```
import Fem
import math
import time
import ObjectsFem
import numpy as np
import Part as Part
import FreeCAD as App
from numba import jit
import FreeCADGui as Gui
import scipy.sparse as scsp
import matplotlib.pyplot as plt
from femtools import membertools
from femmesh import meshsetsgetter
from femmesh import meshtools as mt
from numba import types, __version__
from sksparse.cholmod import cholesky
from matplotlib.widgets import Button
from feminout import importToolsFem as itf
from femsolver.writerbase import FemInputWriter
```

```
np.set_printoptions(precision=5, linewidth=300)


def prn_upd(*args):
    for obj in args:
        print(str(obj), end='')
    print('\n')
    Gui.updateGui()


def setUpAnalysis():
    doc = App.ActiveDocument
    mesh = doc.getObject("FEMMeshGmsh").FemMesh
    if mesh == None:
        prn_upd("No Gmsh object. Please create one first")
        raise SystemExit()
    analysis = doc.getObject("Analysis")
    if analysis == None:
        prn_upd("No Analysis object. Please create one first")
        raise SystemExit()
    # purge result objects
    for obj in App.ActiveDocument.Objects:
        name = obj.Name[:11]
        if name in ['MechanicalR', 'Result_Mesh']:
            doc.removeObject(obj.Name)
    doc.recompute()

    return doc, mesh, analysis


def setUpInput(doc, mesh, analysis):
    analysis = doc.getObject("Analysis")
    solver = doc.getObject("SolverCcxTools")
    docmesh = doc.getObject("FEMMeshGmsh")
    femmesh = docmesh.FemMesh
    member = membertools.AnalysisMember(analysis)
    # determine elements connected to a node using FC API
    fet = mt.get_femelement_table(mesh)
    # fet is dictionary: { elementid : [ nodeid, nodeid, ... , nodeid ] }
    net = mt.get_femnodes_ele_table(mesh.Nodes, fet)
    # net is dictionary: {nodeID : [[eleID, binary node position], [], ...], nodeID : [[], [], ...], ...}
    # node0 has binary node position 2^0 = 1, node1 = 2^1 = 2, ..., node10 = 2^10 = 1024

    # create connectivity array elNodes for mapping local node number -> global node number
    elNodes = np.array([mesh.getElementNodes(el) for el in mesh.Volumes])  # elNodes[elementIndex] = [node1,...,Node10]
    elo = dict(zip(mesh.Volumes, range(len(mesh.Volumes))))  # elo : {elementNumber : elementIndex}
    ole = dict(zip(range(len(mesh.Volumes)), mesh.Volumes))  # ole : {elementIndex : elementNumber}

    # create nodal coordinate array nocoord for node number -> (x,y,z)
    ncv = list(mesh.Nodes.values())
    nocoord = np.asarray([[v.x, v.y, v.z] for v in ncv])  # nocoord[nodeIndex] = [x-coord, y-coord, z-coord]

    # get access to element sets: meshdatagetter.mat_geo_sets

    # for index, el in enumerate(elNodes):
    #     if all(node in el for node in [6, 7, 24]):
    #         print("element: ", index, el, " contains nodes [6, 7, 24]")

    t1 = time.time()
    meshdatagetter = meshsetsgetter.MeshSetsGetter(
        analysis,
        solver,
        docmesh,
        member)
    meshdatagetter.get_mesh_sets()
    t2 = time.time()
    print("querying meshsetsgetter: ", t2 - t1, "seconds\n")

    if len(member.mats_linear) == 1:
        element_sets = [mesh.Volumes]
    else:
        element_sets = [es["FEMElements"] for es in member.mats_linear]

    matCon = {}
    ppEl = {}
    materialbyElement = []

    prn_upd("Number of material objects: ", len(member.mats_linear))

    for obj in App.ActiveDocument.Objects:
        if obj.Name == "BooleanFragments":
            num_BF_els = len(App.ActiveDocument.BooleanFragments.Objects)  # number of primitives in BooleanFragments
            num_Mat_obs = len(member.mats_linear)  # number of material objects
            if num_BF_els != num_Mat_obs:
                prn_upd("Each BooleanFragment primitive needs its own material object")
                raise SystemExit()
            for indp, primobject in enumerate(obj.Objects):  # primobject: primitive in BooleanFragment
                for indm, matobject in enumerate(member.mats_linear):  # matobject: material object
                    # print(matobject['Object'].References[0][0].Name, primobject.Name)
                    if matobject['Object'].References[0][0] == primobject:
                        matCon[indm] = indp  # the boolean primitive the material object refers to

    # print("matCon: ", matCon)
    for indm, matobject in enumerate(member.mats_linear):
        E = float(App.Units.Quantity(matobject['Object'].Material['YoungsModulus']).getValueAs('MPa'))
        Nu = float(matobject['Object'].Material['PoissonRatio'])
        prn_upd("Material Object: ", matobject['Object'].Name, "   E= ", E, "   Nu= ", Nu)
        for el in element_sets[indm]:  # element_sets[indm]: all elements with material indm
            if matCon: ppEl[el] = matCon[indm]  # ppEl[el]: primitive el belongs to
            materialbyElement.append([E, Nu])  # materialbyElement[elementIndex] = [E, Nu]
    materialbyElement = np.asarray(materialbyElement)

    # print("element at index 6 is called: ", ole[6], "and is part of primitive ", ppEl[ole[6]])
    # print("element at index 28 is called: ", ole[28], "and is part of primitive ", ppEl[ole[28]])
    # print("element at index 60 is called: ", ole[60], "and is part of primitive ", ppEl[ole[60]])
    # print("element at index 87 is called: ", ole[87], "and is part of primitive ", ppEl[ole[87]])
    #
    # print("CHECK")
    # print("element ", 120, " has index ", elo[120])

    # set up interface element connectivity
    nodecount = len(nocoord)
    # print("nodecount: ", nodecount)
    interface_elements = []
    tk = []
    tv = []
    ti = []
    NiP = {}
    shiftelements = []

    # print("PRE-SHIFT")
    # print("element 6: ",elNodes[6])
    # print("element 28: ",elNodes[28])
    # print("element 60: ",elNodes[60])
    # print("element 87: ",elNodes[87])
```

```
for obj in App.ActiveDocument.Objects:
    if obj.Name == "BooleanFragments":
        for indp1, primitive1 in enumerate(obj.Objects):  # primitive: n-th primitive in BooleanFragment
            for face1 in primitive1.Shape.Faces:  # face: a face of n-th primitive
                elfaces1 = np.asarray(
                    mesh.getFacesByFace(face1))  # el1faces[primitive face index] = [6-node-face numbers]
                for sixNdFace in elfaces1:  # sixNdFace: 6-node face element
                    nodes = mesh.getElementNodes(sixNdFace)
                    for nd in nodes:
                        if nd not in NiP.keys():  # NiP[node] = Primitive Number
                            NiP[nd] = indp1
                for indp2, primitive2 in enumerate(obj.Objects[indp1 + 1:]):
                    for face2 in primitive2.Shape.Faces:  # face: a face of n+1-th primitive
                        elfaces2 = np.asarray(
                            mesh.getFacesByFace(face2))  # el1faces[primitive face index] = [6-node-face numbers]
                        contact = list(set(elfaces1) & set(elfaces2))  # triangles in contact
                        if contact:
                            npflist = []  # contact nodes on primitive face
                            for sixNdFace in elfaces1:  # sixNdFace: 6-node-face element in face
                                nodes = mesh.getElementNodes(sixNdFace)
