jupytext					
cell_metadata_filter	formats	text_representation			
		extension	format_name	format_version	jupytext_version
-all	md:myst	.md	myst	0.13	1.11.5

A transient LES-VMS scheme: flow through a perforated cylinder

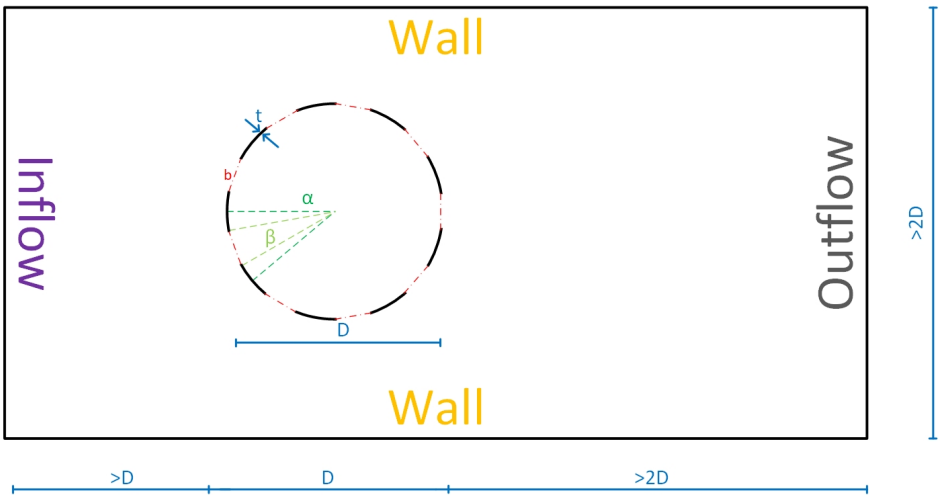
Author: Ruben Dekeyser and Oriol Colomés

Published: June 2023
Gridap version: Gridap@0.17.17

This tutorial shows the implementation of an external mesh for a perforated cylinder in 2D in Gridap.jl. The goal is to get a time trace of the drag and lift force experienced by the monopile.

A. Problem description

The goal is to model a steady uniform flow through a perforated cylinder. The general domain looks as follows:



To this end, the problem is split into a series of consecutive steps:

1. Set up the problem
    i. Define the model parameters and input values
    ii. Define the geometry
    iii. Define the boundary conditions
2. Set up the numerical scheme
    i. Set up the test spaces
    ii. Build trial spaces
    iii. Combine them in a (transient) multifield

    iv. Initialize the domain
    v. Define the weak form
    vi. Set up the time integration scheme
3. Apply the nonlinear solver and time integration
4. Do post-processing

For engineering research into perforated monopiles the post processing should be built around the following data:

1. The velocity and pressure fields (usually in vtk format). This will be useful in the early stages of the development to look into the exact flow effects, eddies, ...
2. Drag and lift time-traces, either in the time or frequency domain, for further design analysis and trend-seeking.

B. Set up of the module and surrounding parallelization options

We start by importing all required packages in a module. This module allows for easier parallel execution in cluster machines. The 2 files that are imported containt the actual Navier Stokes implementation, and the mesh generator. the latter is not elaborated on here. However, the code can still be publically accessed.

```
using Gridap
using Gridap.FESpaces: zero_free_values, interpolate!
using Gridap.Fields: meas
using GridapGmsh: gmsh, GmshDiscreteModel
using GridapDistributed
using GridapDistributed: DistributedTriangulation, DistributedCellField
using GridapPETSc
using GridapPETSc: PETSC
using PartitionedArrays
using SparseMatricesCSR

using CSV
using DataFrames

include("NavierStokesParallel.jl")
include("mesh_generation.jl")
```

The meshes can be generated by calling a dedicated funtion, taking the different diameters, wall thicknesses, perforations, porosities, angles of attack and domain lengths as inputs. This creates meshes of all their combinations. To get a single mesh, just input 1-element lists. By changing the boolean `do_mesh` value it is possible to easily get a list of all generated files, without having to actually generate them. This is useful if one wants to re-run simulations.

```
function generate_meshes(; nD=[10], nRt=[100], nperf=[12], nβ=[0.5], nα=[0], nR_L=[7], do_mesh=true)
  # Create cases
  cases = []
  # RDK values
  for D in nD
    for Rt in nRt
      for num_perforations in nperf
        for β in nβ
          for α in nα
            for R_L in nR_L
                D2 =  round(D;digits=2)
                Rt2 = round(Rt;digits=0)
```

```
                β2 =  round(β;digits=2)
                α2 =  round(α,digits=2)
                filename = "D$D2-Rt$Rt2-n$num_perforations-beta$β2-alfa$α2-RL$R_L.msh"
                push!(cases,replace(filename, r".msh"=>""))

                if do_mesh==true
                  create_mesh(filename=filename, D=D, R_t=Rt, num_perforations=num_perforations, R_β=β,
α=360/num_perforations*α,
                              R_L=R_L, R_Cx=R_L-5.5)
                end
              end
            end
          end
        end
      end
    end

    return cases

end
```

