A. Bio-inspired generative design algorithm

The bio-inspired algorithm is divided into four main scripts. First, a `json` script provides the parameters of the simulation. This file is read by a `python` script that executes the mesh/remesh algorithm (a `python` script) and code aster through a `comm` file.

A.1. The parameter script

Below it is shown an example of the `json` parameter file. It is divided in six section: initial geometry, mesh, file names, model, boundary conditions, and code aster settings.

The initial geometry is defined with two entries: “mai” and “esc”. Each entry has two obligatory entries: “main” and “aux”. The former describes the geometry by means of a closed loop of “lines” described by a series of “points”. The latter describes an auxiliary line indicated the zone of finer mesh size.

The mesh is defined by the minimum (“lcMin”) and maximum (“lcMax”) mesh size used at the expected contact zone (dependent on “fine_x_rate”) and far from the contact zone, respectively. The transition size between these zones is controlled by the “sizeGrwRate”. The contact algorithm used in code aster divides the slave contact elements into three; therefore, the real minimum size is “lcMin”/3. During the simulation, if the maximum size of the slave elements is higher than “lcMin*lcLimFactor”, the geometry is remeshed. The last mesh entry is used in code aster to create new element and node groups using the code aster command `DEFI_GROUP`.

“fileNames” indicates the name of the mesh and remeshing script (“geoGen”), the code aster file (“dummFile”) and the name of the directory where the simulation will be run.

The model starts with “materials” including the elastic properties and the type of plane model (“D_PLAN” refers to plane strain in code aster). It also contains the growth parameters with “maxDis” referring to $\mathrm{D_g}$, “Gr” to α_g, “tauLim” to τ_lim, “sigLim” to σ_lim, “vel” to ϑ, and “a_f” to $l_{c_f}/2$. “timeParams” indicates the maximum desired number of iterations (“final”/“deltaT”), and the maximum number of outputs. Lastly, “algoParams” controls the number of load steps allowed to solve the contact problem.

In “boundary_condition”, the name of the element groups is given (“esc_groups” and “mai_groups”) as well as the groups defining the master (“GROUP_MA_MAIT”) and the slave contact boundaries (“GROUP_MA_ESCL”). The known displacements during the contact and the growth step are respectively defined in “contactDisps” and “growthDisps”; the are used in the code aster function `AFFE_CHAR_MECA`. Lastly, the contact load is also defined with “contactLoad”. It indicates M_f through “M” and $S_f/2$ through “FY”.

```
{
    "initialGeometry": {
        "esc": {"main": {"lines": [{"physical_name": "esc_con",
                                    "type": "circlearc",
                                    "ordered_points": [0, 4, 1]},
                                   {"physical_name": "esc_dxR",
                                    "type": "straight",
                                    "ordered_points": [1, 2]},
                                   {"physical_name": "esc_dy",
                                    "type": "straight",
                                    "ordered_points": [2, 3]},
                                   {"physical_name": "esc_dxM",
                                    "type": "straight",
                                    "ordered_points": [3, 0]}],
                         "points": [[0.0, 0.0], [1.0, 0.029],
                                    [1.0, 2.0], [0.0, 2.0],
                                    [0.0, 17.26]]},
                "aux": {"lines": [{"physical_name": "esc_aux",
                                   "type": "straight",
                                   "ordered_points": [0, 1]}],
                        "refLine": ["esc_con"],
                        "points": [[0.0, 0.0], [0.44, 0.0]]}},
        "mai": {"flip_boundaries": ["mai_con"],
                "main": {"lines": [{"physical_name": "mai_con",
                                    "type": "straight",
                                    "ordered_points": [0, 1]},
                                   {"physical_name": "mai_dxR",
                                    "type": "straight",
                                    "ordered_points": [1, 2]},
                                   {"physical_name": "mai_dy",
                                    "type": "straight",
                                    "ordered_points": [2, 3]},
                                   {"physical_name": "mai_dxM",
                                    "type": "straight",
                                    "ordered_points": [3, 0]}],
                         "points": [[0.0, 0.0], [3.0, 0.0],
                                    [3.0, -3.0], [0.0, -3.0]]},
                "aux": {"lines": [{"physical_name": "mai_aux",
                                   "type": "straight",
                                   "ordered_points": [0, 1]}],
                        "refLine": ["mai_con"],
                        "points": [[0.0, 0.0], [0.44, 0.0]]}}
    },
    "mesh": {
        "lcMin": 0.0025,
        "lcMax": 0.1,
        "sizeGrwRate": 0.25,
        "lcLimFactor": 0.75,
        "fine_x_rate": 1.25,
        "newgroups":
            [{"CREA_GROUP_MA": [{"NOM": "w_dx0",
                                 "UNION": ["esc_dy", "esc_dxR",
                                           "mai_dy", "mai_dxR"]},
                                {"NOM": "w_dy0",
                                 "UNION": ["esc_dy", "mai_dy"]}]},
             {"CREA_GROUP_NO": [{"TOUT_GROUP_MA": "OUI"}]},
             {"CREA_GROUP_NO": [{"NOM": "esc_p0",
                                 "INTERSEC": ["esc_con",
                                              "esc_dxM"]},
                                {"NOM": "mai_p0",
                                 "INTERSEC": ["mai_con",
                                              "mai_dxM"]},
                                {"NOM": "esc_pR",
                                 "INTERSEC": ["esc_con",
                                              "esc_dxR"]}]},
             {"CREA_GROUP_NO": [{"NOM": "wear_dx0",
                                 "DIFFE": ["w_dx0", "esc_pR"]},
                                {"NOM": "wear_dy0",
                                 "DIFFE": ["w_dy0", "esc_pR"]}]}]
    },
    "fileNames": {
        "geoGen": "/geometry.py",
        "dummFile": "/GrowthWearGeneral.dumm",
        "asterFolder": "/Tests/Be_0.4_St_1.0"
    },
    "model": {
        "materials": {"MODELISATION": "D_PLAN",
                      "E": 1000.0,
                      "nu": 0.3},
        "growthParams": {"maxDis": 0.1,
                         "Gr": 6.25,
                         "tauLim": 0.5,
                         "sigLim": 0.5,
                         "vel": 7.1,
                         "a_f": 0.5},
        "timeParams": {"deltaT": 1.0,
                       "final": 100.0,
                       "prints": 20},
        "algoParams": {"maxSteps": 200,
                       "minSteps": 1,
                       "maxNewton": 50}
    },
    "boundary_condition": {
        "esc_groups": ["esc_con", "esc_dxR", "esc_dy", "esc_dxM"],
        "mai_groups": ["mai_con", "mai_dxM", "mai_dy", "mai_dxR"],
        "GROUP_MA_MAIT": ["mai_con"],
        "GROUP_MA_ESCL": ["esc_con"],
        "contactDisps": {"DDL_IMPO": [{"DX": 0.0,
                                       "GROUP_MA": ["esc_dxM",
                                                    "mai_dxM"]},
                                      {"DY": 0.0,
                                         "GROUP_MA": ["mai_dy"]}]},
        "growthDisps": {"DDL_IMPO": [{"DX": 0.0,
                                      "GROUP_MA": ["esc_dxM",
                                                   "mai_dxM"]},
                                     {"DY": 0.0,
                                      "GROUP_NO": ["esc_p0",
                                                   "mai_p0"]}],
                        "LIAISON_UNIF": [{"DDL": "DY",
                                          "GROUP_MA": "mai_dy"},
                                         {"DDL": "DY",
                                          "GROUP_MA": "esc_dy"}]},
        "contactLoad": [{"M": 0.0,
                         "GROUP_MA": ["esc_dy"],
                         "FY": -0.5}]},
    "code_aster": {"dict_P": {"memory_limit": 4000}},
    "c_a_unit": 34
}
```