                                npflist.append(nodes)  # add nodes of sixNdFace to primitive contact face
                            cn = list(set([node for npf in npflist for node in
                                           npf]))  # remove duplicates
                            cn.sort()  # sort nodes
                            cn1_new = list(range(nodecount + 1, nodecount + len(cn) + 1))  # new nodes on face n
                            cn2_new = list(range(nodecount + len(cn) + 1,
                                                 nodecount + 2 * len(cn) + 1))  # new nodes on face n+1
                            nodecount += 2 * len(cn)
                            tk += cn + cn
                            tv += cn1_new + cn2_new
                            ti += len(cn) * [indp1] + len(cn) * [indp1 + indp2 + 1]
                            for i in range(2):
                                for node in cn:
                                    nocoord = np.append(nocoord, [nocoord[node - 1]],
                                                        axis=0)  # add coordinates for new nodes
                            for sixNdFace in contact:
                                rn = mesh.getElementNodes(sixNdFace)  # reference nodes
                                nodeset1 = []
                                nodeset2 = []
                                for nd in rn:
                                    nodeset1.append(cn1_new[cn.index(nd)])
```

```
                                        nodeset2.append(cn2_new[cn.index(nd)])
                                    interface_elements.append(
                                        nodeset1 + nodeset2)  # interface_elements[index] = [nodeA1,..,nodeA6,
nodeB1,..,nodeB6]
                                for node in cn:
                                    for el in net[
                                        node]:  # net: {nodeID : [[eleID, binary node position], [], ...], nodeID : ...}
                                        nodepos = el[1]
                                        nonum = int(math.log(nodepos,
                                                             2))  # local node number from binary node number net[node]
[1]
                                        if ppEl[el[0]] == indp1:
                                            new_node = cn1_new[cn.index(node)]
                                            elNodes[elo[el[0]]][
                                                nonum] = new_node  # connect element to new node in primitive n
                                            NiP[new_node] = indp1
                                        elif ppEl[el[0]] == indp1 + indp2 + 1:
                                            new_node = cn2_new[cn.index(node)]
                                            elNodes[elo[el[0]]][
                                                nonum] = new_node  # connect element to new node in primitive n+1
                                            NiP[new_node] = indp1 + indp2 + 1
                                        shiftelements.append(elo[el[0]])

    # check elnodes
    # for index, el in enumerate(elNodes):
    #     for nd in el:
    #         if nd in tk:
    #             print("THIS CANNOT BE RIGHT")
    #             print("node: ", nd, "primitive: ", ppEl[ole[index]], "el: ", index, el)
    #
    #
    # print("POST-SHIFT")
    # print("element 6: ",elNodes[6])
    # print("element 28: ",elNodes[28])
    # print("element 60: ",elNodes[60])
    # print("element 87: ",elNodes[87])

    # twins = dict(zip(tk, tv))  # twins : {oldnode : [newnodes]}
    twins = {}

    # for index, node in enumerate(tk):
    #     print(tk[index], tv[index], ti[index])
```

```
for ind, node in enumerate(tk):
    if node not in twins:
        twins[node] = {}
    if ti[ind] not in twins[node]:
        twins[node][ti[ind]] = []
    twins[node][ti[ind]].append(tv[ind])

if interface_elements == []:
    interface_elements = np.asarray([[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]])  # explicit signature required for numba

interface_elements = np.asarray(interface_elements)

prn_upd("number of interface elements: {}".format(len(interface_elements)))

ks_red = np.ones(6 * len(interface_elements))

# for index, ife in enumerate(interface_elements):
#     print(index, ife)

# for node in twins:
#     print(node, twins[node])

link0 = [0]
link1 = [0]
for nd in twins:
    for prim in twins[nd]:
        if len(twins[nd][prim]) == 2:
            link0.append(twins[nd][prim][0])
            link1.append(twins[nd][prim][1])

link0 = np.asarray(link0)
link1 = np.asarray(link1)

noce = np.zeros((len(nocoord)), dtype=np.int16)
for inno in range(len(nocoord)):
    i, j = np.where(elNodes == inno + 1)
    noce[inno] = len(i)

# create boundary condition array dispfaces
dispfaces = []
for obj in App.ActiveDocument.Objects:
    if obj.isDerivedFrom('Fem::ConstraintFixed') or obj.isDerivedFrom('Fem::ConstraintDisplacement'):

        bcnodes = []

        if obj.isDerivedFrom('Fem::ConstraintFixed'):
            bctype = [False, False, False]
            bcvalue = [0, 0, 0]
        else:
            bctype = [obj.xFree, obj.yFree, obj.zFree]
            bcvalue = [obj.xDisplacement, obj.yDisplacement, obj.zDisplacement]

        for part, boundaries in obj.References:
            for boundary in boundaries:
                ref = part.Shape.getElement(boundary)
                if type(ref) == Part.Vertex:
                    bc = mesh.getNodesByVertex(ref)
                    for bcn in bc: bcnodes.append(bcn)
                elif type(ref) == Part.Edge:
                    bc = mesh.getNodesByEdge(ref)
                    for bcn in bc: bcnodes.append(bcn)
                elif type(ref) == Part.Face:
                    bc = mesh.getNodesByFace(
                        ref)  # all nodes on a primitive face with a displacement boundary condition
                    for bcn in bc:
                        bcnodes.append(bcn)
                else:
                    prn_upd("No Boundaries Found")
        bcnodes = list(dict.fromkeys(bcnodes))  # remove duplicates in bcnodes
        # print("bcnodes: ", bcnodes)
        bc_ref_nodes = list(set(list(twins.keys())) & set(bcnodes))
        bc_internal_nodes = list(set(bcnodes) - set(list(twins.keys())))
        for nd in bc_ref_nodes:
            for prim in twins[nd]:
                if prim == NiP[bc_internal_nodes[0]]:
                    bcnodes += twins[nd][prim]
        if bcnodes: dispfaces.append([bcnodes, bctype, bcvalue])

# print(dispfaces)
# fix reference nodes
# print(list(twins.keys()))
dispfaces.append([list(twins.keys()), [False, False, False], [0, 0, 0]])
# print(dispfaces)
```

```
lf = [[0, 0, 0, 0, 0, 0]]  # signature for numba
pr = [0]  # idem
for obj in App.ActiveDocument.Objects:
    if obj.isDerivedFrom('Fem::ConstraintPressure'):
        if obj.Reversed:
            sign = 1
        else:
            sign = -1
        # print(obj.References)
        for part, faces in obj.References:  # obj.References: references to loaded primitive faces
            for face in faces:
                ref = part.Shape.getElement(face)
                if type(ref) == Part.Face:
                    for faceID in mesh.getFacesByFace(ref):  # face ID: ID of a 6-node face element
                        face_nodes = list(mesh.getElementNodes(faceID))  # 6-node element node numbers
                        if twins:
                            regular_nodes = list(
                                set(face_nodes) - set(list(twins.keys())))  # nodes outside interface
                            prim = NiP[regular_nodes[0]]  # select primitive of first regular node
                            for index, nd in enumerate(face_nodes):
                                if nd in twins:
                                    face_nodes[index] = twins[nd][
                                        prim][0]  # swap node number from reference node to new node
                        lf.append(face_nodes)
                        pr.append(sign * obj.Pressure)
                else:
                    prn_upd("No Faces with Pressure Loads")
loadfaces = np.asarray(lf)
pressure = np.array(pr)

# re-order element nodes
for el in elNodes:
    temp = el[1]
    el[1] = el[2]
    el[2] = temp
    temp = el[4]
    el[4] = el[6]
    el[6] = temp
    temp = el[8]
    el[8] = el[9]
    el[9] = temp
    return elNodes, nocoord, dispfaces, loadfaces, materialbyElement, interface_elements, noce, pressure, link0, link1,
ks_red


# shape functions for a 4-node tetrahedron - only used for stress interpolation
def shape4tet(xi, et, ze, xl):
    shp = np.zeros((4), dtype=np.float64)

    # shape functions
    shp[0] = 1.0 - xi - et - ze
    shp[1] = xi
    shp[2] = et
    shp[3] = ze

    return shp


@jit(nopython=True, cache=True)
def shp10tet(xi, et, ze):
    shp = np.zeros((10), dtype=np.float64)

    # shape functions - source: Calculix, G Dhondt
    a = 1.0 - xi - et - ze
    shp[0] = (2.0 * a - 1.0) * a
    shp[1] = xi * (2.0 * xi - 1.0)
    shp[2] = et * (2.0 * et - 1.0)
    shp[3] = ze * (2.0 * ze - 1.0)
    shp[4] = 4.0 * xi * a
    shp[5] = 4.0 * xi * et
    shp[6] = 4.0 * et * a
    shp[7] = 4.0 * ze * a
    shp[8] = 4.0 * xi * ze
    shp[9] = 4.0 * et * ze
    return shp


@jit(nopython=True, cache=True)
def dshp10tet(xi, et, ze, xl):
    dshp = np.zeros((3, 10), dtype=np.float64)
    dshpg = np.zeros((3, 10), dtype=np.float64)
    bmat = np.zeros((6, 30), dtype=np.float64)
```

```
xs = np.zeros((3, 3), dtype=np.float64)
xsi = np.zeros((3, 3), dtype=np.float64)

# local derivatives of the shape functions: xi-derivative - source: Calculix, G Dhondt
dshp[0][0] = 1.0 - 4.0 * (1.0 - xi - et - ze)
dshp[0][1] = 4.0 * xi - 1.0
dshp[0][2] = 0.0
dshp[0][3] = 0.0
dshp[0][4] = 4.0 * (1.0 - 2.0 * xi - et - ze)
dshp[0][5] = 4.0 * et
dshp[0][6] = -4.0 * et
dshp[0][7] = -4.0 * ze
dshp[0][8] = 4.0 * ze
dshp[0][9] = 0.0

# local derivatives of the shape functions: eta-derivative - source: Calculix, G Dhondt
dshp[1][0] = 1.0 - 4.0 * (1.0 - xi - et - ze)
dshp[1][1] = 0.0
dshp[1][2] = 4.0 * et - 1.0
dshp[1][3] = 0.0
dshp[1][4] = -4.0 * xi
dshp[1][5] = 4.0 * xi
dshp[1][6] = 4.0 * (1.0 - xi - 2.0 * et - ze)
dshp[1][7] = -4.0 * ze
dshp[1][8] = 0.0
dshp[1][9] = 4.0 * ze

# local derivatives of the shape functions: zeta-derivative - source: Calculix, G Dhondt
dshp[2][0] = 1.0 - 4.0 * (1.0 - xi - et - ze)
dshp[2][1] = 0.0
dshp[2][2] = 0.0
dshp[2][3] = 4.0 * ze - 1.0
dshp[2][4] = -4.0 * xi
dshp[2][5] = 0.0
dshp[2][6] = -4.0 * et
dshp[2][7] = 4.0 * (1.0 - xi - et - 2.0 * ze)
dshp[2][8] = 4.0 * xi
dshp[2][9] = 4.0 * et

# xs = np.dot(xl, dshp.T) # local derivative of the global coordinates

for i in range(3):
    for j in range(3):
        xs[i][j] = 0.0
        for k in range(10):
            xs[i][j] += xl[i][k] * dshp[j][k]

# xsj = np.linalg.det(xs) # Jacobian

xsj = (xs[0][0] * xs[1][1] * xs[2][2] -
       xs[0][0] * xs[1][2] * xs[2][1] +
       xs[0][2] * xs[1][0] * xs[2][1] -
       xs[0][2] * xs[1][1] * xs[2][0] +
       xs[0][1] * xs[1][2] * xs[2][0] -
       xs[0][1] * xs[1][0] * xs[2][2])

# xsi = np.linalg.inv(xs) # global derivative of the local coordinates

xsi[0][0] = (xs[1][1] * xs[2][2] - xs[2][1] * xs[1][2]) / xsj
xsi[0][1] = (xs[0][2] * xs[2][1] - xs[0][1] * xs[2][2]) / xsj
xsi[0][2] = (xs[0][1] * xs[1][2] - xs[0][2] * xs[1][1]) / xsj
xsi[1][0] = (xs[1][2] * xs[2][0] - xs[1][0] * xs[2][2]) / xsj
xsi[1][1] = (xs[0][0] * xs[2][2] - xs[0][2] * xs[2][0]) / xsj
xsi[1][2] = (xs[1][0] * xs[0][2] - xs[0][0] * xs[1][2]) / xsj
xsi[2][0] = (xs[1][0] * xs[2][1] - xs[2][0] * xs[1][1]) / xsj
xsi[2][1] = (xs[2][0] * xs[0][1] - xs[0][0] * xs[2][1]) / xsj
xsi[2][2] = (xs[0][0] * xs[1][1] - xs[1][0] * xs[0][1]) / xsj

# dshp = np.dot(xsi.T, dshp) # global derivatives of the shape functions

for i in range(3):
    for j in range(10):
        for k in range(3):
            dshpg[i][j] += xsi[k][i] * dshp[k][j]

# computation of the strain interpolation matrix bmat
for i in range(10):
    i3 = 3 * i
    d00 = dshpg[0][i]
    d10 = dshpg[1][i]
    d20 = dshpg[2][i]
    bmat[0][i3] = d00
    bmat[1][i3 + 1] = d10
    bmat[2][i3 + 2] = d20
```

```
            bmat[3][i3] = d10
            bmat[3][i3 + 1] = d00
            bmat[4][i3] = d20
            bmat[4][i3 + 2] = d00
            bmat[5][i3 + 1] = d20
            bmat[5][i3 + 2] = d10

    return xsj, dshpg, bmat


# shape functions and their derivatives for a 6-node triangular interface element
@jit(nopython=True, cache=True)
def shape6tri(xi, et, xl):
    # numba
    # shp = np.zeros((6), dtype=types.float64)
    # dshp = np.zeros((2, 6), dtype=types.float64)
    # bmat = np.zeros((3, 36), dtype=types.float64)

    # no numba
    shp = np.zeros((6), dtype=np.float64)
    dshp = np.zeros((2, 6), dtype=np.float64)
    bmat = np.zeros((3, 36), dtype=np.float64)

    # shape functions
    shp[0] = (1.0 - xi - et) * (1.0 - 2.0 * xi - 2.0 * et)
    shp[1] = xi * (2.0 * xi - 1.0)
    shp[2] = et * (2.0 * et - 1.0)
    shp[3] = 4.0 * xi * (1.0 - xi - et)
    shp[4] = 4.0 * xi * et
    shp[5] = 4.0 * et * (1 - xi - et)

    # local derivatives of the shape functions: xi-derivative
    dshp[0][0] = -3.0 + 4.0 * et + 4.0 * xi
    dshp[0][1] = -1.0 + 4.0 * xi
    dshp[0][2] = 0.0
    dshp[0][3] = -4.0 * (-1.0 + et + 2.0 * xi)
    dshp[0][4] = 4.0 * et
    dshp[0][5] = -4.0 * et

    # local derivatives of the shape functions: eta-derivative
    dshp[1][0] = -3.0 + 4.0 * et + 4.0 * xi
    dshp[1][1] = 0.0
    dshp[1][2] = -1.0 + 4.0 * et
    dshp[1][3] = -4.0 * xi
    dshp[1][4] = 4.0 * xi
    dshp[1][5] = -4.0 * (-1.0 + 2.0 * et + xi)

    xs = np.dot(dshp, xl.T)  # xs = [ [[dx/dxi],[dy/dxi],[dz/dxi]] , [[dx/det],[dy/det],[dz/det]] ]

    xp = np.cross(xs[0], xs[1])  # vector normal to surface

    xsj = np.linalg.norm(xp)  # Jacobian

    # xsj = np.sqrt(xp[0]*xp[0]+xp[1]*xp[1]+xp[2]*xp[2])

    xx = xs[0] / np.linalg.norm(xs[0])  # unit vector in xi direction

    # xx = np.sqrt(xs[0]*xs[0]+xs[1]*xs[1]+xs[2]*xs[2])

    xp /= xsj  # unit vector normal to surface
    xt = np.cross(xp, xx)  # unit vector tangential to surface and normal to xx

    # computation of the "strain" interpolation matrix bmat
    for i in range(6):
        ia = 3 * i
        ib = ia + 18
        ni = shp[i]
        bmat[0][ia] = ni
        bmat[1][ia + 1] = ni
        bmat[2][ia + 2] = ni
        bmat[0][ib] = -ni
        bmat[1][ib + 1] = -ni