After that a series of `mumps` flags is defined for the module to work correctly:

```
options_mumps = "-snes_type newtonls \
-snes_linesearch_type basic  \
-snes_linesearch_damping 1.0 \
-snes_rtol 1.0e-8 \
-snes_atol 1.0e-10 \
-ksp_error_if_not_converged true \
-ksp_converged_reason -ksp_type preonly \
-pc_type lu \
-pc_factor_mat_solver_type mumps \
-mat_mumps_icntl_7 0"
```

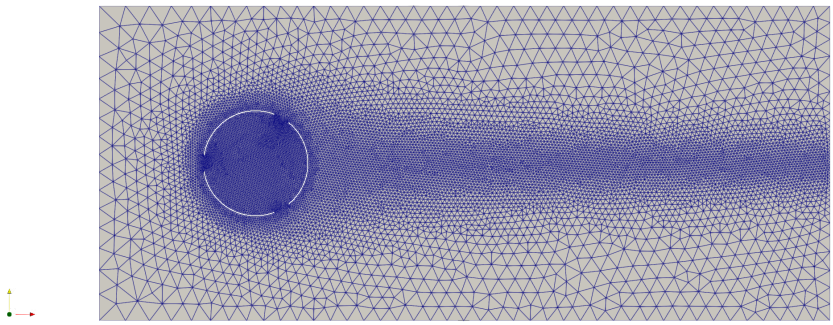
Lastly the actual parallel solver can be called. The default values of the parameters allow easy execution of a warmup loop, allowing a system image to be stored on a cluster. That way the system image can be used on all calculation cores, without having to recompile the module each time. This saves multiple hours of computation time for high parallel computing.

```
function main_parallel(np;
  mesh_file="tmp_mesh_coarse.msh",
  vtk_outpath="tmp_mesh_coarse",
  Vinf=1,
  Δt=0.1,
  tf=1.0,
  Δtout=0.5,
  output_path=joinpath(ENV["PerforatedCylinder_DATA"],"vtk"))

  filename = replace(mesh_file, r".msh"=>"")
  run_name = filename*"-Vinf$Vinf-dt$Δt"

  current_path = pwd()
  cd(output_path)
  with_backend(MPIBackend(),np) do parts
    options = options_mumps
    GridapPETSc.with(args=split(options)) do
      run_test_parallel(parts,run_name,mesh_file,vtk_outpath,Vinf,Δt,tf,Δtout)
    end
```

```
  end
  cd(current_path)
end
```

1. Setup of the problem

The model gets 3 main inputs: the mesh, the inflow velocity, and the time step. The mesh is triangulated and split into a fluid and a solid domain. Additionally the boundaries are read from predefined tags from the mesh-file. The bounds of all these sub-domains are `Measured` to be used in integration schemes later, and additionally the normal directions of the structure and the outflow boundary are calculated. The boundary conditions are: a uniform (Dirichlet) inflow of speed V_{in}, prescribed-velocity $(V_{in}, 0)$ walls and a no-slip no-penetration structure. The outflow boundary is included in the weak form directly.



For the code we define a more elaborate structure.

1.1 Define the model parameters and input values

We start by referencing the `part`, or subdomain which we are currently in. This is needed since we are using parallel computing, and this way the differenc processors can communicate with each other by sharing their outputs. Additionally, while this can be omitted, we generate files to log the progress. Also we generate a file to store the force outputs.

```
if i_am_main(parts)

    logs_path=ENV["PerforatedCylinder_LOGS"]
    log_file = run_name*"-output.log"
    full_logs_path = joinpath(logs_path,log_file)
    io = open(full_logs_path, "w")

    forces_path=ENV["PerforatedCylinder_FORCES"]
    force_file=run_name*"-forces.csv"
    full_force_path = joinpath(forces_path,force_file)
    io_force = open(full_force_path, "w")

  end

  function to_logfile(x...)
    if i_am_main(parts)
      write(io,join(x, " ")...)
      write(io,"\n")
      flush(io)
    end
  end
```

```
function to_forcefile(x...)
  if i_am_main(parts)
    write(io_force,join(x, " ")...)
    write(io_force,"\n")
    flush(io_force)
  end
end
```

