# Lid Driven Cavity Flow (2D)

Incompressible Lid Driven Cavity Flow, solved using first order elements and $\theta$-method ode method

In this example you are going to see:

- Cartesian Mesh
- Multifield problem
- Time dependent problem
- Non Linear solver

```
using Gridap
```

```
function stretching_y_function(x)
        gamma1 = 2.5
        S = 0.5815356159649889 #for rescaling the function over the domain -0.5 ->
0.5
        -tanh.(gamma1 .* (x)) ./ tanh.(gamma1) .* S
end
```

```
    function stretching(x::Point)
        m = zeros(length(x))
        m[1] = stretching_y_function(x[1])


        m[2] = stretching_y_function(x[2])
        Point(m)
    end
```

Parameters defintion

```julia
begin
    L = 0.5 #Domain dimension
    N = 50 #Number of cells for each dimension
    order = 1 #element order
    t0 = 0.0
    tF = 5.0
    dt = 0.01

    D = 2 #dimension (2d or 3d)
    u_in = 1.0 #Lid Velocity

    Re = 1000
    θ = 1.0 # ODE parameter

end
```

```julia
ν = u_in*2*L/Re
```

# Model definition

```julia
begin
    domain = (-L, L, -L, L)
    partition = (N, N)
    model = CartesianDiscreteModel(domain, partition, map=stretching)
end
```

# Set boundary conditions

```julia
begin
    u_wall(x,t) = VectorValue(0.0, 0.0)
    u_wall(t::Real) = x -> u_wall(x,t)


    u_top(x,t) = VectorValue(u_in, 0.0)
    u_top(t::Real) = x -> u_top(x,t)
end
```

```
begin
    labels = get_face_labeling(model)
    add_tag_from_tags!(labels, "diri1", [5,])
    add_tag_from_tags!(labels, "diri0", [1, 2, 4, 3, 6, 7, 8])
    add_tag_from_tags!(labels, "p", [4,])
    u_diri_tags=["diri0", "diri1"]
    u_diri_values = [u_wall, u_top]
    p_diri_tags= "p"
    p_diri_values = 0.0

end
```

Creation FE spaces, MULTIFIELD

```
begin
    reffeᵤ = ReferenceFE(lagrangian, VectorValue{D,Float64}, order)
    reffeₚ = ReferenceFE(lagrangian, Float64, order)
    V = TestFESpace(model, reffeᵤ, conformity=:H1, dirichlet_tags=u_diri_tags)
    U = TransientTrialFESpace(V, u_diri_values)
    Q = TestFESpace(model, reffeₚ, conformity=:H1, dirichlet_tags=p_diri_tags)
    P = TrialFESpace(Q, p_diri_values)
    Y = MultiFieldFESpace([V, Q])
    X = TransientMultiFieldFESpace([U, P])
end
```

# Numerical discretization

```
degree = 4*order;
```

```
Ω = Triangulation(model);
```

```
dΩ = Measure(Ω,degree);
```

Volume force definition

```
begin
    hf(x,t) = VectorValue(0.0, 0.0);
    hf(t::Real) = x -> hf(x,t);
end
```

# Variational Formulation

## Conservation equations

```
Rm(t, (u, p)) = ∂t(u) + u ⋅ ∇(u) + ∇(p) - hf(t);  #- ν*Δ(u)
```

```
dRm((u, p), (du, dp), (v, q)) = du ⋅ ∇(u) + u ⋅ ∇(du) + ∇(dp); #- ν*Δ(du)
```

```
Rc(u) = ∇ ⋅ u;
```

```
dRc(du) = ∇ ⋅ du;
```

$$\int_\Omega \frac{\partial u}{\partial t} \cdot v \, d\Omega + \int_\Omega \nabla v \cdot \nabla u \, d\Omega - \int_\Omega (\nabla \cdot v) \, p \, d\Omega + \int_\Omega q \, (\nabla \cdot u) \, d\Omega + \int_\Omega (u \cdot \nabla(u)) \cdot v \, d\Omega -$$

```
var_eq(t, (u, p), (v, q)) = ∫(∂t(u) ⋅ v)dΩ + ∫((u ⋅ ∇(u)) ⋅ v)dΩ - ∫((∇ ⋅ v) * p)dΩ
+ ∫((q * (∇ ⋅ u)))dΩ + ν * ∫(∇(v) ⊙ ∇(u))dΩ - ∫(hf(t) ⋅ v)dΩ