A.2. The execution script

The algorithm is executed with:

```
python3 run.py -i params.json
```

where run.py is the execution script shown bellow. This script requires two in-house modules: AsterStudyUtilities and datfile. The former is used to create the code aster simulation and the latter to deal with the reported data.

```
#!/usr/bin/env python3
# Libraries {{{
import os
import sys
import json
import getopt
import time
import psutil
import numpy as np
import multiprocessing

# In-house modules
abspath = os.path.dirname(os.path.abspath(__file__))
sys.path.append(abspath + '/PythonUtilities/')
import AsterStudyUtilities as asus
from datfile import datfile
# }}}
```

The console parameters are read with getopt where the name of the input file ("params.json") is provided. Then, the parameters are read from config.json (default parameters) and params.json.

```
# Start time
startTime = time.time()
```

```
# Get console parameters {{{
opts, args = getopt.getopt(sys.argv[1:], 'i:kcpad')
...
# }}}

# Parameters {{{
# Load default parameter file
srcFolder = os.path.dirname(os.path.abspath(__file__))
with open(os.path.join(srcFolder, 'config.json')) as fle:
    defaultParameters = json.load(fle)
locals().update(defaultParameters)
memory_limit_default = psutil.virtual_memory().available
   *0.1/1000000.0
# Load specific parameter file
with open(parFile) as fle:
    params = json.load(fle)
fileNames.update(params.get("fileNames", {}))
codeParams = params.get("code_aster", {})
modelParams     = params["model"]

locals().update(fileNames)
# }}}
```

Once the parameters are read, the time-iteration variables at set and tested for consistency.

```
# Set up iteration parameters {{{
timeParams = modelParams["timeParams"]
final   = timeParams["final"]
deltaT  = timeParams["deltaT"]
prints  = timeParams["prints"]
numItes = int(abs(final/deltaT))
if numItes < prints:
    raise ValueError("Error: timeParams are not consistent. There
   are too many prints.")
if not numItes%prints == 0:
    raise ValueError("The number of prints (" + str(prints) + ")
   is not a multiple of the number of iterations (" + str(numItes)
    + ").")
# }}}
```

Next, necessary path names are determined.

```
# Compute necessary paths {{{
baseDir = os.getcwd()
if asterFolder[:5] == "/home":
    workDir = asterFolder
else:
    workDir = baseDir + asterFolder
if parFile[:5] == '/home':
    paramsPath = parFile
else:
    if continuee or not remove:
        paramsPath = os.path.join(baseDir, parFile)
    else:
        paramsPath = os.path.join(workDir, parFile)

geoGenerator = baseDir + geoGen
# }}}
```

Finally, the code aster study is created setting up the `export` code aster file; the mesh is generated; and the code aster study is run. Here, the algorithm enters in a loop that is in charge of re-executing the mesh generator and the code aster study if the maximum contact element size gets too large or the minimum element quality too low.

```
# Initialisation of the growth process {{{
if not continuee:
    # Set up aster study
    study.CreateStudy(deletePrevious = remove, workDir = workDir)
    with open(paramsPath, 'w') as fle:
        json.dump(params, fle, indent = 4)
    os.makedirs(workDir + resuFolder, exist_ok = True)

    dummies = [('#workDir', workDir),
               ('#parFile', paramsPath),
               ('#srcDir', srcFolder)]
    asus.ReplaceDummyFile(srcFolder + '/codeasters' + dummFile,
               workDir + commFile, dummies)
    # Create mesh
    if makeMesh:
        fail = os.system("python3 {} -i {}".format(geoGenerator,
    paramsPath))
        if fail != 0:
            raise ValueError("Unsuccessful meshing.")
    makeMesh = True
    # Run aster study
    fail = study.RunStudy(outSalome = True)
# }}}

# Continue simulating if necessary {{{
# Get the last time simulated
meshReport = datfile(workDir + meshSizeFile).datablocks['0'].
    variables
ite = meshReport["ite"][-1]
run = True if ite < numItes else False
# Set prevIte to avoid error loops
prevIte = -1
while run:
    # Rewrite export
    study.CreateStudy(deletePrevious = False, workDir = workDir)
    # Remesh if necessary
    if makeMesh:
        fail = os.system("python3 {} -i {} -r".format(geoGenerator
    , paramsPath))
        if fail != 0:
            raise ValueError("Unsuccessful remeshing.")
    makeMesh = True
    # Run aster study
    fail = study.RunStudy(outSalome = True)
    if fail:
        if prevIte == ite:
            raise ValueError("Error: unsuccesful aster study, even
    after remesh.")
        else:
            prevIte = ite
    # Get the last time simulated
    meshReport = datfile(workDir + meshSizeFile).datablocks['0'].
    variables
    ite = meshReport["ite"][-1]
    run = True if ite < numItes else False
# }}}
```

A.3. The mesh script

The algorithm is executed with:

```
python3 geometry.py -i params.json -r
```

by `run.py`; `-r` is optional and indicates a remeshing step. This script requires two in-house modules: `mesh` and `datfile`. The former is used to create the mesh using `Gmsh` [1], and the latter to deal with the reported data.

```
#!/usr/bin/env python3
# Libraries {{{
import os
import sys
import json
import getopt
import time
import psutil
import gmsh
import numpy as np
from scipy.interpolate import UnivariateSpline
from matplotlib import pyplot as plt
from shapely.geometry import LineString
from scipy import integrate

# In-house modules
srcFolder = os.path.join(os.getcwd(), '../../src/')
sys.path.append(os.path.join(srcFolder, 'PythonUtilities'))
import mesh
from datfile import datfile
# }}}
```

The console parameters are read with `getopt` where the name of the input file (“params.json”) is provided.

```
# Get the file with the parameters {{{
```

```
opts, args = getopt.getopt(sys.argv[1:], 'i:r')
...
# }}}
```

Then, the parameters are read from `config.json` (default parameters) and `params.json`, and the mesh settings are extracted.

```
# Read parameters {{{
# Default
with open(os.path.join(srcFolder, 'config.json')) as fle:
    defaultParameters = json.load(fle)
locals().update(defaultParameters)
# Specific
with open(parFile) as fle:
    params = json.load(fle)

initialGeometry = params["initialGeometry"]
meshParams = params["mesh"]
fileNames.update(params.get("fileNames", {}))

locals().update(fileNames)
workDir = os.path.dirname(parFile)
# }}}

# Set mesh parameters {{{
sizeMin = meshParams["lcMin"]
sizeGrwRate = meshParams["sizeGrwRate"]
distMin = meshParams.get("distMin", sizeMin*5.0)
distGrwRate = meshParams.get("distGrwRate", sizeGrwRate*5.0)
sizeMax = meshParams.get("lcMax", 10.0*sizeMin)
fine_x_rate = meshParams.get("fine_x_rate", 2.0)
# }}}
```