1.2 Define the geometry

After that we call the mesh that we want, and extract all tags and boundaries. Throughout the code some key steps get pushed to the log file so that it is easier to troubleshoot the code if, or rather when some simulations were to fail.

```
starting_timestamp = time()
to_logfile("Start time = $starting_timestamp")
# Geometry
to_logfile("Geometry")
DIRICHLET_tags = ["inlet", "walls", "monopile"]
# FLUID_LABEL = "fluid"
# OUTLET_LABEL = "outlet"
meshes_path=ENV["PerforatedCylinder_MESHES"]
full_mesh_path = joinpath(meshes_path,mesh_file)
to_logfile("Mesh file: ",full_mesh_path)
model =  GmshDiscreteModel(parts,full_mesh_path)
Ω = Triangulation(model)
Ω_f = Triangulation(model, tags = "fluid")
Γ_S = Boundary(model, tags = "monopile")
Γ_out = Boundary(model, tags = "outlet")
```

1.3 Define the boundary conditions

We start by defining the model order and assigning it to the domain and its boundaries.

```
to_logfile("Measures")
order = 2
degree = 2 * order
dΩ_f = Measure(Ω_f, degree)
dΓₛ = Measure(Γ_S, degree)
dΓout = Measure(Γ_out, degree)
n_ΓS = get_normal_vector(Γ_S)
n_Γout = get_normal_vector(Γ_out)
```

In addition, we define some general physical parameters:

```
# Physics parameters
to_logfile("Parameters")
rho = 1.025e3 # kg/m^3
μ_f = 1.0e-3 # rho * Vinf * D / Re #0.01 # Fluid viscosity
ν_f = μ_f / rho # kinematic viscosity
```

as well as the boundary conditions for the inflow side, walls and cylinder itself. The outflow boundary condition is immediately incorporated in the weak form. While not used here, this code allows immediate 3D expansion.

```
# Boundary conditions and external loads
dims = num_cell_dims(model)
u0(x,t,::Val{2}) = VectorValue(0.0, 0.0)
u1(x,t,::Val{2}) = VectorValue( Vinf, 0.0 )
u0(x,t,::Val{3}) = VectorValue(0.0, 0.0, 0.0)
u1(x,t,::Val{3}) = VectorValue( Vinf, 0.0, 0.0 )
u0(x,t::Real) = u0(x,t,Val(dims))
u1(x,t::Real) = u1(x,t,Val(dims))
u0(t::Real) = x -> u0(x,t,Val(dims))
u1(t::Real) = x -> u1(x,t,Val(dims))
println("Dimensions in this problem = $dims")
U0_dirichlet = [u1, u1, u0]
g(x) = 0.0
```

2 Set up the numerical scheme

2.1/2.2/2.3 Set up the test and trial spaces and combining them into a transient multifield

With the domain fully defined the next step is to define the internal structure of each cell through reference elements in their respective transient test spaces. With one for velocity (in 2 dimensions), and one for pressure (1 dimension), a multifield test space and trial space is created.
The velocity reference elements in the test spaces are Lagrangian second order elements. For numerical stability, this means that the pressure elements should be Lagrangian elements as well, but of order 1. These 2 test spaces have H1 and C0 conformity respectively. H1 conformity indicates that the test space functions are continuous, although their gradients can contain jumps. C0 conformity allows the test space to contain jumps already, which is required for the pressure field.
With all of that done, we can generate the actual trial and test spaces for transient use.

```
to_logfile("FE spaces")
# ReferenceFE
reffeᵤ = ReferenceFE(lagrangian, VectorValue{dims,Float64}, order)#,space=:P)
reffeₚ = ReferenceFE(lagrangian, Float64, order - 1)#,space=:P)

# Define test FESpaces
V = TestFESpace(Ω, reffeᵤ,  dirichlet_tags = DIRICHLET_tags, conformity = :H1)
Q = TestFESpace(Ω, reffeₚ,   conformity= :C0)
Y = MultiFieldFESpace([V, Q])

# Define trial FESpaces from Dirichlet values
U = TransientTrialFESpace(V, U0_dirichlet)
P = TrialFESpace(Q)
X = TransientMultiFieldFESpace([U, P])
```