```

## Stabilization Parameters

Get cell dimension

```
h = lazy_map(h -> h^(1 / D), get_cell_measure(Ω))
```

$$\tau = \left(\frac{2u}{h} + \frac{4\nu}{h^2} + \frac{2}{dt}\right)^{-1}$$

```
function τ(u, h)
        r = 1
        τ₂ = h^2 / (4 * ν)
        val(x) = x
        val(x::Gridap.Fields.ForwardDiff.Dual) = x.value
        u = val(norm(u))

        if iszero(u)
            return τ₂
        end

        τ₃ = dt / 2
        τ₁ = h / (2 * u)
        return 1 / (1 / τ₁^r + 1 / τ₂^r + 1 / τ₃^r)
end;
```

$$\tau_b = (u \cdot u)\tau$$

```
τb(u, h) = (u ⋅ u) * τ(u, h);
```

## Stabilization Equation

$$\int_\Omega \tau \cdot (u \cdot \nabla(v) + \nabla(q)) R_m \, d\Omega + \int_\Omega \tau_b \cdot (\nabla \cdot v) R_c \, d\Omega$$

```
stab_eq(t, (u, p), (v, q)) = ∫((τ ∘ (u, h) * (u ⋅ ∇(v) + ∇(q))) ⊙ Rm(t, (u, p))
+   τb ∘ (u, h) * (∇ ⋅ v) ⊙ Rc(u) )dΩ;
```

```
res_eq(t, (u, p), (v, q)) = var_eq(t, (u, p), (v, q)) + stab_eq(t, (u, p), (v, q));
```

Jacobians

```
begin
    dvar_eq(t, (u, p), (du, dp), (v, q)) = ∫(((du ⋅ ∇(u)) ⋅ v) + ((u ⋅ ∇(du)) ⋅ v)
    + (∇(dp) ⋅ v) +  (q * (∇ ⋅ du)))dΩ + ν * ∫(∇(v) ⊙ ∇(du))dΩ

    dstab_eq(t, (u, p), (du, dp), (v, q)) = ∫(((τ ∘ (u, h) * (u ⋅ ∇(v)' +  ∇(q))) ⊙
    dRm((u, p), (du, dp), (v, q))) + ((τ ∘ (u, h) * (du ⋅ ∇(v)')) ⊙ Rm(t, (u, p)))
    + (τb ∘ (u, h) * (∇ ⋅ v) ⊙ dRc(du)))dΩ

    jac(t, (u, p), (du, dp), (v, q)) = dvar_eq(t, (u, p), (du, dp), (v, q)) +
    dstab_eq(t, (u, p), (du, dp), (v, q))

end;
```

```
jac_t(t, (u, p), (dut, dpt), (v, q)) = ∫(dut ⋅ v)dΩ + ∫(τ ∘ (u, h) * (u ⋅ ∇(v) + (1/
θ)*∇(q)) ⊙ dut)dΩ;
```

```
op = TransientFEOperator(res_eq,jac,jac_t,X,Y);
```

# ODE settings

```
uh0 = interpolate_everywhere(VectorValue(0.0,0.0), U(0));
```

```
ph0 = interpolate_everywhere(0.0, P(0));
```

```
xh0 = interpolate_everywhere([uh0, ph0], X(0));
```

# Solver settings

```
nls_solver = NLSolver(show_trace=true, method=:newton);
```

```
ode_solver = ThetaMethod(nls_solver,dt,θ);
```

```
sol_t = solve(ode_solver,op,xh0,t0,tF)
```

# View Results

```
using Plots
```

```
x = range(domain[1],domain[2]; length=32)
```

```
y = range(domain[3],domain[4]; length=32)
```

```
function gettimeseries(sol)
    pressures = []
    vmagnitudes = []
    coords = Point.(x',y)
    for ((uh, ph),tn) in sol
        push!(pressures, ph.(coords))
        push!(vmagnitudes, norm.(uh.(coords)))
    end
    return pressures, vmagnitudes
end
```

```
# Actual solution happens here, grab a ☕
p,v = gettimeseries(sol_t)
```

```
@bind tstep Slider(1:length(p))
```

```
heatmap(x,y,p[tstep]; aspect_ratio=:equal)
```

```
heatmap(x,y,v[tstep]; aspect_ratio=:equal)
```

# Using PETSc as solver

```
petsc_options = "-snes_type newtonls -snes_linesearch_type basic  -snes_linesearch_damping 1.0 -snes_rtol 1.0e-14 -snes_atol 0.0 -snes_monitor -pc_type asm -sub_pc_type lu  -ksp_type gmres -ksp_gmres_restart 30  -snes_converged_reason -ksp_converged_reason -ksp_error_if_not_converged true ";
```

```
# using GridapPETSc
```

```
#  GridapPETSc.with(args=split(petsc_options)) do
#     petsc_nls_solver = NLSolver(show_trace=true, method=:newton);
#     petsc_ode_solver = ThetaMethod(petsc_nls_solver,dt,θ);
#     petsc_sol_t = solve(petsc_ode_solver,op,xh0,t0,tF);
#    createpvd("lid_driven") do pvd
#      for (xh_tn,tn) in petsc_sol_t
#         uh_tn, ph_tn = xh_tn
#        pvd[tn] = createvtk(Ω,"lid_driven_$tn"*".vtu",cellfields=["uh"=>uh_tn,
"ph"=>ph_tn])
#      end
#    end
#  end
```

# Other features

- Alpha-method for ODE
- Higher-order elements
- Discontinuous Galerkin with interior penalty
- Adjoint optimization
- Uniform refinement