The script is divided in two options either creating a new mesh from scratch or to remesh an existing one. In the first case, the mesh is created from the geometric parameters indicated in “initialGeometry” of `params.json` using the class `Geometry` of `mesh`.

```
if newGeo:
    # Make geometries {{{
    # Make meshes
    for name, geo in initialGeometry.items():
        mainParams = geo["main"]
        auxParams  = geo["aux"]
        main = mesh.Geometry(name, **mainParams)
        aux = mesh.Geometry("aux", **auxParams)
        main.MakeGeometry()
        aux.MakeGeometry(makeFace = False, clearAll = False)
        curveList = [aux.lines[0]["id"]]
        mesh.MeshFieldDistanceCurve(0, curveList, sizeMin, distMin
  ,
                    sizeGrwRate, distGrwRate, sizeMax)
        main.MakeMesh()
        main.Export(folder = workDir, fmt = geomFMT)
    # }}}
```

In the second case, the geometry is reconstructed using splines for each physical line conserving the initial names and organisation (see the organisation of “lines” and “points” in “initialGeometry”). The mesh reconstruction creates a `Geometry` object, as in the first case, that can be used to create the mesh.

```
else:
    # Remesh {{{
    # Get xlims
    meshReport = datfile(workDir + meshSizeFile).datablocks['0'].variables
    xlims = [meshReport["xlim0"][-1], meshReport["xlim1"][-1]]
    xmin = min(xlims)
    xmax = max(xlims)
    xdist = xmax - xmin
    xmin -= xdist*fine_x_rate
    xmax += xdist*fine_x_rate
    # Set old mesh file name
    oldMeshFile = workDir + recoFile
    # Reconstruct and remesh geometry
    for name, geo in initialGeometry.items():
        # Read initial geometry parameters
        mainParams = geo["main"]
        max_x = mainParams.get("max_x", 10.0e+9)
        flip_boundaries = mainParams.get("flip_boundaries", [])
        auxParams  = geo["aux"]
        oldpoints = mainParams["points"]
        oldlines = mainParams["lines"]
        refLine = auxParams["refLine"]
        # Set reference line list
        if isinstance(refLine, str):
            refLine = [refLine]
        # Reconstruct geometries
        main = mesh.Reconstruct(name, oldpoints, oldlines, oldMeshFile)
        cutGeom_points, cutGeom_lines = main.CutGeometry(xmax = max_x,
                flip_boundaries = flip_boundaries)
        #main = mesh.Geometry(name, cutGeom_points, cutGeom_lines)
        # Create cut reference lines
        cutLines = []
        for refLine_i in refLine:
            cutLines.append(main.CutLine(refLine_i, xmin = xmin,
            xmax = xmax))
        # Create auxiliary lines
        auxLines = []
        k_i = 0
        for cutLine_i in cutLines:
            if len(cutLine_i[0]) == 0:
                pass
            else:
                auxLines.append(mesh.Geometry("aux{}".format(k_i),
    *cutLine_i))
        # Make geometries
        main.MakeGeometry()
        curveList = []
        for aux in auxLines:
            aux.MakeGeometry(makeFace = False, clearAll = False)
            curveList.append(aux.lines[0]["id"])
        # Make mesh
        mesh.MeshFieldDistanceCurve(0, curveList, sizeMin, distMin
    ,
                sizeGrwRate, distGrwRate, sizeMax)
        main.MakeMesh()
        main.Export(folder = workDir, fmt = geomFMT)
    # }}}
```

A.4. The code aster script

The algorithm is executed with:

```
/opt/aster/bin/as_run PATH/Study.export
```

where /opt/aster/bin/as_run is code aster path of installation. This script requires six in-house modules. FemMesh2D allows the computation of mesh quality; MorphoDesignFunctions contains the growth functions; PyAster contains in-house macros of code aster functions; datfile deals with the reported data; AsterStudyUtilities is used here to delete unnecessary code aster files; and LineMeasures allows the extraction of contact lines and the computation of curvature and conformity metrics.

```
# Libraries {{{
import os
import sys
import json
import numpy as np
from Utilitai.partition import *
import time
import psutil
from scipy.spatial import cKDTree
import pandas as pd
from pdb import set_trace
import importlib

# In-house modules
srcFolder = '#srcDir'
sys.path.append(os.path.join(srcFolder, 'PythonUtilities'))
from FemMesh2D import FemMesh2D as fm2
import MorphoDesignFunctions as mdf
from PyAster import *
from datfile import datfile
import AsterStudyUtilities as asus
from LineMeasures import *
# }}}
```