2.4 Initialize the domain

To facilitate the calculation start-up, it is better to not start from a no-flow situation, where all velocities are 0. To have an initial state, a Stokes solver is ran once. The strong form of the steady state Stokes equation is:

$$\begin{cases} -\mu\Delta\mathbf{u} + \nabla p = \mathbf{f}, & \text{in } \Omega, \\ -\nabla\mathbf{u} = 0, & \text{in } \Omega \end{cases}$$

where $\mathbf{u}$ denotes the velocity field, 2 dimensional, and p denotes the pressure field. μ is the in-compressible fluid viscosity. Multiplying the test function $v \in \mathbf{H}_0^1(\Omega)$ to the former momentum equation on the top, and ding the same for the continuity equation on the bottom with $q \in L^2(\Omega)$, we can find the weak form problem through integration by parts. The problem then reads:

Find $\mathbf{u} \in \mathbf{H}_0^1(\Omega)$ and $p \in L^2(\Omega)$ such that

$$\begin{cases} (\mu \nabla \mathbf{u}, \nabla \mathbf{v}) - (p, \nabla v) = \langle \mathbf{f}, \mathbf{v} \rangle, & \forall v \in \mathbf{H}_0^1(\Omega) \\ -(\nabla \mathbf{u}, q) = 0, & \forall q \in L^2(\Omega) \end{cases}$$

Finally this can be combined into the bilinear form:

$$\begin{cases} a(\mathbf{u}, \mathbf{v}) = \mu \int_\Omega \nabla \mathbf{u} : \nabla \mathbf{v} d\mathbf{x} \\ b(\mathbf{v}, \mathbf{q}) = -\int_\Omega (\nabla \mathbf{v}) q d\mathbf{x} \end{cases}$$

Rewriting this such that the formulation includes the outflow boundary condition and the non-forced fluid flow inside the domain, and then rewriting the inputs so that the form is equal to the one implemented in Gridap.jl, the problem to be solved becomes:

Find $\mathbf{u} \in \mathbf{H}_0^1(\Omega)$ and $p \in L^2(\Omega)$ such that

$$a\left((u,p),(v,q)\right) = \int \left(\epsilon\left(v\right) \odot \left(\sigma_{dev_f} \cdot \epsilon\left(u\right)\right) - \left(\nabla \cdot v\right) \cdot p + q \cdot \left(\nabla \cdot u\right)\right) d\Omega_f$$

with $\sigma_{dev_f}\left(\epsilon\right) = 2 \cdot \nu_f \cdot \epsilon$ and ϵ the symmetric gradient of the field; and such that

$$l\left((v,q)\right) = \int \left(0\right) d\Omega_f$$

in which the 0 indicates a free fluid flow inside the domain.

This is done through the definition of the weak form:

```
# Stokes for pre-initalize NS
σ_dev_f(ε) = 2 * ν_f * ε #  Cauchy stress tensor for the fluid
a((u, p), (v, q)) = ∫(ε(v) ⊙ (σ_dev_f ∘ ε(u)) - (∇ ⋅ v) * p + q * (∇ ⋅ u))dΩ_f
l((v, q)) = ∫(0.0 * q)dΩ_f
stokes_op = AffineFEOperator(a,l,X(0.0),Y)
```

and the set-up of the parallel solver:

```
# Setup solver via low level PETSC API calls
function mykspsetup(ksp)
    pc       = Ref{GridapPETSc.PETSC.PC}()
    mumpsmat = Ref{GridapPETSc.PETSC.Mat}()
    @check_error_code GridapPETSc.PETSC.KSPSetType(ksp[],GridapPETSc.PETSC.KSPPREONLY)
    @check_error_code GridapPETSc.PETSC.KSPGetPC(ksp[],pc)
    @check_error_code GridapPETSc.PETSC.PCSetType(pc[],GridapPETSc.PETSC.PCLU)
    @check_error_code GridapPETSc.PETSC.PCFactorSetMatSolverType(pc[],GridapPETSc.PETSC.MATSOLVERMUMPS)
    @check_error_code GridapPETSc.PETSC.PCFactorSetUpMatSolverType(pc[])
    @check_error_code GridapPETSc.PETSC.PCFactorGetMatrix(pc[],mumpsmat)
    @check_error_code GridapPETSc.PETSC.MatMumpsSetIcntl(mumpsmat[],  4, 2)
    @check_error_code GridapPETSc.PETSC.MatMumpsSetIcntl(mumpsmat[],  7, 0)
    @check_error_code GridapPETSc.PETSC.MatMumpsSetIcntl(mumpsmat[],  14, 5000)
    @check_error_code GridapPETSc.PETSC.MatMumpsSetIcntl(mumpsmat[],  24, 1)
    @check_error_code GridapPETSc.PETSC.MatMumpsSetCntl(mumpsmat[], 3, 1.0e-10)
    @check_error_code GridapPETSc.PETSC.KSPSetFromOptions(ksp[])
end
```

```
# Linear Solver
to_logfile("Stokes solve")
ls₀ = PETScLinearSolver(mykspsetup)
u_ST, p_ST = solve(ls₀,stokes_op)
# u_ST, p_ST = solve(stokes_op)
```

