LMGC90v2_Pre : le pré-processeur de LMGC90

A. Martin, M. Bagnéris, F. Dubois, R. Mozul

Laboratoire de Mécanique et Génie Civil

Formation LMGC90 - janvier 2013

Génération d'un objet *mesh*

- "à la main" catalogue :

```
1 m = mesh(dimension=2)
2 m.addNode( node('NO2xx', numpy.array([0.,0.])) )
3 m.addNode( node('NO2xx', numpy.array([1.,0.])) )
4 m.addNode( node('NO2xx', numpy.array([0.,1.])) )
5 m.addNode( node('NO2xx', numpy.array([1.,1.])) )
6 m.addBulk( element('Q4xxx', 4, [1,2,4,3]) )
```

Génération d'un objet *mesh*

- "à la main" catalogue :

```
1 m = mesh(dimension=2)
2 m.addNode( node('NO2xx', numpy.array([0.,0.])) )
3 m.addNode( node('NO2xx', numpy.array([1.,0.])) )
4 m.addNode( node('NO2xx', numpy.array([0.,1.])) )
5 m.addNode( node('NO2xx', numpy.array([1.,1.])) )
6 m.addBulk( element('Q4xxx', 4, [1,2,4,3]) )
```

- maillage généré par un mailleur interne catalogue :

```
1 # maillage en Q4
2 m3 = buildMesh2D(type_mesh='Q4', x0=0., y0=0.,
3     lx=0.1, ly=0.01, nb_elem_x=10, nb_elem_y=1)
```

Génération d'un objet *mesh*

- "à la main" catalogue :

```
1 m = mesh(dimension=2)
2 m.addNode( node('NO2xx', numpy.array([0.,0.])) )
3 m.addNode( node('NO2xx', numpy.array([1.,0.])) )
4 m.addNode( node('NO2xx', numpy.array([0.,1.])) )
5 m.addNode( node('NO2xx', numpy.array([1.,1.])) )
6 m.addBulk( element('Q4xxx', 4, [1,2,4,3]) )
```

- maillage généré par un mailleur interne catalogue :

```
1 # maillage en Q4
2 m3 = buildMesh2D(type_mesh='Q4', x0=0., y0=0.,
3     lx=0.1, ly=0.01, nb_elem_x=10, nb_elem_y=1)
```

- lecture d'un fichier généré par un mailleur externe (*e.g.* Gmsh) catalogue :

```
1 m2 = lecture("2_briques.msh", dim=3)
```

Génération d'objets *avatar* à partir d'objets *mesh*

- génération d'un corps déformable ▸ catalogue :

```
1 av = m.buildMeshedAvatar(model=m2dl, material=stone)
```

Génération d'objets *avatar* à partir d'objets *mesh*

- génération d'un corps déformable catalogue :

```
1 av = m.buildMeshedAvatar(model=m2dl, material=stone)
```

- génération d'un corps rigide
  - avec un contacteur polygone (POLYG) catalogue :

```
1 av2 = rigidFromMesh2D(surfacique_mesh=m2,
2                       model=mod, material=tdur,
3                       color='BLEUx')
```

Génération d'objets *avatar* à partir d'objets *mesh*

- génération d'un corps déformable catalogue :

```
1 av = m.buildMeshedAvatar(model=m2dl, material=stone)
```

- génération d'un corps rigide
  - avec un contacteur polygone (POLYG) catalogue :

```
1 av2 = rigidFromMesh2D(surfacique_mesh=m2,
2                       model=mod, material=tdur,
3                       color='BLEUx')
```

  - avec un contacteur polyèdre (POLYR) catalogue :

```
1 av3 = surfacicMeshToRigid3D(surfacic_mesh=m3,
2                             model=mod3d, material=pdur,
3                             color='BLEUx')
4 # ou
5 av3 = volumicMeshToRigid3D(volumic_mesh=m3,
6                            model=mod3d, material=pdur,
7                            color='BLEUx')
```

Fonctions utilitaires pour les objets maillés

- séparation d'un objet *mesh* en plusieurs objets *mesh* (*e.g.* séparation de pièces d'une collection stockées dans un même fichier) ▸ catalogue :

```
1 # separation du maillage (dictionnaire d'objets mesh)
2 entity2mesh = complete_mesh.separateMeshes(dim=dim,
3     entity_type="physicalEntity", keep_all_elements=True)
4 # traitement des maillages
5 for m in entity2mesh.values()
6     # ...
```

Fonctions utilitaires pour les objets maillés

- séparation d'un objet *mesh* en plusieurs objets *mesh* (*e.g.* séparation de pièces d'une collection stockées dans un même fichier) ▸ catalogue :

