Computational dynamics

Computer Laboratory 1: Introduction to Elastodynamics in FEniCSx. Boundary Conditions

Dr. Miguel Ángel Moreno-Mateos

Institute of Applied Mechanics (LTM, Paul Steinmann)

Friedrich-Alexander-Universität Erlangen-Nürnberg

Summer semester 2025

OUTLINE

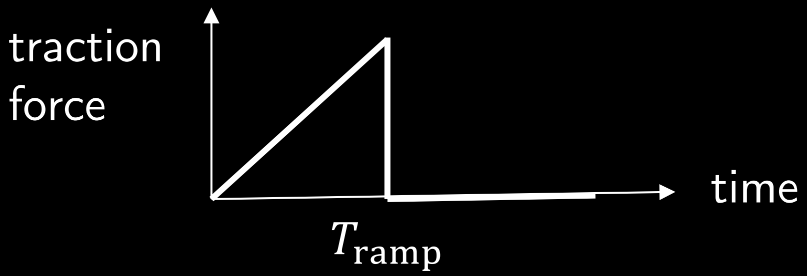
- Introduction
- Demo code
- Tasks

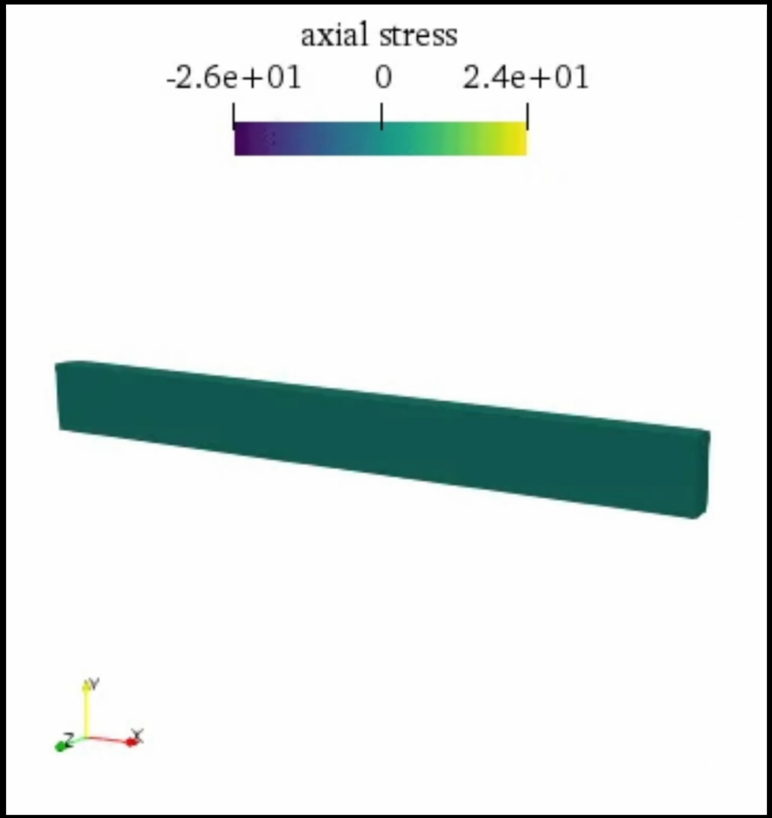
INTRODUCTION

- Reminder: Installation and Demo Code on https://github.com/MorenoMiguelES/CD-computdynamics

- Tutorial on Elastodynamics on FEniCS legacy (2019): https://comet-fenics.readthedocs.io/en/latest/demo/elastodynamics/demo_elastodynamics.py.html

- Fixed on Dirichlet boundary on left side.
- Traction forces on Neumann boundary on right side.