Lastly this solution is interpolated over the different elements in order to be used as initial conditions for the transient calculations.

```
# initial condition NS
to_logfile("Navier-Stokes operator")
xh₀ = interpolate_everywhere([u_ST, p_ST],X(0.0))
vh₀ = interpolate_everywhere((u0(0),0.0),X(0.0))
```

2.5 Define the weak form

The problem formulation starts with the Navier-Stokes equation:

$$\begin{cases} \partial_t \mathbf{u} - \nu \Delta \mathbf{u} + \mathbf{u} \cdot \nabla \mathbf{u} + \nabla p = \mathbf{f}, & in \Omega \times (0,T) \\ -\nabla \mathbf{u} = 0, & in \Omega \end{cases}$$

with ν the kinematic viscosity. Using the same space definitions as for the Stokes initialization. the weak form with a residual notation then reads:

Find $\mathbf{u} \in \mathbf{H}_0^1(\Omega)$ and $p \in L^2(\Omega)$ such that

$$(\partial_t \mathbf{u}, \mathbf{v}) + B(\mathbf{u}; [\mathbf{u}, p], [\mathbf{v}, q]) = \langle \mathbf{f}, \mathbf{v} \rangle \forall v \in \mathbf{H}_0^1(\Omega) \cap \forall q \in L^2(\Omega)$$

where $B(\mathbf{a}; [\mathbf{u}, p], [\mathbf{v}, q]) = \nu(\nabla \mathbf{u}, \nabla \mathbf{v}) + c(\mathbf{a}, \mathbf{u}, \mathbf{v}) - (p, \nabla \cdot \mathbf{v}) + (q, \nabla \cdot \mathbf{u})$.

The final step is to add a stabilizing closure model:

$$res\,(t,(u,p),(v,q)) \;=\; \int \left(\frac{\partial u}{\partial t} \cdot v + c\,(u,u,v) + \nu_f \cdot (\nabla\,(u) \odot \nabla\,(v)) - p \cdot (\nabla \cdot v) + (\nabla \cdot u) \cdot q \right) d\Omega_f \;+\; \int (\tau_m \cdot ((\nabla\,(u) \cdot$$

This can all be elegantly, in the same formulation, be converted into Gridap code:

```
# Explicit FE functions
global ηₙₕ = interpolate(u0(0),U(0.0))
global uₙₕ = interpolate(u_ST,U(0.0))
global fv_u = zero_free_values(U(0.0))

# Stabilization Parameters
c₁ = 12.0
c₂ = 2.0
cc = 4.0
h2map = map_parts(Ω_f.trians) do trian
CellField(get_cell_measure(trian),trian)
end
h2 = DistributedCellField(h2map)
hmap = map_parts(Ω_f.trians) do trian
CellField(lazy_map(dx->dx^(1/2),get_cell_measure(trian)),trian)
end
h = DistributedCellField(hmap)
τₘ = 1/(c₁*v_f/h2 + c₂*(meas∘uₙₕ)/h)
τc = cc *(h2/(c₁*τₘ))
```

```
# Weak form
c(a,u,v) = 0.5*((∇(u)'·a)·v - u·(∇(v)'·a))
res(t,(u,p),(v,q)) = ∫( ∂t(u)·v  + c(u,u,v) + ε(v) ⊙ (σ_dev_f ∘ ε(u)) - p*(∇·v) + (∇·u)*q +
                        τₘ*((∇(u)'·u - ηₙₕ)·(∇(v)'·u)) + τc*((∇·u)*(∇·v)) )dΩ_f +
                    ∫( 0.5*(u·v)*(u·n_Γout) )dΓout
jac(t,(u,p),(du,dp),(v,q)) = ∫( c(du,u,v) + c(u,du,v) + ε(v) ⊙ (σ_dev_f ∘ ε(du)) - dp*(∇·v) + (∇·du)*q +
                                τₘ*((∇(u)'·u - ηₙₕ)·(∇(v)'·du) + (∇(du)'·u + ∇(u)'·du)·(∇(v)'·u)) +
                                τc*((∇·du)*(∇·v)) )dΩ_f +
                            ∫( 0.5*((du·v)*(u·n_Γout)+(u·v)*(du·n_Γout)) )dΓout
jac_t(t,(u,p),(dut,dpt),(v,q)) = ∫( dut·v )dΩ_f

# Orthogonal projection
aη(η,κ) = ∫( τₘ*(η·κ) )dΩ_f
bη(κ) = ∫( τₘ*((∇(uₙₕ)'·uₙₕ)·κ) )dΩ_f
op_proj = AffineFEOperator(aη,bη,U(0.0),V)
ls_proj = PETScLinearSolver()

# NS operator
op = TransientFEOperator(res, jac, jac_t, X, Y)
```