After loading the necessary libraries, the script reads and defines the default and external parameters.

```
# Parameters {{{
# Default {{{
workDir = '#workDir'
parFile = '#parFile'
surfName         = "SURFS"
group_DESPO_WEAR = "zeroWear"
NUME_RESU_GROWTH = 5
exportFile = os.path.join(workDir, 'EXPORT/Study.export')
esc_name = "esc"
mai_name = "mai"
# Load default parameter file
with open(os.path.join(srcFolder, 'config.json')) as fle:
    defaultParameters = json.load(fle)
locals().update(defaultParameters)
# }}}
# External {{{
with open(parFile, 'r') as inFile:
    params = json.load(inFile)
# Get each block of parameters {{{
c_a_unit        = params["c_a_unit"]
fileNames.update(params.get("fileNames", {}))
modelParams     = params["model"]
meshParams      = params["mesh"]
bcParams        = params["boundary_condition"]
# }}}
# File name parameters {{{
locals().update(fileNames)

resuName, resuExt = os.path.splitext(workDir + resuFolder +
    resuFile)
presResuName, presResuExt = os.path.splitext(workDir + resuFolder
    +
        presResuFile)
# }}}
# Model parameters {{{
# Material properties {{{
materialParams = modelParams["materials"]
E  = materialParams.get("E", 1.0e+3)
nu = materialParams.get("nu", 0.3)
E_esc = materialParams.get("E_esc", E)
E_mai = materialParams.get("E_mai", E)
nu_esc = materialParams.get("nu_esc", nu)
nu_mai = materialParams.get("nu_mai", nu)
modelisation = materialParams["MODELISATION"]
# }}}
# Growth parameters {{{

growthParams      = modelParams["growthParams"]
maxDis            = growthParams["maxDis"]
tauLim            = growthParams["tauLim"]
sigLim              = growthParams["sigLim"]
try:
    velmin = growthParams["velmin"]
    velmax = growthParams["velmax"]
except KeyError:
    vel = growthParams["vel"]
    velmin = vel
    velmax = vel
Sr                  = growthParams["Sr"] # Controls the isotropic
    growth
Gr                  = growthParams["Gr"] # Growth strength
Wr                  = growthParams["Wr"] # Wear strength
Gr_esc_factor       = growthParams.get("Gr_esc_factor", 1.0)
Gr_mai_factor       = growthParams.get("Gr_mai_factor", 1.0)
Wr_esc_factor       = growthParams.get("Wr_esc_factor", 1.0)
Wr_mai_factor       = growthParams.get("Wr_mai_factor", 1.0)
max_jeu             = growthParams.get("max_jeu", 1000000000000.0)
a_f                 = growthParams["a_f"]

alpha = Gr
kappa = Wr

smoothDistanceFactor = growthParams.get("smoothDistanceFactor",
    0.2)
smoothDis = np.log(1.0/smoothDistanceFactor - 1.0)/(
    smoothDistanceFactor*maxDis)
# }}}
# Time parameters {{{
timeParams = modelParams["timeParams"]
deltaT           = timeParams["deltaT"]
final            = timeParams["final"]
prints           = timeParams["prints"]
# }}}
# Algorithm parameters {{{
algoParams = modelParams["algoParams"]
a_dot_ref  = algoParams["a_dot_ref"]
rate_a_dot = algoParams["rate_a_dot"]
xlimFactor = algoParams["xlimFactor"]
minSteps   = algoParams["minSteps"]
maxSteps   = algoParams["maxSteps"]
maxNewton  = algoParams["maxNewton"]
maxItes    = algoParams.get("maxItes", 1000000)
# }}}
# }}}
# Mesh parameters {{{
lcMin            = meshParams["lcMin"]
lcLimFactor      = meshParams["lcLimFactor"]
lcLim = lcLimFactor*lcMin
newgroups = meshParams.get("newgroups", [])
# }}}
# Boundary conditions {{{
esc_groups = bcParams["esc_groups"]
mai_groups = bcParams["mai_groups"]
```

```
group_MAIT = bcParams["GROUP_MA_MAIT"]
group_ESCL = bcParams["GROUP_MA_ESCL"]
contactDisps = bcParams["contactDisps"]
growthDisps = bcParams["growthDisps"]
wearDisp    = bcParams["wearDisp"]
springParams = bcParams.get("springParams", {})
growthFixedPoints = bcParams.get("growthFixedPoints", {})
escFixedPoints = growthFixedPoints.get("esc", {})
maiFixedPoints = growthFixedPoints.get("mai", {})
relocate = bcParams.get("relocate", {})
contactLoad = bcParams["contactLoad"]
# Add deltaT to contactLoad if necessary {{{
for k1 in range(len(contactLoad)):
    cl_i = contactLoad[k1]
    if "FACTOR_FUNCTION" in cl_i:
        # Get FACTOR_FUNCTION
        facFun_i = cl_i["FACTOR_FUNCTION"]
        # Add deltaT to FACTOR_PARAMETERS if necessary
        if "deltaT" in facFun_i:
            contactLoad[k1]["FACTOR_PARAMETERS"]["deltaT"] =
   deltaT
# }}}
# }}}
# }}}
# }}}
```

Then, the model parameters to ensure static equilibrium are predefined as code aster entries.

```
# Prepare parameters {{{
# model, discret and crea_poi dictionaries
model_Fs = []
crea_poi1_Fs = []
discret_Fs = []
for spr_i in springParams:
    crea_poi1_Fs.append(_F(GROUP_MA = spr_i["GROUP_MA"],
                           NOM_GROUP_MA = spr_i["NOM_GROUP_MA"]))
    model_Fs.append(_F(MODELISATION = "2D_DIS_T",
                       PHENOMENE = "MECANIQUE",
                       GROUP_MA = spr_i["NOM_GROUP_MA"]))
    discret_Fs.append(_F(GROUP_MA = spr_i["NOM_GROUP_MA"],
                         CARA = spr_i["CARA"],
                         VALE = spr_i["VALE"]))
# Reference fixed points
group_REF_NODE_ESC_DX = escFixedPoints.get("DX", None)
group_REF_NODE_ESC_DY = escFixedPoints.get("DY", None)
group_REF_NODE_MAI_DX = maiFixedPoints.get("DX", None)
group_REF_NODE_MAI_DY = maiFixedPoints.get("DY", None)
# }}}
```

The fixed boundary where the reactions to the external load appear is defined; this is later used to monitor the simulation.

```
# Get dy fixed boundary in contactDisps {{{
```

```
contDisp_DDL_IMPO = contactDisps["DDL_IMPO"]
if isinstance(contDisp_DDL_IMPO, dict):
    contDisp_DDL_IMPO = [contDisp_DDL_IMPO]
gr_dy0 = []
for bc_i in contDisp_DDL_IMPO:
    if ("DY" in bc_i) or ("LIAISON" in bc_i):
        # Get groups of nodes
        gr_noeud = bc_i.get("GROUP_NO", [])
        if isinstance(gr_noeud, str):
            gr_noeud = [gr_noeud]
        # Get groups of elements
        gr_maille = bc_i.get("GROUP_MA", [])
        if isinstance(gr_maille, str):
            gr_maille = [gr_maille]
        # Add groups
        gr_dy0 += gr_maille + gr_noeud
# }}}
```

The code aster block is initialised, and the meshes are read and assembled.

```
# Initialisation {{{
DEBUT(LANG = 'FR', PAR_LOT = 'NON', IMPR_MACRO = 'NON')
U = c_a_unit
# }}}
# Read and set up mesh union {{{
# Read each mesh (esc and mai)
escFile = os.path.join(workDir, esc_name + '.' + geomFMT)
maiFile = os.path.join(workDir, mai_name + '.' + geomFMT)
meshT = READ_MESH(MESH_NAME = escFile, FORMAT = 'IDEAS', UNITE = U
    )
meshB = READ_MESH(MESH_NAME = maiFile, FORMAT = 'IDEAS', UNITE = U
    )
# Mesh union
meshU = ASSE_MAILLAGE(MAILLAGE_1 = meshT,
                      MAILLAGE_2 = meshB,
                      OPERATION = 'SUPERPOSE')
# Group of elements for all the mesh
DEFI_GROUP(MAILLAGE = meshU,
           CREA_GROUP_MA = _F(NOM = surfName,
                              UNION = [esc_name, mai_name]),
           reuse = meshU)
# Define contact group if necessary
if len(group_ESCL) > 1:
    DEFI_GROUP(MAILLAGE = meshU,
               CREA_GROUP_MA = _F(NOM = "ESC_CON",
                                  UNION = group_ESCL),
               reuse = meshU)
    group_ESCL = ["ESC_CON"]
if len(group_MAIT) > 1:
    DEFI_GROUP(MAILLAGE = meshU,
               CREA_GROUP_MA = _F(NOM = "MAI_CON",
                                  UNION = group_MAIT),
               reuse = meshU)
    group_MAIT = ["MAI_CON"]
# }}}
```