        bmat[2][ib + 2] = -ni

    return xsj, shp, bmat, xx, xt, xp


# linear-elastic material stiffness matrix
@jit(nopython=True, cache=True)
def hooke(element, materialbyElement):
    dmat = np.zeros((6, 6), dtype=np.float64)
    e = materialbyElement[element][0]  # Young's Modulus
    nu = materialbyElement[element][1]  # Poisson's Ratio
    dm = e * (1.0 - nu) / (1.0 + nu) / (1.0 - 2.0 * nu)
    od = nu / (1.0 - nu)
    sd = 0.5 * (1.0 - 2.0 * nu) / (1.0 - nu)

    dmat[0][0] = dmat[1][1] = dmat[2][2] = 1.0
    dmat[3][3] = dmat[4][4] = dmat[5][5] = sd
    dmat[0][1] = dmat[0][2] = dmat[1][2] = od
    dmat[1][0] = dmat[2][0] = dmat[2][1] = od
    dmat *= dm

    return dmat


# Gaussian integration points and weights
@jit(nopython=True, cache=True)
def gaussPoints():
    # Gaussian integration points and weights for 10-noded tetrahedron
    gp10 = np.array([[0.138196601125011, 0.138196601125011, 0.138196601125011,
                      0.041666666666667],
                     [0.585410196624968, 0.138196601125011, 0.138196601125011,
                      0.041666666666667],
                     [0.138196601125011, 0.585410196624968, 0.138196601125011,
                      0.041666666666667],
                     [0.138196601125011, 0.138196601125011, 0.585410196624968,
                      0.041666666666667]])
    # Gaussian integration points and weights for 6-noded triangle
    gp6 = np.array([[0.445948490915965, 0.445948490915965,
                     0.111690794839005],
                    [0.10810301816807, 0.445948490915965,
                     0.111690794839005],
                    [0.445948490915965, 0.10810301816807,
                     0.111690794839005],
                    [0.091576213509771, 0.091576213509771,
                     0.054975871827661],
                    [0.816847572980458, 0.091576213509771,
                     0.054975871827661],
                    [0.091576213509771, 0.816847572980458,
                     0.054975871827661]])
    return gp10, gp6


# Nodal point locations
@jit(nopython=True, cache=True)
```

```
def nodalPoints():
    # Nodal point locations for a 10-noded tetrahedron
    np10 = np.array([[0.0, 0.0, 0.0],
                     [1.0, 0.0, 0.0],
                     [0.0, 1.0, 0.0],
                     [0.0, 0.0, 1.0],
                     [0.5, 0.0, 0.0],
                     [0.5, 0.5, 0.0],
                     [0.0, 0.5, 0.0],
                     [0.0, 0.0, 0.5],
                     [0.5, 0.0, 0.5],
                     [0.0, 0.5, 0.5]])
    # Nodal point locations for 6-noded triangle + Newton Cotes integration weights
    np6 = np.array([[0.0, 0.0, 0.0],
                    [1.0, 0.0, 0.0],
                    [0.0, 1.0, 0.0],
                    [0.5, 0.0, 0.166666666666666],
                    [0.5, 0.5, 0.166666666666666],
                    [0.0, 0.5, 0.166666666666666]])
    return np10, np6


# caculate the global stiffness matrix and load vector
@jit(nopython=True, cache=True)
def calcGSM(elNodes, nocoord, materialbyElement, loadfaces, interface_elements,
            grav, kn, ks, pressure, link0, link1, ks_red):
    gp10, gp6 = gaussPoints()
    # np10, np6 = nodalPoints()  # required here only for Newton Coates integration
    nn = len(nocoord[:, 0])

    # numba
    dmatloc = np.zeros((3, 3), dtype=types.float64)
    T = np.zeros((3, 3), dtype=types.float64)
    xlf = np.zeros((3, 6), dtype=types.float64)
    xlv = np.zeros((3, 10), dtype=types.float64)
    xli = np.zeros((3, 6), dtype=types.float64)
    nodes_int = np.zeros(12, dtype=types.int64)
    gsm = np.zeros((3 * nn, 3 * nn), dtype=types.float64)
    glv = np.zeros((3 * nn), dtype=types.float64)

    # no numba
    # dmatloc = np.zeros((3, 3), dtype=np.float64)
    # T = np.zeros((3, 3), dtype=np.float64)
    # xlf = np.zeros((3, 6), dtype=np.float64)
    # xlv = np.zeros((3, 10), dtype=np.float64)
    # xli = np.zeros((3, 6), dtype=np.float64)
    # nodes_int = np.zeros(12, dtype=np.int64)
    # gsm = np.zeros((3 * nn, 3 * nn), dtype=np.float64)
    # glv = np.zeros((3 * nn), dtype=np.float64)

    #   calculate element load vectors for pressure and add to global vector

    for face in range(len(pressure) - 1):
        if len(pressure) == 1:
            break

        nda = loadfaces[face + 1]
        for i in range(3):
            for j in range(6):
                nd = nda[j]
                xlf[i][j] = nocoord[nd - 1][i]

        # integrate element load vector
        for index in range(len(gp6)):
            xi = gp6[index][0]
            et = gp6[index][1]
            xsj, shp, bmat, xx, xt, xp = shape6tri(xi, et, xlf)
            nl = 0
            for i in range(len(loadfaces[face] - 1)):
                nd = loadfaces[face + 1][i]
                iglob = nd - 1
                iglob3 = 3 * iglob
                for k in range(3):
                    load = shp[nl] * pressure[face + 1] * xp[k] * abs(xsj) * gp6[index][2]
                    glv[iglob3 + k] += load
                nl += 1

    # for each volume element calculate the element stiffness matrix
    # and gravity load vector and add to global matrix and vector

    for el in range(len(elNodes)):
        nodes = elNodes[el]
        V = 0.0
        esm = np.zeros((30, 30), dtype=np.float64)
```

```
        gamma = np.zeros((30), dtype=np.float64)
        dmat = hooke(el, materialbyElement)

        # set up nodal values for this element
        el14 = False
        for i in range(3):
            for j in range(10):
                nd = nodes[j]
                xlv[i][j] = nocoord[nd - 1][i]

        # integrate element matrix
        for index in range(len(gp10)):
            ip = gp10[index]
            xi = ip[0]
            et = ip[1]
            ze = ip[2]
            shp = shp10tet(xi, et, ze)
            xsj, dshp, bmat = dshp10tet(xi, et, ze, xlv)
            esm += np.dot(bmat.T, np.dot(dmat, bmat)) * ip[3] * abs(xsj)
            gamma[2::3] += grav * shp * ip[3] * abs(xsj)
            V += xsj * ip[3]  # Element volume - not used

        # add element matrix to global stiffness matrix and element gravity
        # load vector to global load vector
        for i in range(10):
            iglob = nodes[i] - 1
            iglob3 = 3 * iglob
            i3 = 3 * i
            glv[iglob3 + 2] += gamma[i3 + 2]
            for j in range(10):
                jglob = nodes[j] - 1
                jglob3 = 3 * jglob
                j3 = j * 3
                for k in range(3):
                    for l in range(3):
                        gsm[iglob3 + k, jglob3 + l] += esm[i3 + k, j3 + l]

    # interface stiffness value
    kmax = 0.01 * np.amax(np.diag(gsm))

    print("minimum diagonal stiffness: ", np.amin(np.diag(gsm)))
    print("maximum diagonal stiffness: ", np.amax(np.diag(gsm)))
```

```
# For each interface element calculate the element matrix and
# add to global stiffness matrix

for el in range(len(interface_elements)):

    if interface_elements[0][0] == 0: break
    for i in range(12):
        nodes_int[i] = interface_elements[el][i]
    for i in range(6):
        nd = nodes_int[i]
        for j in range(3):
            xli[j][i] = nocoord[nd - 1][j]

    esm = np.zeros((36, 36), dtype=np.float64)