2.6 Set up the integration scheme

These results, $\mathbf{u}$ and p should then be integrated to the next time step. This is done through the implicit generalized alpha scheme with $\rho_\infty = 0.5$, as defined before.

```
# ODE solver
t₀ = 0.0 # start [s]
ρ∞ = 0.5
ode_solver = GeneralizedAlpha(nls,Δt,ρ∞)
```

3. Apply the nonlinear solver

While being the core of the problem, Gridap allows the solution to be executed in a lazy format with just one line of code.

```
xₜ = solve(ode_solver,op,(xh₀,vh₀),t₀,tf)
```

4. Do post processing

The processes above are compiled as a single function. While calculating each time step, the pressure and velocity are stored. The intermediate numerical artefacts u_{nh} and η_{nh} are stored too. The pressure and velocity fields are used to determine the drag and lift force on the perforated cylinder through:

$$\vec{F} = \left(\sum \int \left(\left(n_\Gamma \cdot \sigma_{dev_f} \left(\epsilon \left(u \right) \right) \right) - p \cdot n_\Gamma \right) \cdot d\Gamma_s \right) \cdot \rho$$

The force vector contains then the total drag and lift components of the cylinder. We use some relay-parameters since the parallel use of global variables gave some discrepancies in the nested loops. Additionally these calculations contain the fluid domain, so integration of pressures determines the forces as experienced by the fluid. To get the forces on the cylinder we add a minus sign.

```
# Postprocess
if i_am_main(parts)
```

```
    println("Postprocess")
    to_logfile("Postprocess")
end
global tout = 0
createpvd(parts,run_name) do pvd
    global t_out_relay = tout
    global Δtout_relay = Δtout
    global u_nh_relay = u_nh
    global η_nh_relay = η_nh
    for ((uh,ph),t) in x_t
        to_logfile("Time: $t")
        to_logfile("========================")
        Fx, Fy = -sum(∫((n_ΓS ⋅ σ_dev_f(ε(uh))) - ph * n_ΓS) * dΓ_s)
        to_forcefile(t,Fx,Fy)
        if t>t_out_relay
        # pvd[t] = createvtk(Ω,joinpath(full_vtk_path,run_name*"_$t"),cellfields=
["u"=>uh,"p"=>ph,"un"=>u_nh_relay,"eta_n"=>η_nh_relay])
        # t_out_relay=t+Δtout_relay
        pvd[t] = createvtk(Ω,run_name*"_$t",cellfields=
["u"=>uh,"p"=>ph,"un"=>u_nh_relay,"eta_n"=>η_nh_relay])
        t_out_relay=t+Δtout_relay
        end
        u_nh_relay = interpolate!(uh,fv_u,U(t))
        η_nh_relay = solve(ls_proj,op_proj)
    end
end
```

Finally the post processing is finished by closing the opened log files and ending the function which was originally called from the main module.

```
if i_am_main(parts)
    close(io)
    close(io_force)
end

ending_timestamp = time()
to_logfile("Start time = $ending_timestamp")
elapsed_time = ending_timestamp - starting_timestamp
to_logfile("elapsed time = $elapsed_time seconds")

return nothing
```