The mesh is then modified for the Mortar contact algorithm. Additionally, it can comprise springs to ensure equilibrium, particularly under non centred loads.

```
# Set up Mortar mesh {{{
# Orient contact elements
MODI_MAILLAGE(reuse = meshU,
              MAILLAGE = meshU,
              ORIE_PEAU_2D = _F(GROUP_MA = (group_ESCL +
    group_MAIT)))
# Set up Mortar mesh
if springParams == {}:
    mesh = CREA_MAILLAGE(MAILLAGE = meshU,
                         DECOUPE_LAC = _F(GROUP_MA_ESCL = (
    group_ESCL)))
else:
    mesh_Poi = CREA_MAILLAGE(CREA_POI1 = crea_poi1_Fs,
                             MAILLAGE = meshU)
    mesh = CREA_MAILLAGE(MAILLAGE = mesh_Poi,
                         DECOUPE_LAC = _F(GROUP_MA_ESCL = (
    group_ESCL)))
# Ordered contact node set
DEFI_GROUP(DETR_GROUP_NO = _F(NOM = (group_ESCL + group_MAIT)),
           CREA_GROUP_NO = (_F(GROUP_MA = (group_MAIT),
                               NOM = group_MAIT,
                               OPTION = 'NOEUD_ORDO'),
                            _F(GROUP_MA = (group_ESCL),
                               NOM = group_ESCL,
                               OPTION = 'NOEUD_ORDO')),
           MAILLAGE = mesh,
           reuse = mesh)
mail_py = MAIL_PY()
mail_py.FromAster(mesh)
esc_con_nods = mail_py.gno[group_ESCL[0]]
mai_con_nods = mail_py.gno[group_MAIT[0]]
# }}}
```

Lastly, necessary element and node groups are created.

```
# Define new groups {{{
for newgroup in newgroups:
    DEFI_GROUP(reuse = mesh,
               MAILLAGE = mesh,
               **newgroup)
# Order contour groups
for groupName in esc_groups + mai_groups:
    DEFI_GROUP(DETR_GROUP_NO = _F(NOM = groupName),
               CREA_GROUP_NO = _F(GROUP_MA = groupName,
                                  NOM = groupName,
                                  OPTION = 'NOEUD_ORDO'),
               MAILLAGE = mesh,
               reuse = mesh)
# }}}
```

Once the mesh is established, the material and mechanical model are defined.

```
# Order contour groups
for groupName in esc_groups + mai_groups:
    DEFI_GROUP(DETR_GROUP_NO = _F(NOM = groupName),
               CREA_GROUP_NO = _F(GROUP_MA = groupName,
                                  NOM = groupName,
                                  OPTION = 'NOEUD_ORDO'),
               MAILLAGE = mesh,
               reuse = mesh)
# }}}
# Define materials {{{
esc_mat = DEFI_MATERIAU(ELAS = _F(E = E_esc,
                                  NU = nu_esc,
                                  RHO = 1.0))
mai_mat = DEFI_MATERIAU(ELAS = _F(E = E_mai,
                                  NU = nu_mai,
                                  RHO = 1.0))
matwe = DEFI_MATERIAU(ELAS = _F(E = 1.0,
                                RHO = 1.0,
                                NU = 0.49))
# }}}
# Set up model and material fields {{{
model_Fs.append(_F(MODELISATION = (modelisation, ),
                   PHENOMENE = 'MECANIQUE',
                   TOUT = 'OUI'))
# Code aster model
mode = AFFE_MODELE(AFFE = model_Fs,
                   MAILLAGE = mesh)
# Material fields
matf = AFFE_MATERIAU(AFFE = (_F(MATER = esc_mat, GROUP_MA = "esc")
    ,
                             _F(MATER = mai_mat, GROUP_MA = "mai")
    ),
                     MODELE = mode)
matwf = AFFE_MATERIAU(AFFE = _F(MATER = (matwe, ),
                                TOUT = 'OUI'),
                      MODELE = mode)
if len(discret_Fs) > 0:
    springs = AFFE_CARA_ELEM(DISCRET_2D = discret_Fs,
                             MODELE = mode)
else:
    springs = None
# }}}
```

For the computation of the stress state due to the external force and the contact boundary, an adaptive load step is defined using the minimum and maximum number of load steps defined in `params.json`.

```
# Set increment and time lists {{{
t_one_l = DEFI_FONCTION(NOM_PARA='INST',
                        VALE=(0.0, 0.0, 1.0, 1.0),
                        PROL_DROITE = 'CONSTANT')
listapri = DEFI_LIST_REEL(DEBUT = 0.0,
                              INTERVALLE = (_F(JUSQU_A = 1.0,
                                               NOMBRE = minSteps)))
defsteps = DEFI_LIST_INST(DEFI_LIST = _F(LIST_INST = listapri),
                          ECHEC = _F(EVENEMENT = 'ERREUR',
                                     SUBD_METHODE = 'AUTO',
                                     SUBD_PAS_MINI = 1.0/maxSteps)
   )

# }}}
```

Next, the fixed displacements and the contact boundaries are defined.

```
# Set up loads and contact {{{
contDisp = AFFE_CHAR_MECA(MODELE = mode, **contactDisps)
growDisp = AFFE_CHAR_MECA(MODELE = mode, **growthDisps)
# Contact
contactO = DEFI_CONTACT(FORMULATION = 'CONTINUE',
                        MODELE = mode,
                        LISSAGE = 'OUI',
                        ALGO_RESO_GEOM = 'NEWTON',
                        ALGO_RESO_CONT = 'NEWTON',
                        ZONE=_F(GROUP_MA_MAIT = group_MAIT,
                                GROUP_MA_ESCL = group_ESCL,
                                ALGO_CONT = 'LAC',
                                TYPE_APPA = "ROBUSTE",
                                CONTACT_INIT = 'OUI'))
contactI = DEFI_CONTACT(FORMULATION = 'CONTINUE',
                        MODELE = mode,
                        LISSAGE = 'OUI',
                        ALGO_RESO_GEOM = 'NEWTON',
                        ALGO_RESO_CONT = 'NEWTON',
                        ZONE=_F(GROUP_MA_MAIT = group_MAIT,
                                GROUP_MA_ESCL = group_ESCL,
                                ALGO_CONT = 'LAC',
                                TYPE_APPA = "ROBUSTE",
                                CONTACT_INIT = 'INTERPENETRE'))
# }}}
```

Before the initiation of the growth loop, the time and iteration variables are initialised. This requires the creation of the report files or reading from them if the execution is the continuation after a remeshing step.

```
# Growth loop {{{
# Set up time parameters {{{
# Set initial time {{{
try:
    # Get initial time {{{
    report = datfile(workDir + repoFile).datablocks['0'].variables
    t0 = report["Time"][-1] + deltaT
    contArea = report["cont_area"][-1]
    vel_i = report["vel_i"][-1]
    deltaA_i = report["deltaA_i"][-1]
    alpha_i = report["alpha_i"][-1]
    # Load curve data frame
```

```python
    curve_df = pd.read_csv(workDir + curveDataFile)
    # }}}
    # Get iteration and printout state {{{
    meshSize = datfile(workDir + meshSizeFile).datablocks['0'].variables
    ite = meshSize["ite"][-1] + 1
    printout = meshSize["printout"][-1] + 1
    # }}}
    firstRun = False
except FileNotFoundError:
    # Set initial time, printout and iteration to 0 if repoFile does
    # not exist, i.e., the simulation has just started
    t0 = 0.0
    printout = 0
    ite = 0
    vel_i = (velmax + velmin)/2.0
    deltaA_i = 0.0
    contArea = 0.0
    #alpha_i = alpha/(1.0 + deltaT*rate_a_dot*(2.0*10.0/(1.0 + 10.0) - 1.0)) # Related to a_dot = 10.0, when dadt = 0.0
    alpha_i = alpha
    # Start files {{{
    with open(workDir + meshSizeFile, 'w') as fle:
        fle.write('TITLE = "Report of numerical variables"\n')
        fle.write('TIME = None\n')
        fle.write('VARIABLES = "ite", "printout", "xlim0", "xlim1", "max_length_ratio", "F_reac", "minQua",\n')
    with open(workDir + repoFile, 'w') as fle:
        fle.write('TITLE = "Report"\n')
        fle.write('TIME = None\n')
        fle.write('VARIABLES = "Time", "p_c_max", "cont_area", "Q_p", "Q_a", "alpha_i", "eta_i", "a_i/a_f", "vel_i", "deltaA_i", "mea_kesc", "mea_kmai", "mea_kdif", "mea_krelDif",\n')
    with open(workDir + presResuFile, 'w') as fle:
        fle.write('TITLE = "Contact pressure"\n')
    # }}}
    firstRun = True
    # Initialisation of curve data frame
    curve_df = pd.DataFrame(columns = ["Time", "x", "yesc", "ymai"])
# }}}
# Set time discretisation {{{
growthTime = np.linspace(t0, final, int(final/deltaT) - ite + 1)
printeach = int((final/deltaT)/prints)
effKappa = kappa*deltaT
# }}}
# Initialisation of variables {{{
contEleLength = 0.0
if printout >= printeach:
    printout = 0
prevResu = None
# }}}
# }}}
minContElementLength = lcMin
count_ite = 0
```