    dmatloc[0][0] = kn * kmax

    # integrate element matrix (np6: Newton Cotes, gp6: Gauss)

    # ORIGINAL PYTHON
    # for ip in gp6:
    #     xi = ip[0]
    #     et = ip[1]
    #     xsj, shp, bmat, xx, xt, xp = shape6tri(xi, et, xli)
    #     T = np.array([xp, xx, xt])
    #     dmatglob = np.dot(T.T, np.dot(dmatloc, T))
    #     esm += np.dot(bmat.T, np.dot(dmatglob, bmat)) * abs(xsj) * ip[2]

    # NUMBA
    for index in range(len(gp6)):
        dmatloc[1][1] = dmatloc[2][2] = ks_red[6 * el + index] * ks * kmax
        # print("integration point: ", 6 * el + index, "ks_red: ", ks_red[6 * el + index])

        ip = gp6[index]
        xi = ip[0]
        et = ip[1]
        xsj, shp, bmat, xx, xt, xp = shape6tri(xi, et, xli)
        T[0] = xp
        T[1] = xx
        T[2] = xt
        dmatglob = np.dot(T.T, np.dot(dmatloc, T))
        esm += np.dot(bmat.T, np.dot(dmatglob, bmat)) * abs(xsj) * ip[2]
    # add Element matrix to global stiffness matrix
    for i in range(12):
        iglob = nodes_int[i] - 1
        iglob3 = 3 * iglob
        i3 = 3 * i
        for j in range(12):
            jglob = nodes_int[j] - 1
            jglob3 = 3 * jglob
            j3 = j * 3
            for k in range(3):
                for l in range(3):
                    gsm[iglob3 + k, jglob3 + l] += esm[i3 + k, j3 + l]

# add stiff links
# print("link0: ", link0)
for index, node in enumerate(link0):
    if node != 0:
        print("ADD STIFF LINK - BE WARE OF ASSOCIATED FORCES")
        n3a = 3 * (int(node) - 1)
        n3b = 3 * (int(link1[index]) - 1)
        for i in range(3):
            stiff = 100 * kmax
            gsm[n3a + i, n3a + i] += stiff
            gsm[n3b + i, n3b + i] += stiff
            gsm[n3a + i, n3b + i] += -stiff
            gsm[n3b + i, n3a + i] += -stiff

loadsumx = 0.0
loadsumy = 0.0
loadsumz = 0.0
for node in range(nn):
    dof = 3 * node
    loadsumx += glv[dof]
    loadsumy += glv[dof + 1]
    loadsumz += glv[dof + 2]

# print("sumFx: ", loadsumx)
# print("sumFy: ", loadsumy)
# print("sumFz: ", loadsumz)

# for node in range(len(nocoord)):
    #     base = 3*node
    #     for dof in range(3):
    #         if (gsm[base+dof][base+dof] == 0.0):
    #             print(node+1, nocoord[node])

    return gsm, glv, kmax


# calculate load-deflection curve
def calcDisp(elNodes, nocoord, dispfaces, materialbyElement, interface_elements, kmax, gsm, glv, nstep, iterat_max,
             error_max, relax, scale_re,
             scale_up, scale_dn,
             sig_yield, shr_yield, kn,
             ks, out_disp, link0, link1,
             elMat, loadFaces, gravity, pressure, ks_red):
    ndof = len(glv)
    nelem = len(elNodes)

    if interface_elements[0][0] != 0:
        ninter = len(interface_elements)
    else:
        ninter = 0

    # glv and gsm will be impacted by non-zero prescribed displacements, so make a copy
    # to preserve both for use in the iterative scheme
    qex = np.copy(glv)
    qnorm = np.linalg.norm(qex)
    if qnorm < 1.0: qnorm = 1.0

    # modify the global stiffness matrix and load vector for displacement BC
    gsm, glv, fixdof = bcGSM(gsm, glv, dispfaces)

    if np.min(np.diag(gsm)) <= 0.0:
        prn_upd("non positive definite matrix - check input")
        raise SystemExit()

    # Cholesky decomposition of the global stiffness matrix and elastic solution
    # TODO: Apply reverse Cuthill McKee and banded Cholesky to speed things up
    # TODO: Experiment with Intel Distribution for Python (Math Kernal Library) to optimize speed

    # using cholmod
    A = scsp.csc_matrix(gsm, dtype=np.float64)
```

```
b = glv
t0 = time.time()
factor = cholesky(A)
t1 = time.time()
ue = factor(b)
t2 = time.time()
prn_upd("Sparse Cholesky Decomposition: {} log10(s), Solution: {} log10(s)".format(np.log10(t1 - t0),
                                                                                  np.log10(t2 - t1)))

# initiate analysis
dl0 = 1.0 / nstep  # nstep == 1 execute an elastic analysis
dl = dl0
du = dl * ue

sig = np.array([np.zeros((24 * nelem), dtype=np.float64)])  # stress in Tet10
trac = np.array([np.zeros((18 * max(ninter, 1)), dtype=np.float64)])  # contact stress in Tri6
disp = np.array([np.zeros((ndof), dtype=np.float64)])  # displacement results
lbd = np.zeros((1), dtype=np.float64)  # load level

step = -1
cnt = True

un = []

if (nstep == 1.0):
    # if (interface_elements[0][0] != 0 or nstep == 1.0):
    # perform an elastic (one-step) analysis only
    step = 0
    out_disp = 1
    un = [0.]
    lbd = np.append(lbd, 1.0)
    disp = np.append(disp, [ue], axis=0)
    un.append(np.max(np.abs(disp[1])))
    cnt = False

while (cnt == True):
    cnt = False
    for istep in (range(nstep)):
        step += 1
        restart = 0
        prn_upd("Step: {}".format(step))
        # Riks control vector
```

```
a = du
# lbd = load level
lbd = np.append(lbd, lbd[step] + dl)

# update stresses and loads

sig_update, trac_update, qin, ks_red = update_stress_load(elNodes, nocoord, materialbyElement,
                                                          sig_yield, du, np.array(interface_elements),
                                                          trac[step], kmax, kn, ks, shr_yield, sig[step],
                                                          link0, link1, ks_red)

sig = np.append(sig, np.array([sig_update]), axis=0)

trac = np.append(trac, np.array([trac_update]), axis=0)

# calculate residual load vector
fex = fixdof * lbd[step + 1] * qex
fin = fixdof * qin
r = fex - fin
rnorm = np.linalg.norm(r)

# out-of-balance error
error = rnorm / qnorm

iterat = 0
prn_upd("Iteration: {}, Error: {}".format(iterat, error))

# raise SystemExit()

while error > error_max:

    iterat += 1

    if iterat == 1 and interface_elements[0][0] != 0:
        # if interface_elements[0][0] == 0: break
        prn_upd("Update Tangent Stiffness Matrix for Interface Elements")
        gsm, glv, kmax = calcGSM(elNodes, nocoord, elMat, loadFaces,
                                 interface_elements,
                                 gravity, kn, ks, pressure, link0, link1,
                                 ks_red)

        # modify the tangent stiffness matrix and load vector for displacement BC
        gsm, glv, fixdof = bcGSM(gsm, qex, dispfaces)
        A = scsp.csc_matrix(gsm, dtype=np.float64)
        factor = cholesky(A)
        # K_inv * External Load - required in Riks control
        ue = factor(b)

    # K_inv * Unbalanced Force - displacement correction
    due = factor(relax * r)

    # Riks control correction to load level increment
    dl = -np.dot(a, due) / np.dot(a, ue)
    lbd[step + 1] += dl

    # Riks control correction to displacement increment
    du += due + dl * ue

    # update stresses and loads
    sig_update, trac_update, qin, ks_red = update_stress_load(elNodes, nocoord, materialbyElement,
                                                              sig_yield, du, np.array(interface_elements),
                                                              trac[step], kmax, kn, ks, shr_yield,
                                                              sig[step], link0, link1, ks_red)

    sig[step + 1] = np.array([sig_update])

    trac[step + 1] = np.array([trac_update])

