crit}}\right) \sin\left(\pi \frac{y}{L_{crit}}\right) \sin\left(\pi \frac{z}{L_{crit}}\right) \tag{31}$$

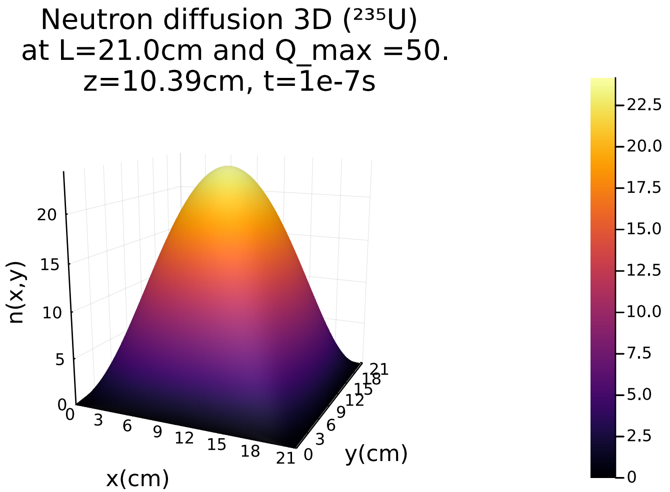
```
L_range_n=range(0.,L_crit_U,nx);
n_plot = n_t[:,:,50] #set the z-axis
BC1=zeros(Float64,nx) #we need the BC
newMatrix = hcat(BC1,n_plot,BC1)
BC2=zeros(Float64,nx+2)
newmatrix2 = vcat(BC2',newMatrix, BC2')
newL_range = [0.0;L_range_n;L_crit_U]
#plot
Plot2=plot(newL_range,newL_range,newmatrix2, st=:surface,
    xlabel="x(m)", ylabel="t(μs)", zlabel="n(x,t)",
    title="Neutron diffusion 3D (²³U)\n at L=$(L_crit_U*10^2)cm and Q_max =$Q_max",
     camera=(24,14),dpi=1000)
```

Thus, we make a plot of the neutron diffusion in $3D$


Neutron diffusion 3D (²³⁵U)
at L=19.14cm and Q_max =50.
z=9.47cm t=0.0s
n(x,y)
0.0
0.2
0.4
0.6
0.8
x(cm)
0
5
10
15
y(cm)
0
5
10
15
0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9



Here we can see the diffusion in the case of a sovra critical L.


Neutron diffusion 3D (²³⁵U)
at L=21.0cm and Q_max =50.
z=10.39cm, t=1e-7s
n(x,y)
0
5
10
15
20
x(cm)
0 3 6 9 12 15 18 21
y(cm)
0 3 6 9 12 15 18 21
0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0
22.5

Thank You for the attention!