(Bleyer, J., comet demo)

DEMO CODE, FEniCSx v0.9.0

Import modules

```
"""
Demo code for the Computational Dynamics SS2025 course
ELASTODYNAMICS: Newmark-beta method, traction forces / displacement loading.
Author: Dr. Miguel Angel Moreno-Mateos
"""

#Import: not necessary to specify it later
from __future__ import print_function
import numpy as np
from numpy import array
import dolfinx
import ufl
from mpi4py import MPI
import dolfinx.io
import dolfinx.geometry
import math
from petsc4py.PETSc import ScalarType
from petsc4py import PETSc
import petsc4py
import matplotlib.pyplot as plt
from petsc4py import PETSc
default_scalar_type = PETSc.ScalarType
from dolfinx import fem
import basix
import dolfinx.fem.petsc
import dolfinx.nls.petsc
```

DEMO CODE, FEniCSx v0.9.0

Import mesh:

Gmsh

```
#Import mesh & subdomains from .msh file:
filename="Box"
from dolfinx.io import gmshio
mesh, cell_markers, facet_markers = gmshio.read_from_msh(filename+'.msh',
MPI.COMM_WORLD, gdim=3)

metadata = {"quadrature_degree": 2}
dx = ufl.Measure('dx')(domain=mesh, subdomain_data=cell_markers,
metadata=metadata) #Integration measures for each subdomain
```

Function Space & functions:

```
P1 = basix.ufl.element("CG", mesh.basix_cell(), 2, shape=(mesh.geometry.dim,)) #
For displacements
V = dolfinx.fem.functionspace(mesh, P1)


# FUNCTIONS IN THE FUNCTION SPACE V (Function - Trial - Test).
u = dolfinx.fem.Function(V) #Unkown.
u_old = dolfinx.fem.Function(V)
v_old = dolfinx.fem.Function(V)
a_old = dolfinx.fem.Function(V)
du = ufl.TrialFunction(V)
v = ufl.TestFunction(V)
dd = len(u) # space dimension
```

DEMO CODE, FEniCSx v0.9.0

Boundary conditions:

a) Encastre $\partial\Omega_D$:

b) Traction forces $\partial\Omega_N$:

```
########## BOUNDARY CONDITIONS ############################################
###### Encastre Displacement:
def bnd_encastre(x):
tol=1e-4
return np.isclose(x[2],0)
bnd_encastre_dofs0 = dolfinx.fem.locate_dofs_geometrical(V,bnd_encastre)
u_encastre_vect0 = np.array((0,) * mesh.geometry.dim, dtype=default_scalar_type)
bc_boundar = dolfinx.fem.dirichletbc(u_encastre_vect0,bnd_encastre_dofs0,V)

# Definition of 2 problems: bcu1 with Dirichlet BC on right side,
#bcu2 without them for free oscillation. If traction forces, it is not necessary.
bcu1 = [bc_boundar]#, bc_dispload] #Step 1: Dirichlet BC
bcu2 = [bc_boundar] #Step 2: Oscillation


########## Option B:TRACTION FORCES #####################################
def bnd_side(x):
return np.isclose(x[2],1)
forcemax = 1
t_ = dolfinx.fem.Constant(mesh, 0.0)
traction = ufl.as_vector([forcemax * (t_)/(T_ramp),0,0])
right_facets = dolfinx.mesh.locate_entities_boundary(mesh, mesh.topology.dim - 1,
bnd_side)
facet_tag = dolfinx.mesh.meshtags(mesh, mesh.topology.dim - 1, right_facets, 1)
dss = ufl.Measure('ds', domain=mesh, subdomain_data=facet_tag, subdomain_id=1)#,
metadata=metadata)
#########################################################################
```