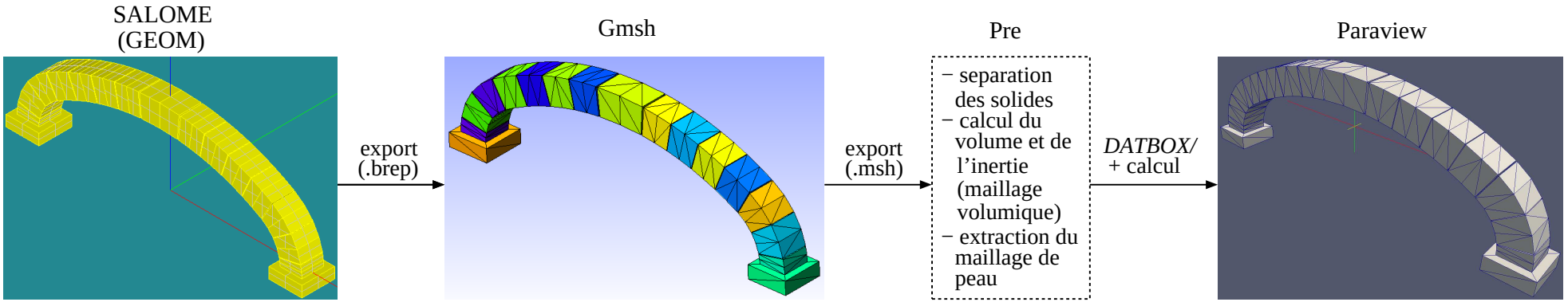
```
# separation du maillage (dictionnaire d'objets mesh)
entity2mesh = complete_mesh.separateMeshes(dim=dim,
    entity_type="physicalEntity", keep_all_elements=True)
# traitement des maillages
for m in entity2mesh.values()
    # ...
```

- éclatement d'un objet *avatar* en plusieurs objets *avatar* pour utiliser des CZM (ajout automatique de contacteurs) ▸ catalogue :

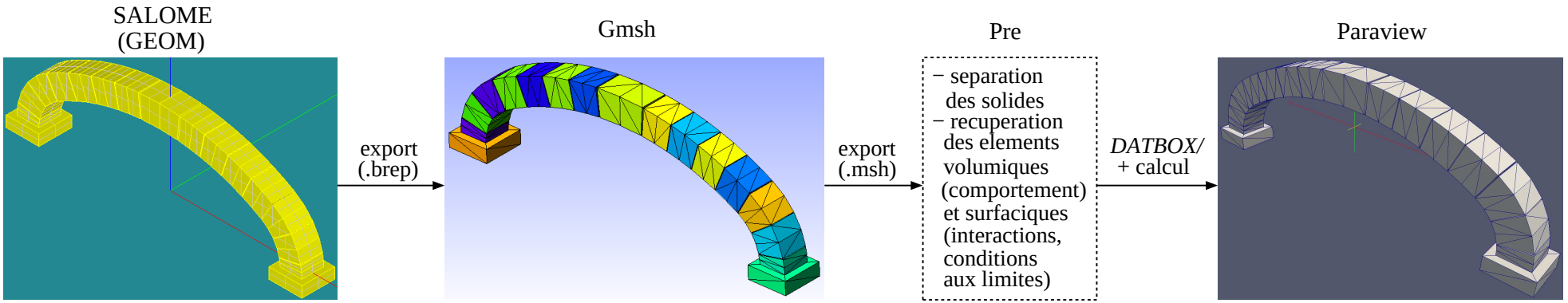
```
# on eclate le corps maille (liste d'objets avatar)
new_bodies = explodeMeshedAvatar2D(body, nbPoints=2,
                                   color='BLEUx')
# traitement des corps resultants
for new_body in new_bodies:
    # ...
```

Utilisation d'un modeleur CAO : chaîne de construction d'objets avatar rgides et déformables

- construction de corps rigides :