The growth loop starts testing the mesh to check whether a remesh is necessary.

```
for ti in growthTime:
    count_ite += 1
    if count_ite > maxItes:
        break
    # Initialisation of detr_lis
    detr_lis = []
    # Test the mesh {{{
    # Test maximum contact element length
    if contEleLength > lcMin*lcLimFactor:
        break
    # Test minimum contact element length
    if minContElementLength < lcMin/3.0/1.5:
        break
    # }}}
```

If the test is passed, the external load is applied using an in-house macro that allows the application of a time-dependent distributed load.

```
    # Set loads {{{
    # Set load_y {{{
    load_y = APPLY_2D_TOTAL_FORCE_CONTOUR(
            MESH = mesh,
            MODELE = mode,
            TIME = ti,
            TOTAL_FORCE_CONTOUR = contactLoad)
    detr_lis.append(_F(NOM = load_y))
    # }}}
    # Set EXCIT for the contact analysis {{{
    excitCont = [_F(CHARGE = load_y, FONC_MULT = t_one_l),
                 _F(CHARGE = contDisp)]
    # }}}
    # }}}
```

Then, the contact problem is solved with an in-house macro that ensures an initial interpenetration which is necessary for code aster to converge.

```
    # Solve contact problem {{{
    # Execution of the solver {{{
    solvPara = {
                "MESH" : mesh,
                "DIR_X" : 0.0,
                "DIR_Y" : -1.0,
                "CARA_ELEM" : springs,
                "RELOC_H" : lcMin/3.0,
                "RELOC_GROUP_MA" : esc_name,
                "GROUP_MA_MAIT" : group_MAIT,
                "GROUP_MA_ESCL" : group_ESCL,
                "CHAM_MATER" : matf,
                "maxNewton" : maxNewton,
                "CONTACT" : contactI,
                "EXCIT" : excitCont,
                "LIST_INST" : defsteps,
                "PREVRESU" : prevResu,
                "MODELE" : mode
            }
    # Try to solve with contact interpenetrate and previous result
.
    try:
        resu = SOLVE_CONTACT_BY_FORCE(**solvPara)
    except:
        # Solve with full contact activated and no previous result
        DETRUIRE(CONCEPT = (_F(NOM = resu)))
        solvPara["CONTACT"] = contact0
        solvPara["PREVRESU"] = None
        solvPara["RELOC_H"] = 0.0
        resu = SOLVE_CONTACT_BY_FORCE(**solvPara)
    detr_lis.append(_F(NOM = resu))
    # }}}
    # Set previous result and interpenetrate contact for the next
iteration
    if not prevResu == None:
        DETRUIRE(CONCEPT = (_F(NOM = prevResu)))
    prevResu = COPIER(CONCEPT = resu)
    # }}}
```

The stress state and the reaction force is computed.

```
    # Get sigma and reaction force {{{
    # Get stress
    sigma = CREA_CHAMP(TYPE_CHAM = 'NOEU_SIEF_R',
                       OPERATION = 'EXTR',
                       RESULTAT = resu,
                       NOM_CHAM = 'SIEF_NOEU',
                       INST = 1.0)
    detr_lis.append(_F(NOM = sigma))
    resu = CALC_CHAMP(reuse = resu,
                      RESULTAT = resu,
                      INST = 1.0,
                      FORCE = 'REAC_NODA')
    reacForc = POST_RELEVE_T(ACTION = _F(OPERATION = 'EXTRACTION',
                                         INTITULE = 'Reaction
force',
                                         RESULTAT = resu,
                                         NOM_CHAM = 'REAC_NODA',
                                         GROUP_NO = gr_dy0,
                                         RESULTANTE = ('DY'),
                                         REPERE = 'GLOBAL',
                                         MOYE_NOEUD = 'OUI'))
    detr_lis.append(_F(NOM = reacForc))
    F_reac = reacForc.EXTR_TABLE().rows[-1]["DY"]
    # }}}
```