DEMO CODE, FEniCSx v0.9.0

Time discretization: Newmark Method

$\gamma \in [0,1]$,

$\beta \in [0,0.5]$,

```
########### NEWMARK METHOD (time discretisation) ######################
# alpha_m = alpha_f = 0: recovers Newmark method. Otherwise, generalized-alpha
method.
alpha_m = 0 # in [0,1]
alpha_f = 0 # in [0,1]
# HIGH-FREQUENCY DISSIPATION (NUMERICAL DAMPING) IN NEWMARK METHOD:
gamma = 1
beta = 0.5
######################################################################

########### RAYLEIGH DAMPING ##########################################
# Note: Not necessary for high-frequency dissipation if gamma and beta are
properly chosen.
eta_m = 0 #1e-4
eta_k = 0 #1e-4
######################################################################

def avg(x_old,x_new,alpha):
return alpha*x_old + (1-alpha)*x_new
def update_a(u,u_old,v_old,a_old):
dt_ = dt
beta_ = beta
return (u-u_old-dt_*v_old)/beta_/dt_**2 - (1-2*beta_)/2/beta_*a_old
def update_v(a,u_old,v_old,a_old):
dt_ = dt
gamma_ = gamma
return v_old + dt_*((1-gamma_)*a_old + gamma_*a)
a_new = update_a(u,u_old,v_old,a_old)
v_new = update_v(a_new,u_old,v_old,a_old)
```

DEMO CODE, FEniCSx v0.9.0

Primary fields (Kinematics):

```
## PRIMARY FIELDS
dd = len(u) # Spatial dimension
I = ufl.Identity(dd) # Identity tensor
F = I + ufl.grad(avg(u_old,u,alpha_f)) # Deformation gradient from current time
step
Fv = I + ufl.grad(avg(v_old,v_new,alpha_f)) # Deformation gradient from current
time step
CG = ufl.dot(F.T,F) # Right Cauchy-Green (CG) tensor
BG = ufl.dot(F,F.T) # Left Cauchy-Green (CG) tensor
```

DEMO CODE, FEniCSx v0.9.0

Constitutive model:

```
## CONSTITUTIVE MODEL:
######## YEOH-MODEL (COMPRESSIBLE)
C10 = [0.5*1000, 0]
C20 = [0, 0]
C30 = [0, 0]
poisson = [0.3, 0]
kappa = [2*(2*C10[0])*(1+poisson[0])/3/(1-2*poisson[0]),
2*(2*C10[0])*(1+poisson[0])/3/(1-2*poisson[0])]
#density:
rho = 1.0
def Piso(F,i):
j,k,l,m = ufl.indices(4)
Id = ufl.Identity(3)
FinvT = ufl.inv(F).T
J = ufl.det(F)
A1 = ufl.as_tensor(Id[j,l]*Id[k,m],(j,k,l,m))
A2 = ufl.as_tensor(-1/3*FinvT[j,k]*F[l,m],(j,k,l,m))
Pfourth = A1+A2
Fiso = 1/ufl.det(F)**(1/3)*F
I1 = ufl.tr(ufl.dot(Fiso.T,Fiso) )
P_yeoh= 2* ( C10[i] + C20[i] *2* (I1-3)**1 + C30[i] *3 * (I1-3)**2 )* Fiso
P = 1/(J**(1/3))*ufl.as_tensor(Pfourth[j,k,l,m]*P_yeoh[l,m],(j,k))
return P
def Pvol(F,i): #volumetric part
J = ufl.det(F)
Pvol= kappa[i]*J*(J-1)*ufl.inv(F).T
return Pvol
```

DEMO CODE, FEniCSx v0.9.0

Weak form:

```
####### VARIATIONAL PROBLEM - WEAK FORM
Res_u =
(ufl.inner(Piso(F, 0), ufl.grad(v)) + ufl.inner(Pvol(F, 0), ufl.grad(v))\
+ eta_m*rho*ufl.inner(avg(v_old,v_new,alpha_m),v)\
+ eta_k*(ufl.inner(Piso(Fv, 0), ufl.grad(v)) + ufl.inner(Pvol(Fv, 0), ufl.grad(v)))\
+ rho*ufl.inner(avg(a_old,a_new,alpha_m),v)) * dx - ufl.inner(traction,v) * dss(1)
```