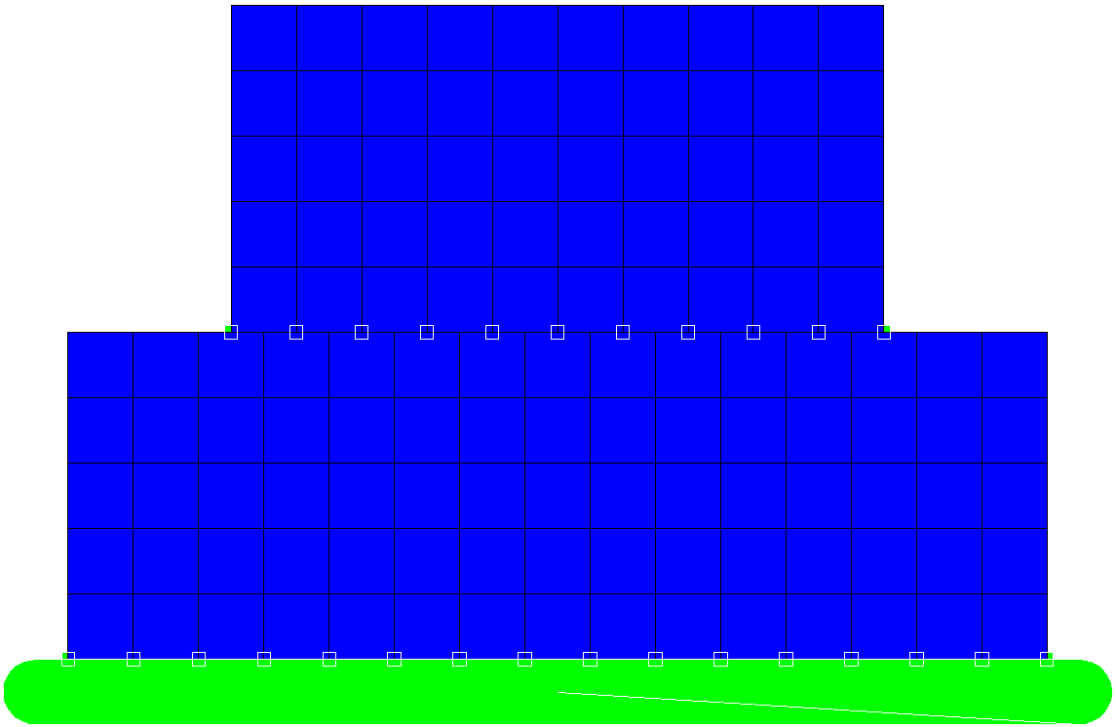


- construction de corps déformables :



Deux blocs

- deux blocs élastiques posés l'un sur l'autre.
- conditions limites :
  - bloc du dessous posé sur une fondation rigide.



Deux blocs : initialisation

```
1  # import des modules
2  import numpy
3  from pre_lmgc import *
4
5  # definition des conteneurs:
6  #   * de corps
7  bodies = avatars()
8  #   * de modeles
9  mods = models()
10 #   * de materiaux
11 mats = materials()
12 #   * de tables de visibilite
13 svs = see_tables()
14 #   * de lois de contact
15 tacts = tact_behavs()
```

Deux blocs : définition des modèles et des matériaux

```
1  # exemple 2D
2  dim = 2
3
4  # definition d'un modele elastique, lineaire, hpp
5  m2Dl = model(name='M2DLx', type='MECAx',
6      element='Q4xxx', dimension=dim,
7      external_model='yes__', kinematic='small',
8      material='elas_', anisotropy='iso__',
9      mass_storage='coher')
10
11 # definition d'un materiau elastique
12 stone = material(name='stone', type='ELAS',
13     density=2750., elas='standard',
14     anisotropy='isotropic', young=7.e10, nu=0.2)
```

Deux blocs : définition des modèles et des matériaux (suite et fin)

```
1  # definition d'un modele rigide
2  mR2D = model(name='rigid', type='MECAx',
3      element='Rxx2D', dimension=dim)
4
5  # definition d'un materiau rigide
6  tdur = material(name='TDURx', type='RIGID',
7      density=2500.)
8
9  # remplissage des conteneurs :
10 #     * de modeles
11 mods.addModel(m2Dl)
12 #     * de materiaux
13 mats.addMaterial(stone)
14 mats.addMaterial(tdur)
```

Deux blocs : définition des corps pour les blocs

```
1  # lecture du maillage du bloc inferieur
2  down_mesh=lecture('gmsh/block.msh', dim)
3  # construction d'un avatar maille a partir du
4  # maillage du bloc inferieur
5  down=down_mesh.buildMeshedAvatar(model=m2Dl,
6      material=stone)
7
8  # contacteurs antagonistes sur le dessus du bloc
9  down.addContactors(group='2', type='ALpxx',
10     color='BLEUx', reverse='yes')
11 # contacteurs candidats sur le dessous du bloc
12 down.addContactors(group='1', type='CLxxx',
13     color='BLEUx', reverse='yes')
14
15 # ajout du bloc dans le conteneur de corps
16 bodies.addAvatar(down)
```