    # calculate out of balance error
    r = fixdof * (lbd[step + 1] * qex - qin)
    rnorm = np.linalg.norm(r)
    error = rnorm / qnorm
    prn_upd("Iteration: {}, Error: {}".format(iterat, error))

    if iterat > iterat_max:
        # scale down
        if restart == 4:
            print("MAXIMUM RESTARTS REACHED")
            raise SystemExit()
        restart += 1
        if step > 0:
            dl = (lbd[step] - lbd[step - 1]) / scale_re / restart
            du = (disp[step] - disp[step - 1]) / scale_re / restart
        else:
                # for first step only
                dl = dl0 / scale_re / restart
                du = dl * ue / scale_re / restart
            lbd[step + 1] = lbd[step] + dl

            sig_update, trac_update, qin, ks_red = update_stress_load(elNodes, nocoord, materialbyElement,
                                                                      sig_yield, du,
                                                                      np.array(interface_elements),
                                                                      trac[step], kmax, kn, ks, shr_yield,
                                                                      sig[step], link0, link1, ks_red)

            sig[step + 1] = np.array([sig_update])

            trac[step + 1] = np.array([trac_update])

            r = fixdof * (lbd[step + 1] * qex - qin)
            rnorm = np.linalg.norm(r)
            error = rnorm / qnorm

            iterat = 0

    # update results at end of converged load step
    disp = np.append(disp, [disp[step] + du], axis=0)
    dl = lbd[step + 1] - lbd[step]

    if iterat > 10:
        # scale down
        dl /= scale_dn
        du /= scale_dn
    if iterat < 5:
        # scale up
        dl *= scale_up
        du *= scale_up

# maximum displacement increment for plotting load-displacement curve
un = []
for index, load in enumerate(lbd):
    un.append(np.max(np.abs(disp[index])))

# plot load-displacement curve - TODO: move to output / post-processing
cnt = plot(un, lbd)
```

```
    out = min(step + 1, abs(int(out_disp)))
    if out_disp > 0:
        u_out = un[out]
        l_out = lbd[out]
        # prn_upd("\n************************************************************\n")
        prn_upd("Step: {0:2d} Load level: {1:.3f} Displacement: {2:.4e}".format(out, l_out,
                                                                                u_out))
        # prn_upd("\n************************************************************\n")
        return disp[out], sig, trac

    else:
        u_out = un[out] - un[out - 1]
        l_out = lbd[out] - lbd[out - 1]
        # prn_upd("\n************************************************************\n")
        prn_upd("Step: {0:2d} Load level increment: {1:.3f} Displacement increment: {2:.4e}".format(out, l_out,
                                                                                                    u_out))
        # prn_upd("\n************************************************************\n")
        return disp[out] - disp[out - 1], sig, trac


# plot the load-deflection curve
def plot(un, lbd):
    class Index(object):
        def stop(self, event):
            self.cnt = False
            plt.close()
            self.clicked = True

        def add(self, event):
            self.cnt = True
            plt.close()
            self.clicked = True

    callback = Index()
    callback.cnt = False
    callback.clicked = False
    fig, ax = plt.subplots()
    plt.subplots_adjust(bottom=0.2)
    ax.plot(un, lbd, '-ok')
    ax.set(xlabel='displacement [mm]', ylabel='load factor [-]',
           title='')
    ax.grid()
    axstop = plt.axes([0.7, 0.05, 0.1, 0.075])
    axadd = plt.axes([0.81, 0.05, 0.1, 0.075])
    bstop = Button(axstop, 'stop')
    bstop.on_clicked(callback.stop)
    badd = Button(axadd, 'add')
    badd.on_clicked(callback.add)
    # fig.savefig("test.png")
    plt.show()

    while True:
        plt.pause(0.01)
        if callback.clicked:
            break

    return callback.cnt


# modify the global stiffness matrix and load vector for displacement boundary conditions
def bcGSM(gsm, glv, dis):
    dim = len(glv)
    zero = np.zeros((dim), dtype=np.float64)

    # fixdof=1: DOF is free; fixdof=0: DOF is fixed - used in calculation of residual load
    fixdof = np.ones((dim), dtype=int)

    for lf in dis:
        lx = lf[1][0]  # True: free x-DOF; False: fixed x-DOF
        ly = lf[1][1]  # True: free y-DOF; False: fixed y-DOF
        lz = lf[1][2]  # True: free z-DOF; False: fixed z-DOF
        ux = uy = uz = 0.0

        if not lx: ux = lf[2][0]  # prescribed displacement in x-direction
        if not ly: uy = lf[2][1]  # prescribed displacement in y-direction
        if not lz: uz = lf[2][2]  # prescribed displacement in z-direction

        for node in lf[0]:
            n3 = 3 * (int(node) - 1)
            if not lx:
                fixdof[n3] = 0
                glv -= ux * gsm[n3]
                gsm[:, n3] = zero
                gsm[n3] = zero
                gsm[n3, n3] = 1.0
                glv[n3] = ux
            if not ly:
                fixdof[n3 + 1] = 0
                glv -= uy * gsm[n3 + 1]
                gsm[:, n3 + 1] = zero
                gsm[n3 + 1] = zero
                gsm[n3 + 1, n3 + 1] = 1.0
                glv[n3 + 1] = uy
            if not lz:
                fixdof[n3 + 2] = 0
                glv -= uz * gsm[n3 + 2]
                gsm[:, n3 + 2] = zero
                gsm[n3 + 2] = zero
                gsm[n3 + 2, n3 + 2] = 1.0
                glv[n3 + 2] = uz

    return gsm, glv, fixdof


# update stresses and loads

@jit(nopython=True, cache=True, nogil=True)
def update_stress_load(elNodes, nocoord, materialbyElement, sig_yield, du,
                       interface_elements, trac, kmax, kn, ks, shr_yield, sig, link0, link1, ks_red):
    gp10, gp6 = gaussPoints()
    np10, np6 = nodalPoints()

    # NUMBA:
    xlv = np.zeros((3, 10), dtype=types.float64)
    xli = np.zeros((3, 6), dtype=types.float64)
    dmatloc = np.zeros((3, 3), dtype=types.float64)
    T = np.zeros((3, 3), dtype=types.float64)
    nodes = np.zeros(10, dtype=types.int64)
    nodes_int = np.zeros(12, dtype=types.int64)
    u10 = np.zeros((30), dtype=types.float64)  # displacements for the 10 tetrahedral nodes
    u12 = np.zeros((36), dtype=types.float64)  # displacements for the 12 triangular node pairs
    sig_update = np.zeros(24 * len(elNodes), dtype=types.float64)
    trac_update = np.zeros(18 * len(interface_elements), dtype=types.float64)
    qin = np.zeros(3 * len(nocoord), dtype=types.float64)  # internal load vector

    # ORIGINAL PYTHON:
```

```
# xlv = np.zeros((3, 10), dtype=np.float64)
# xli = np.zeros((3, 6), dtype=np.float64)
# dmatloc = np.zeros((3, 3), dtype=np.float64)
# T = np.zeros((3, 3), dtype=np.float64)
# nodes = np.zeros(10, dtype=int)
# nodes_int = np.zeros(12, dtype=int)
# u10 = np.zeros((30), dtype=np.float64)  # displacements for the 10 tetrahedral nodes
# u12 = np.zeros((36), dtype=np.float64)  # displacements for the 12 triangular node pairs
# sig_update = np.zeros(24 * len(elNodes), dtype=np.float64)
# trac_update = np.zeros(18 * len(interface_elements), dtype=np.float64)
# qin = np.zeros(3 * len(nocoord), dtype=np.float64)  # internal load vector

# VOLUME ELEMENTS
for el in range(len(elNodes)):