Then, the contact pressure and measures related to the contact boundary, such as curvature and conformity, are computed.

```
    # Get contact pressure and element length {{{
    # Get contact pressure
    tbcont = CONTACT_PRESSURE_ABSC_CURV(RESU = resu,
                                        MESH = mesh,
                                        GROUP_MA = group_ESCL,
                                        MODELE = mode,
                                        INST = 1.0)
    detr_lis.append(_F(NOM = tbcont))
    l_c = np.array(tbcont.EXTR_TABLE().values()['ABSC_CURV'])
    try:
        p_c = np.array(tbcont.EXTR_TABLE().values()['LAGS_C'])
    except:
        p_c = np.array(tbcont.EXTR_TABLE().values()['X1'])
    maxp_c = max(abs(p_c))
    # Get contact area
    tb_co_ar = CONTACT_AREA_CONT_ELEM(RESU = resu,
                                      MESH = mesh,
                                      INST = 1.0)
    detr_lis.append(_F(NOM = tb_co_ar))
    contArea = tb_co_ar.EXTR_TABLE().values()["CONT_AREA"][0]
    lastIte = contArea - a_f > lcMin
    contEleLength = tb_co_ar.EXTR_TABLE().values()["MAX_ELEM_LENGTH"][0]
    minContElementLength = tb_co_ar.EXTR_TABLE().values()["MIN_ELEM_LENGTH"][0]
    activeNodes = tb_co_ar.EXTR_TABLE().values()["ACTIVE_NODES"]
    activeNodes = list(filter((None).__ne__, activeNodes))
    activeNodes = [int(val) for val in activeNodes]
    activeNodeNames = [mail_py.correspondance_noeuds[val] for val in activeNodes]
    DEFI_GROUP(MAILLAGE = mesh,
               DETR_GROUP_NO = _F(NOM = "__ACTNOD"),
               CREA_GROUP_NO = [_F(NOM = "__ACTNOD",
                                   NOEUD = activeNodeNames)],
               reuse = mesh)
    xlim = tb_co_ar.EXTR_TABLE().values()["XLIM"]
    xlimLeft  = xlim[0]
    xlimRight = xlim[1]
    meanXlim = (xlim[0] + xlim[1])/2.0
    xlimWidth = (xlim[1] - xlim[0])*xlimFactor/2.0
    xlim = [meanXlim - xlimWidth, meanXlim + xlimWidth]
    # Measure curvature and conformity {{{
    coords = mail_py.cn
    # slave active coordinates
    esc_coords = []
    for nod_id in esc_con_nods:
        esc_coords.append([coords[nod_id, 0],
                           coords[nod_id, 1]])
    esc_coords = np.array(esc_coords)
    # master active coordinates
    mai_coords = []
    for nod_id in mai_con_nods:
        mai_coords.append([coords[nod_id, 0],
                           coords[nod_id, 1]])
    mai_coords = np.array(mai_coords)
    # Make discrete lines
    escLine = DiscreteLine(esc_coords[:, 0],
            values = {"yesc" : esc_coords[:, 1]})
    maiLine = DiscreteLine(mai_coords[:, 0],
            values = {"ymai" : mai_coords[:, 1]})
    refLine = np.linspace(xlimLeft, xlimRight, int(3.0*(xlimRight
   - xlimLeft)/(lcMin)))
    refLine = DiscreteLine(refLine)
    # Map slave and master lines into reference line
    refLine.MapValues(escLine, ["yesc"])
    refLine.MapValues(maiLine, ["ymai"])
    esc_active_coords = np.column_stack((refLine.x_array,
        refLine.values["yesc"]))
    mai_active_coords = np.column_stack((refLine.x_array,
        refLine.values["ymai"]))
    # Compute curvatures
    kesc = refLine.Curvature(ykey = "yesc")
    kmai = refLine.Curvature(ykey = "ymai")
    refLine.add_value("kesc", kesc)
    refLine.add_value("kmai", kmai)
    kdif = kesc - kmai
    krelDif = abs(kdif)/(abs(kesc) + abs(kmai))
    krelDif = 0.5 + kesc*kmai/(kesc**2.0 + kmai**2.0)
    refLine.add_value("kdif", kdif)
    refLine.add_value("krelDif", krelDif)
    norm_kesc = refLine.LpNorm("kesc")
    norm_kmai = refLine.LpNorm("kmai")
    norm_kdif = refLine.LpNorm("kdif")
    norm_krelDif = refLine.LpNorm("krelDif")
    # Compute length
    refLine.add_value("unit", np.ones(refLine.numPoints))
    norm_unit = refLine.LpNorm("unit")
    # Compute measures
    mea_kesc = norm_kesc/norm_unit
    mea_kmai = norm_kmai/norm_unit
    mea_kdif = norm_kdif/norm_unit
    mea_krelDif = norm_krelDif/norm_unit
    # }}}
    # Pressure quality
    p_o = abs(F_reac)/contArea
    p_r = np.zeros(len(p_c))
    cont_nods = list(mail_py.gno[group_ESCL[0]])
    for act_nod in activeNodes:
        index = cont_nods.index(act_nod)
        p_r[index] = p_o
    p_diff = p_r - p_c
    Q_p = 1.0 - np.linalg.norm(p_diff)/np.linalg.norm(p_c)
    # Update deltaA_i
    try:
        deltaA_i = contArea - contArea_prev
    except NameError:
        pass
    contArea_prev = contArea
    # Update alpha_i
    dadt_i = deltaA_i/deltaT
    if abs(dadt_i) > 1.0e-9:
        a_dot = abs(a_dot_ref/dadt_i)
    else:
        #a_dot = 1.0
        a_dot = 10.0
    alpha_i = alpha_i*(1.0 +
            deltaT*rate_a_dot*(2.0*a_dot/(1.0 + a_dot) - 1.0))
    # }}}
```

With the stress state, the growth force and the growth itself can be computed. This requires the use of an in-house macro (MORPHOGENESIS_GROWTH_BENEATH_CONTOUR) that computes the growth force considering D_g. Once the force is computed, the displacement field is computed with another in-house macro (COMPUTE_MESH_DISPLACEMENT_-FROM_GROWTH_FORCE).

```
        # Compute morphogenesis growth {{{
        # Compute growth fields
        comb_FFF = []
        if abs(Gr) > 0.0:
            # Slave growth {{{
            if Gr_esc_factor > 0.0:
                # Compute growth force
                grwEscFi = MORPHOGENESIS_GROWTH_BENEATH_CONTOUR(
                        RESU = resu,
                        INST = 1.0,
                        GROUP_MA = esc_name,
                        GROUP_MA_CONT = "__ACTNOD",
                        MODELE = mode,
                        MODELISATION = modelisation,
                        MESH = mesh,
                        GFUNC = "mdf.Sgrowth",
                        FUNC_PARAMS = [{"alpha" : Gr_esc_factor*
    alpha_i*deltaT,
                                        "shrlim" : tauLim,
                                        "hydlim": sigLim,
                                        "vel" : vel_i}],
                        SMOOTH_PARAMS = [{"smoothDis" : smoothDis,
                                          "maxDis" : maxDis}],
                        GEOMETRIE = "DEFORMEE",
                        )
                detr_lis.append(_F(NOM = grwEscFi))
                growth1 = COMPUTE_MESH_DISPLACEMENT_FROM_GROWTH_FORCE(
                        GRW_TEN = grwEscFi,
                        GROUP_MA = esc_name,
                        MODELE = mode,
                        CHAM_MATER = matf,
                        REF_FIXED_DX = group_REF_NODE_ESC_DX,
                        REF_FIXED_DY = group_REF_NODE_ESC_DY,
                        EXCIT = (_F(CHARGE = growDisp)),
                        CARA_ELEM = springs,
                        )
                detr_lis.append(_F(NOM = growth1))
                comb_FFF.append(_F(CHAM_GD = growth1, COEF_R = 1.0))
            # }}}
            # Master growth {{{
            if Gr_mai_factor > 0.0:
                # Compute growth force
                grwMaiFi = MORPHOGENESIS_GROWTH_BENEATH_CONTOUR(
                        RESU = resu,
                        INST = 1.0,
                        GROUP_MA = mai_name,
                        GROUP_MA_CONT = "__ACTNOD",
                        MODELE = mode,
                        MODELISATION = modelisation,
                        MESH = mesh,
                        GFUNC = "mdf.Sgrowth",
                        FUNC_PARAMS = [{"alpha" : Gr_mai_factor*alpha_i*deltaT,
                                        "shrlim" : tauLim,
                                        "hydlim": sigLim,
                                        "vel" : vel_i}],
                        SMOOTH_PARAMS = [{"smoothDis" : smoothDis,
                                          "maxDis" : maxDis}],
                        GEOMETRIE = "DEFORMEE",
                        )
                detr_lis.append(_F(NOM = grwMaiFi))
                growth2 = COMPUTE_MESH_DISPLACEMENT_FROM_GROWTH_FORCE(
                        GRW_TEN = grwMaiFi,
                        GROUP_MA = mai_name,
                        MODELE = mode,
                        CHAM_MATER = matf,
                        REF_FIXED_DX = group_REF_NODE_MAI_DX,
                        REF_FIXED_DY = group_REF_NODE_MAI_DY,
                        EXCIT = (_F(CHARGE = growDisp)),
                        CARA_ELEM = springs,
                        )
                detr_lis.append(_F(NOM = growth2))
                comb_FFF.append(_F(CHAM_GD = growth2, COEF_R = 1.0))
            # }}}
            # Compute total displacement
            growth = CREA_CHAMP(OPERATION = 'COMB',
                                TYPE_CHAM = 'NOEU_DEPL_R',
                                COMB = comb_FFF)
            detr_lis.append(_F(NOM = growth))
        # }}}
```