DEMO CODE, FEniCSx v0.9.0

FEniCSx problem:

```
# Setup Non-linear variational problem
problem_u1 = fem.petsc.NonlinearProblem(Res_u,u,bcu1)
from dolfinx import nls
solver_problem_u1 = nls.petsc.NewtonSolver(mesh.comm, problem_u1)
solver_problem_u1.atol = 1e-8
solver_problem_u1.rtol = 1e-8
solver_problem_u1.convergence_criterion = "incremental"

ksp = solver_problem_u1.krylov_solver
opts = PETSc.Options()
option_prefix = ksp.getOptionsPrefix()
opts[f"{option_prefix}ksp_type"] = "preonly" # "preonly" works equally well
opts[f"{option_prefix}pc_type"] = "lu" # do not use 'gamg' pre-conditioner
opts[f"{option_prefix}pc_factor_mat_solver_type"] = "mumps"
opts[f"{option_prefix}ksp_max_it"] = 30
ksp.setFromOptions()

# Note: defining two non linear problems would allow to set different Dirichlet
BC before and after T_ramp
problem_u2 = fem.petsc.NonlinearProblem(Res_u,u,bcu2)
solver_problem_u2 = nls.petsc.NewtonSolver(mesh.comm, problem_u2)
solver_problem_u2.atol = 1e-8
solver_problem_u2.rtol = 1e-8
solver_problem_u2.convergence_criterion = "incremental"
```

DEMO CODE, FEniCSx v0.9.0

Initialisation of variables to store data:

Note: output file is VTK format.

```
# Save results into vtk files
uVector= dolfinx.fem.Function(VV, name = "displacement")
FTensor=dolfinx.fem.Function(TT, name = "F")
PTensor=dolfinx.fem.Function(TT, name = "P")
mesh.topology.create_connectivity(mesh.topology.dim-1, mesh.topology.dim)
vtx_writer=dolfinx.io.VTXWriter(mesh.comm, "Output.bp", [uVector, FTensor,
PTensor], engine="BP4")

##### EVALUATE DISPLACEMENT AT A POINT ########################
time_vec = []
disp_x_point1 = []
point1 = np.array([[0.0, 0.0, 1.0]])

# Create a PointLocator for finding cells containing points
mesh.topology.create_connectivity(mesh.topology.dim, mesh.topology.dim)
bb_tree = dolfinx.geometry.bb_tree(mesh, mesh.topology.dim)

# Find cells containing the points
cell_candidates = dolfinx.geometry.compute_collisions_points(bb_tree, point1)
colliding_cells = dolfinx.geometry.compute_colliding_cells(mesh, cell_candidates,
point1)

points_on_proc = []
cells = []

for i, point in enumerate(point1):
if len(colliding_cells.links(i)) >= 0: # Check if point was found on this process
points_on_proc.append(point)
cells.append(colliding_cells.links(i)[0])

print(points_on_proc)
```

DEMO CODE, FEniCSx v0.9.0

Time-incremental solving loop:

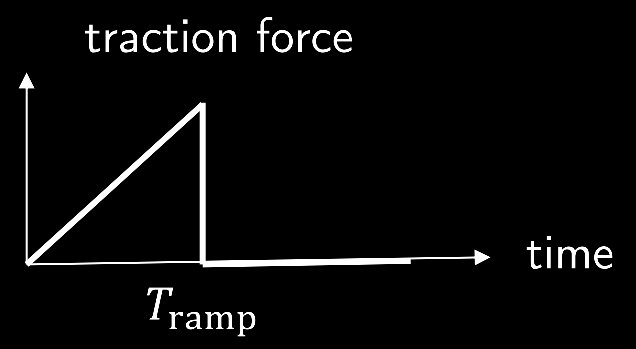
```
###### TIME-INCREMENTAL SOLVING LOOP ########################
while (tiempo < T):
    #Display report simulation
    from dolfinx import log
    log.set_log_level(log.LogLevel.INFO)
    ##Update time-dependent parameters:
    loaddisp_.t = tiempo
    loaddisp.interpolate(loaddisp_.eval)

    if tiempo < T_ramp:
        t_.value = tiempo
    else:
        t_.value = 0

    if tiempo < T_ramp: solver_problem_u1.solve(u)
    else: solver_problem_u2.solve(u)

    a_vec = dolfinx.fem.Expression(update_a(u,u_old,v_old,a_old),
    V.element.interpolation_points());
    v_vec = dolfinx.fem.Expression(update_v(update_a(u,u_old,v_old,a_old),u_old,v_old,a_old),
    V.element.interpolation_points());
    u_vec = dolfinx.fem.Expression(u, V.element.interpolation_points());
    a_old.interpolate(a_vec)
    v_old.interpolate(v_vec)
    u_old.interpolate(u_vec)

    u_expr = dolfinx.fem.Expression(u, VV.element.interpolation_points())
    uVector.interpolate(u_expr)
    F_expr = dolfinx.fem.Expression(F, TT.element.interpolation_points())
    FTensor.interpolate(F_expr)
    P_expr = dolfinx.fem.Expression(Piso(F,0)+Pvol(F,0), TT.element.interpolation_points())
    PTensor.interpolate(P_expr)
```