    for i in range(10):
        nodes[i] = elNodes[el][i]
    elpos = 24 * el
    dmat = hooke(el, materialbyElement)
    elv = np.zeros((30), dtype=np.float64)
    # xl = np.array([nocoord[nd-1] for nd in nodes]).T
    for index, nd in enumerate(nodes):
        for i in range(3):
            xlv[i][index] = nocoord[nd - 1][i]
        n3 = 3 * (nd - 1)
        i3 = 3 * index
        u10[i3] = du[n3]
        u10[i3 + 1] = du[n3 + 1]
        u10[i3 + 2] = du[n3 + 2]
    index = -1
    # for ip in gp10:
    for i in range(4):
        index += 1
        ip = gp10[i]
        ippos = elpos + 6 * index
        xi = ip[0]
        et = ip[1]
        ze = ip[2]
        xsj, dshp, bmat = dshp10tet(xi, et, ze, xlv)

        # elastic test stress
        sig_test = np.dot(dmat, np.dot(bmat, u10))
        sig_test[0] += sig[ippos]
```

```
            sig_test[1] += sig[ippos + 1]
            sig_test[2] += sig[ippos + 2]
            sig_test[3] += sig[ippos + 3]
            sig_test[4] += sig[ippos + 4]
            sig_test[5] += sig[ippos + 5]

            # von Mises stress
            p = (sig_test[0] + sig_test[1] + sig_test[2]) / 3.0

            sig_mises = np.sqrt(1.5 * ((sig_test[0] - p) ** 2 + (sig_test[1] - p) ** 2 + (sig_test[2] - p) ** 2) +
                                3.0 * (sig_test[3] ** 2 + sig_test[4] ** 2 + sig_test[5] ** 2))

            # radial stress return to yield surface
            fac = np.minimum(sig_yield / sig_mises, 1.0)

            sig_update[ippos] = fac * (sig_test[0] - p) + p
            sig_update[ippos + 1] = fac * (sig_test[1] - p) + p
            sig_update[ippos + 2] = fac * (sig_test[2] - p) + p
            sig_update[ippos + 3] = fac * sig_test[3]
            sig_update[ippos + 4] = fac * sig_test[4]
            sig_update[ippos + 5] = fac * sig_test[5]

            elv += np.dot(bmat.T, sig_update[ippos:ippos + 6]) * ip[3] * abs(xsj)

        for i in range(10):
            iglob = nodes[i] - 1
            iglob3 = 3 * iglob
            i3 = 3 * i
            for k in range(3):
                qin[iglob3 + k] += elv[i3 + k]

    # INTERFACE ELEMENTS

    for el in range(len(interface_elements)):
        if interface_elements[0][0] == 0: break

        for i in range(12):
            nodes_int[i] = interface_elements[el][i]

        elpos = 18 * el

        # material matrix in local coordinate system
```

```
dmatloc[0][0] = kn * kmax

# element nodal displacements
for index, nd in enumerate(nodes_int):
    n3 = 3 * (nd - 1)
    i3 = 3 * index
    u12[i3] = du[n3]
    u12[i3 + 1] = du[n3 + 1]
    u12[i3 + 2] = du[n3 + 2]
inelv = np.zeros((36), dtype=np.float64)

for i in range(6):
    nd = nodes_int[i]
    for j in range(3):
        xli[j][i] = nocoord[nd - 1][j]

for i in range(6):
    dmatloc[1][1] = ks_red[6 * el + i] * ks * kmax
    dmatloc[2][2] = ks_red[6 * el + i] * ks * kmax
    ip = gp6[i]
    nppos = elpos + 3 * i
    xi = ip[0]
    et = ip[1]
    xsj, shp, bmat, xx, xt, xp = shape6tri(xi, et, xli)

    T[0] = xp
    T[1] = xx
    T[2] = xt

    # initial interface stress in local coordinates
    int_stress_ini = np.dot(trac[nppos:nppos + 3], T.T)

    # interface strain increment in local coordinates
    int_strain_inc = np.dot(np.dot(bmat, u12), T.T)

    # test stress increment in local coordinates
    int_stress_inc = np.dot(dmatloc, int_strain_inc)

    # total test stress in local coordinates
    int_stress_tot = int_stress_ini + int_stress_inc

    normal = int_stress_tot[0]
```

```
shear1 = int_stress_tot[1]
shear2 = int_stress_tot[2]
shear = np.sqrt(shear1 ** 2 + shear2 ** 2)
if shear != 0.0:
    fac = min(shr_yield / shear, 1.0)
else:
    fac = 1.0

if fac < 1.0: ks_red[6 * el + i] = 0.01

# print("ks_red: ", ks_red)

# updated stress in local coordinates
int_stress_loc = np.array([normal, fac * shear1, fac * shear2])

# updated stress in global coordinates
int_stress_glo = np.dot(T.T, int_stress_loc)

trac_update[nppos:nppos + 3] = int_stress_glo

# dmatglob = np.dot(T.T, np.dot(dmatloc, T))
#
# # elastic test stress in global coordinates
# trac_test_global = trac[nppos:nppos + 3] + np.dot(dmatglob, np.dot(bmat, u12))
#
# # elastic test stress in local coordinates
# trac_test_local = np.dot(trac_test_global, T.T)
#
# normal = trac_test_local[0]
# shear1 = trac_test_local[1]
# shear2 = trac_test_local[2]
# shear = np.sqrt(shear1 ** 2 + shear2 ** 2)
# if shear != 0.0:
#     fac = min(shr_yield / shear, 1.0)
# else:
#     fac = 1.0
# # print(normal, shear, fac)
#
# npstress_local = np.array([normal, fac * shear1, fac * shear2])
# npstress_global = np.dot(T.T, npstress_local)
#
# trac_update[nppos:nppos + 3] = npstress_global
```

```
            # integrate element load vector
            inelv += np.dot(bmat.T, int_stress_glo) * abs(xsj) * ip[2]

        for i in range(12):
            iglob = nodes_int[i] - 1
            iglob3 = 3 * iglob
            i3 = 3 * i
            for k in range(3):
                qin[iglob3 + k] += inelv[i3 + k]

    # STIFF LINK FORCES
    stiff = 100 * kmax
    for index, node in enumerate(link0):
        if node != 0:
            # print("STIFF LINK FORCES")
            n3a = 3 * (int(node) - 1)
            n3b = 3 * (int(link1[index]) - 1)
            for i in range(3):
                flink = stiff * (du[n3a + i] - du[n3b + i])
                qin[n3a + i] += flink
                qin[n3b + i] -= flink

    return sig_update, trac_update, qin, ks_red


# map stresses to nodes
def mapStresses(elNodes, nocoord, interface_elements, disp, sig, trac, noce):
    # map maps corner node stresses to all tet10 nodes
    map = np.array([[1.0, 0.0, 0.0, 0.0],
                    [0.0, 1.0, 0.0, 0.0],
                    [0.0, 0.0, 1.0, 0.0],
                    [0.0, 0.0, 0.0, 1.0],
                    [0.5, 0.5, 0.0, 0.0],
                    [0.0, 0.5, 0.5, 0.0],
                    [0.5, 0.0, 0.5, 0.0],
                    [0.5, 0.0, 0.0, 0.5],
                    [0.0, 0.5, 0.0, 0.5],
                    [0.0, 0.0, 0.5, 0.5]])

    expm = np.zeros((4, 4), dtype=np.float64)  # extrapolation matrix from Gauss points to corner nodes
    expm_int = np.zeros((6, 6), dtype=np.float64)  # extrapolation matrix from Integration Points to 6 tri6 nodes
```

```python
ipstress = np.zeros((4, 6), dtype=np.float64)  # Tet10 stresses by Gauss point
iptrac = np.zeros((6, 3), dtype=np.float64)  # Tri6 tractions by integration point

ip10, ip6 = gaussPoints()
np10, np6 = nodalPoints()

tet10stress = np.zeros((len(nocoord), 6), dtype=np.float64)
contactpressurevector = np.zeros((len(nocoord), 3), dtype=np.float64)
contactpressurevalue = np.zeros((len(nocoord)), dtype=np.float64)
contactshearvector = np.zeros((len(nocoord), 3), dtype=np.float64)

xp_node = np.zeros((6, 3), dtype=np.float64)  # normal vector in each of the 6 integration points
xx_node = np.zeros((6, 3), dtype=np.float64)  # Xi tangential vector in each of the 6 integration points
xt_node = np.zeros((6, 3),
                   dtype=np.float64)  # shear vector ppd to the above 2 vectors in each of the 6 integration points

step = len(sig) - 1  # last step in the results