Before applying the displacements, the results of the current iteration are written into a vtk file for visualisation in Paraview.

```
    # Save field results {{{
    # Write curves to pandas
```

```
xref = refLine.x_array
yesc = refLine.values["yesc"]
ymai = refLine.values["ymai"]
data = {"Time" : ti,
        "x" : xref,
        "yesc" : yesc,
        "ymai" : ymai}
curve_df = curve_df.append(data, ignore_index = True)
curve_df.to_csv(workDir + curveDataFile, index = False)
# Save contact pressure
with open(workDir + presResuFile, 'a') as fle:
    fle.write('TIME = {:5.6f}, N = {},\n'.format(ti, len(p_c)))
    fle.write('VARIABLES =  "l_c [mm]", "p_c [GPa]", "p_r [GPa]",\n')
    for l_i, p_i, p_ri in np.column_stack((l_c, p_c, p_r)):
        fle.write('{:1.5e}, {:1.5e}, {:1.5e},\n'.format(l_i, p_i, p_ri))
if printout == 0 or lastIte:
    # Make growth stress field
    if abs(Gr) > 0.0:
        asse_f = []
        if Gr_esc_factor > 0.0:
            asse_f.append(_F(GROUP_MA = esc_name,
                             CHAM_GD = grwEscFi,
                             NOM_CMP = "SIXX",
                             NOM_CMP_RESU = 'X11'))
        if Gr_mai_factor > 0.0:
            asse_f.append(_F(GROUP_MA = mai_name,
                             CHAM_GD = grwMaiFi,
                             NOM_CMP = "SIXX",
                             NOM_CMP_RESU = 'X11'))
        else:
            grwMaiFi = CREA_CHAMP(OPERATION = "AFFE",
                                  TYPE_CHAM = "NOEU_SIEF_R",
                                  MODELE = mode,
                                  AFFE = [_F(GROUP_MA = mai_name,
                                             NOM_CMP = ["SIXX"],
                                             VALE = [0.0])])
            detr_lis.append(_F(NOM = grwMaiFi))
            asse_f.append(_F(GROUP_MA = mai_name,
                             CHAM_GD = grwMaiFi,
                             NOM_CMP = "SIXX",
                             NOM_CMP_RESU = 'X11'))
        grwAll = CREA_CHAMP(OPERATION = "ASSE",
                            TYPE_CHAM = "NOEU_NEUT_R",
                            MODELE = mode,
                            ASSE = asse_f)
        detr_lis.append(_F(NOM = grwAll))
        node_fields_i = [_F(CHAM_NO = grwAll,
                            NOM = "growthData",
                            NOM_CMP = "X11"),
                         _F(CHAM_NO = growth,
                            NOM = "growth_DX",
                            NOM_CMP = "DX"),
                         _F(CHAM_NO = growth,
                            NOM = "growth_DY",
                            NOM_CMP = "DY")
                         ]
    else:
        node_fields_i = None
    # Save to vtk
    SAVE_RESULTS_VTK(FILE_NAME = workDir + resuFolder + '/resu' + str(ite) + '.vtk',
                     MESH = mesh,
                     INST = 1.0,
                     GROUP_MA_SURF = surfName,
                     NUME_RESU_GROWTH = NUME_RESU_GROWTH,
                     CHAMPS = ('disp', 'STRESS2D', 'tauMis',
                         'sigHyd'),
                     NODE_FIELDS = node_fields_i,
                     RESU = resu)
    # Save mesh
    DEFI_FICHIER(ACTION = 'ASSOCIER',
                 FICHIER = workDir + resuFolder + '/mesh' + str(ite) + '.mail',
                 UNITE = U)
    IMPR_RESU(UNITE = U,
              MODELE = mode,
              RESU = _F(MAILLAGE = mesh),
              FORMAT = 'ASTER')
    DEFI_FICHIER(ACTION = 'LIBERER', UNITE = U)
# }}}
# Apply displacements {{{
if not firstRun:
    # To mesh
    if abs(Gr) > 0.0:
        MODI_MAILLAGE(reuse = mesh,
                      MAILLAGE = mesh,
                      DEFORME = (_F(OPTION = 'TRAN',
                                    DEPL = growth)))
# }}}
```

Next, the mesh quality is computed and the current mesh information is saved for remeshing purposes.

```
    # Compute element quality {{{
    mail_py = MAIL_PY()
    mail_py.FromAster(mesh)
    meshFEM2D = fm2.MeshFromMail_Py(mail_py)
    meshQuality = meshFEM2D.Quality()
    minQua = meshQuality["minQua"]
    meanQua = meshQuality["meanQua"]
    # Save mesh for reconstruction
    DEFI_FICHIER(ACTION = 'ASSOCIER',
                  FICHIER = workDir + recoFile,
                  UNITE = U)
    IMPR_RESU(UNITE = U,
              MODELE = mode,
              RESU = _F(MAILLAGE = mesh),
              FORMAT = 'ASTER')
    DEFI_FICHIER(ACTION = 'LIBERER', UNITE = U)
    # }}}
```

Before finishing the loop iteration, useful values are reported, such as curvature and maximum contact pressure, and the code aster objects of the iteration are deleted. Finally, after the growth loop, the code aster block ends.

```
    # Report iteration and update variables {{{
    # Write reports
    with open(workDir + repoFile, 'a') as fle:
        inFormat = (ti, maxp_c, contArea, Q_p, Q_a,
                alpha_i, eta_i, contArea/a_f, vel_i, deltaA_i,
                mea_kesc, mea_kmai, mea_kdif, mea_krelDif)
        txt = len(inFormat)*'{:1.5e}, '
        txt = txt[:-1] + '\n'
        fle.write(txt.format(*inFormat))
    with open(workDir + meshSizeFile, 'a') as fle:
        inFormat_d = (ite, printout)
        inFormat_e = (xlim[0], xlim[1], contEleLength/(lcMin*
   lcLimFactor),
                F_reac, minQua)
        txt = len(inFormat_d)*'{:10d}, ' + len(inFormat_e)*'{:1.5e
   }, '
        txt = txt[:-1] + '\n'
        inFormat = inFormat_d + inFormat_e
        fle.write(txt.format(*inFormat))
    # Update iteration control variables
    ite += 1
    printout += 1
    if printout >= printeach:
        printout = 0
    # }}}
    # Test initial run or last {{{
    if firstRun or lastIte:
        break
    # }}}
    # Destroy aster objects {{{
    DETRUIRE(CONCEPT = detr_lis)
    # Delete glob.*
    asus.DeleteTmpFiles(exportFile, ["glob.*", "../proc.0/glob.*"
   ])
    # }}}
# }}}
FIN()
# }}}
```