DEMO CODE, FEniCSx v0.9.0

Save displacement; update output file:

```
# EVALUATE DISPLACEMENT AT A POINT (CONTINUATION)#####
points_on_proc = np.array(points_on_proc, dtype=np.float64)
disp1 = u.eval(points_on_proc, cells)
disp_x_point1 = np.append(disp_x_point1,disp1[0])
time_vec = np.append(time_vec,tiempo)
######################################################

# VTK output
vtx_writer.write(tiempo)
np.savetxt("displacement_x_point1.txt", disp_x_point1)
np.savetxt("time.txt", time_vec)

plt.figure()
plt.plot(time_vec[:],disp_x_point1[:])
plt.grid();plt.xlabel("Time [s]");plt.ylabel("Displacement x [mm]")
plt.savefig('Oscillation_x_point1.png')
plt.close()
# Define next time step
steps += 1
tiempo += dt
```

TASKS

1. **Run simulation.**

   Terminal, navigate to folder, activate `conda` environment, run python script.

2. **Visualize the output file in Paraview.**

   Drag VTK output folder to ParaView workspace. Display nodal values, clips, etc.

3. Modify code: **Replace Neumann BC by Dirichlet BCs.**

Call the instructor for doubts and after successful completion of each task and report it. Enjoy!

HINT TASK 3. APPLY DIRICHLET BCs

Dirichlet BC:

```
########## BOUNDARY CONDITIONS ############################################

###### Option A: Displacement load: #######################################
def bnd_point(x):
    return np.isclose(x[2],1) & np.isclose(x[0],0.04)
dispmax = 0.2
class MyDisp:
    def __init__(self):
        self.t = 0.0
    def eval(self,x):
    #.   Add some spatial/temporal variation here:
        #x.shape[1]: number of columns.
        if True:
            return np.full(x.shape[1], dispmax * (self.t)/(T_ramp))
        else:
            return np.full(x.shape[1], dispmax * (self.t)/(T_ramp))
V_real = dolfinx.fem.functionspace(mesh, ("CG",2))
loaddisp_ = MyDisp()
loaddisp_.t = 0.0
loaddisp = dolfinx.fem.Function(V_real)
loaddisp.interpolate(loaddisp_.eval)
bnd_load_dofs0 =
dolfinx.fem.locate_dofs_geometrical((V.sub(0),V.sub(0).collapse()[0]),bnd_point)
bc_dispload = dolfinx.fem.dirichletbc(loaddisp,bnd_load_dofs0,V.sub(0))
###########################################################################

# Definition of 2 problems: bcu1 with Dirichlet BC on right side,
#bcu2 without them for free oscillation. If traction forces, it is not necessary.
bcu1 = [bc_boundar, bc_dispload]#, bc_dispload] #Step 1: Dirichlet BC
bcu2 = [bc_boundar] #Step 2: Oscillation
```