# map stresses in volumes to nodal points
for el, nodes in enumerate(elNodes):
    elpos = 24 * el
    xl = np.array([nocoord[nd - 1] for nd in nodes]).T
    for index, ip in enumerate(ip10):
        xi = ip[0]
        et = ip[1]
        ze = ip[2]
        shp = shape4tet(xi, et, ze, xl)
        ippos = elpos + 6 * index
        ipstress[index] = sig[step][ippos:ippos + 6]  # ipstress (4x6): 6 stress components for 4 integration points
        for i in range(4):
            expm[index, i] = shp[i]
    expm_inv = np.linalg.inv(expm)
    npstress4 = np.dot(expm_inv, ipstress)  # npstress4 (4x6): for each corner node (4) all stress components (6)
    numnodes = np.array(
        [noce[nodes[n] - 1] for n in range(10)])  # numnodes = number of nodes connected to node "nodes[n]-1"
    npstress10 = np.divide(np.dot(map, npstress4).T,
                           numnodes).T  # nodal point stress all nodes divided by number of connecting elements
    for index, nd in enumerate(nodes): tet10stress[nd - 1] += npstress10[index]

# For each interface element map the tractions to element nodes (np6: Newton Cotes, gp6: Gauss)
# TODO: for extrapolated Gauss point results nodal averaging is required
for el, nodes in enumerate(interface_elements):
        if interface_elements[0][0] == 0: break
        elpos = 18 * el
        xl = np.array([nocoord[nd - 1] for nd in nodes[:6]]).T
        for index, ip in enumerate(np6):
            xi = ip[0]
            et = ip[1]
            xsj, shp, bmat, xx, xt, xp = shape6tri(xi, et, xl)
            xp_node[index] = xp
            xx_node[index] = xx
            xt_node[index] = xt
            T = np.array([xp, xx, xt])
            ippos = elpos + 3 * index
            iptrac[index] = np.dot(T, trac[step][ippos:ippos + 3])  # local stresses at integration points
            for i in range(6):
                expm_int[index, i] = shp[i]
        expm_int_inv = np.linalg.inv(expm_int)
        nptrac = np.dot(expm_int_inv, iptrac)  # local stresses extrapolated to nodes
        # add local contact stresses to global vectors
        for index, nd in enumerate(nodes[:6]):
            contactpressurevector[nd - 1] = nptrac[index][0] * xp_node[index]
            contactpressurevalue[nd - 1] = nptrac[index][0]
            contactshearvector[nd - 1] = nptrac[index][1] * xx_node[index] + nptrac[index][2] * xt_node[index]

    return tet10stress, contactpressurevector, contactpressurevalue, contactshearvector


# fill resultobject with results
def pasteResults(doc, elNodes, nocoord, interface_elements, dis, tet10stress, contactpressurevector,
                 contactpressurevalue, contactshearvector):
    analysis = doc.getObject("Analysis")

    if analysis == None:
        prn_upd("No Analysis object. Please create one first")
        raise SystemExit()

    resVol = analysis.addObject(ObjectsFem.makeResultMechanical(doc))[0]
    resInt = analysis.addObject(ObjectsFem.makeResultMechanical(doc))[0]

    # VOLUME MESH START
    volnodes = {}
    mode_disp_vol = {}
    elements_tetra10 = {}
```

```
mode_results_vol = {}
results = []

for index, coord in enumerate(nocoord):
    n3 = 3 * index
    volnodes[index + 1] = App.Vector(coord[0], coord[1], coord[2])
    mode_disp_vol[index + 1] = App.Vector(dis[n3], dis[n3 + 1], dis[n3 + 2])

for index, elem in enumerate(elNodes):
    elements_tetra10[index + 1] = (
        elem[0], elem[2], elem[1], elem[3], elem[6], elem[5], elem[4], elem[7], elem[9], elem[8])

mode_results_vol['disp'] = mode_disp_vol

results.append(mode_results_vol)

mvol = {
    'Nodes': volnodes,
    'Seg2Elem': {},
    'Seg3Elem': {},
    'Tria3Elem': {},
    'Tria6Elem': {},
    'Quad4Elem': {},
    'Quad8Elem': {},
    'Tetra4Elem': {},
    'Tetra10Elem': elements_tetra10,
    'Hexa8Elem': {},
    'Hexa20Elem': {},
    'Penta6Elem': {},
    'Penta15Elem': {},
    'Results': results
}

meshvol = itf.make_femmesh(mvol)

result_mesh_object_1 = ObjectsFem.makeMeshResult(doc, 'Result_Mesh_Volume')
result_mesh_object_1.FemMesh = meshvol

numnodes = len(nocoord)
resVol.DisplacementVectors = [App.Vector(dis[3 * n], dis[3 * n + 1], dis[3 * n + 2]) for n in range(numnodes)]
resVol.DisplacementLengths = [np.linalg.norm([dis[3 * n], dis[3 * n + 1], dis[3 * n + 2]]) for n in range(numnodes)]
resVol.NodeStressXX = tet10stress.T[0].T.tolist()
resVol.NodeStressYY = tet10stress.T[1].T.tolist()
resVol.NodeStressZZ = tet10stress.T[2].T.tolist()
resVol.NodeStressXY = tet10stress.T[3].T.tolist()
resVol.NodeStressXZ = tet10stress.T[4].T.tolist()
resVol.NodeStressYZ = tet10stress.T[5].T.tolist()

resVol.Mesh = result_mesh_object_1
resVol.NodeNumbers = [int(key) for key in resVol.Mesh.FemMesh.Nodes.keys()]

resVol = itf.fill_femresult_mechanical(resVol, results)

# VOLUME MESH FINISH

# INTERFACE MESH START
if interface_elements[0][0] != 0:

    intnodes = {}
    mode_disp_int = {}
    intconnect = {}
    newnode = {}
    oldnode = {}
    elements_tria6 = {}
    mode_results_int = {}
    results = []

    index = 0
    for i, intel in enumerate(interface_elements):
        for nd in intel[:6]:
            if nd not in intconnect:
                index += 1
                intconnect[nd] = index
                intnodes[index] = App.Vector(nocoord[nd - 1][0], nocoord[nd - 1][1], nocoord[nd - 1][2])
                mode_disp_int[index] = App.Vector(contactpressurevector[nd - 1][0],
                                                  contactpressurevector[nd - 1][1],
                                                  contactpressurevector[nd - 1][2])
                newnode[nd] = index
                oldnode[index] = nd

        elements_tria6[i + 1] = (newnode[intel[0]], newnode[intel[1]], newnode[intel[2]], newnode[intel[3]],
                                 newnode[intel[4]], newnode[intel[5]])

    mode_results_int['disp'] = mode_disp_int
```

```
        results.append(mode_results_int)

        mint = {
            'Nodes': intnodes,
            'Seg2Elem': {},
            'Seg3Elem': {},
            'Tria3Elem': {},
            'Tria6Elem': elements_tria6,
            'Quad4Elem': {},
            'Quad8Elem': {},
            'Tetra4Elem': {},
            'Tetra10Elem': {},
            'Hexa8Elem': {},
            'Hexa20Elem': {},
            'Penta6Elem': {},
            'Penta15Elem': {},
            'Results': results
        }

        meshint = itf.make_femmesh(mint)

        result_mesh_object_2 = ObjectsFem.makeMeshResult(doc, 'Result_Mesh_Interface')
        result_mesh_object_2.FemMesh = meshint

        resInt.DisplacementVectors = [App.Vector(dis[3 * (oldnode[n + 1] - 1)], dis[3 * (oldnode[n + 1] - 1) + 1],
                                                 dis[3 * (oldnode[n + 1] - 1) + 2]) for n in range(len(oldnode))]
        resInt.DisplacementLengths = [contactpressurevalue[nd - 1] for nd in
                                      oldnode.values()]  # TODO: This is a dirty hack. move contact pressure to its own
result object attribute

        resInt.Mesh = result_mesh_object_2
        resInt.NodeNumbers = [int(key) for key in resInt.Mesh.FemMesh.Nodes.keys()]

        resInt = itf.fill_femresult_mechanical(resInt, results)

    # INTERFACE MESH FINISH

    doc.recompute()

    return resInt, resVol
```