Deux blocs : définition des corps pour les blocs (suite et fin)

```
1  # generation du maillage du bloc superieur
2  up_mesh=buildMesh2D(type_mesh='Q4', x0=0.025, y0=0.05,
3      lx=0.1, ly=0.05, nb_elem_x=10, nb_elem_y=5)
4
5  # construction d'un avatar maille a partir du
6  # maillage du bloc superieur
7  up=up_mesh.buildMeshedAvatar(model=m2Dl,
8      material=stone)
9
10 # contacteurs candidats sur le dessous du bloc
11 up.addContactors(group='down', type='CLxxx',
12     color='BLEUx')
13
14 # ajout du bloc dans le conteneur de corps
15 bodies.addAvatar(up)
```

Deux blocs : définition de la fondation

```
1  # creation d'un nouvel avatar rigide pour la fondation
2  floor = avatar(type='RBDY2', dimension=dim)
3
4  # ajout d'un element rigide a la fondation
5  floor.addBulk( rigid2d() )
6  # ajout d'un noeud a la fondation
7  floor.addNode( node(type='NO2xx',
8      coor=numpy.array([0.075, -0.005]), number=1) )
9  # definition des groupes (seulement 'all')
10 floor.defineGroups()
11
12 # affectation du modele a la fondation
13 floor.defineModel(group='all', model=mR2D)
14 # affectation du materiau a la fondation
15 floor.defineMaterial(group='all', material=tdur)
```

Deux blocs : définition de la fondation (suite et fin)

```
1  # ajout du contacteur jonc a la fondation
2  floor.addContactors(group='all', type='JONCx',
3      color='FLOOR', axe1=0.08, axe2=0.005)
4  # calcul de la surface et de l'inertie de la fondation
5  floor.computeRigidProperties()
6
7  # condition limites : fondation bloquee
8  floor.imposeDrivenDof(component=[1, 2, 3],
9      dofty='vlocy')
10
11 # ajout de la fondation au conteneur de corps
12 bodies.addAvatar(floor)
```

Deux blocs : définition des contacts envisagés

```
1  # definition d'une loi de contact frottant
2  gapc0=tact_behav(name='gapc0', type='GAP_SGR_CLB',
3      fric=0.3)
4  # ajout de la loi dans le conteneur de lois
5  tacts.addBehav(gapc0)
6
7  # definition d'une table de visibilite pour le
8  # contact ligne-ligne (i.e. entre blocs)
9  sv = see_table(CorpsCandidat='MAILx', candidat='CLxxx',
10 colorCandidat='BLEUx', behav=gapc0,
11 CorpsAntagoniste='MAILx', antagoniste='ALpxx',
12 colorAntagoniste='BLEUx', alert=0.001)
13 # ajout de la table de visibilite dans le conteneur
14 # de tables de visibilite
15 svs.addSeeTable(sv)
```

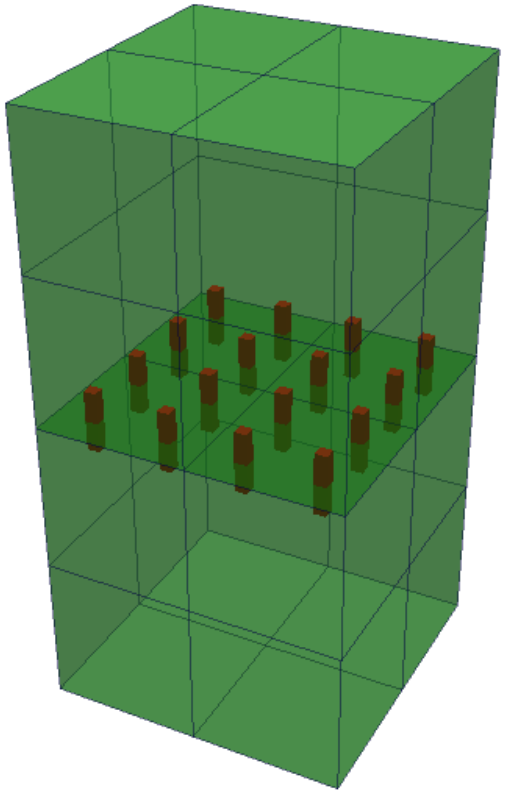
Deux blocs : définition des contacts envisagés (suite et fin)

```
1  # definition d'une loi de contact frottant
2  gapc1=tact_behav(name='gapc1', type='GAP_SGR_CLB',
3      fric=0.5)
4  # ajout de la loi dans le conteneur de lois
5  tacts.addBehav(gapc1)
6
7  # definition d'une table de visibilite pour le
8  # contact ligne-jonc (i.e. avec la fondation)
9  sv = see_table(CorpsCandidat='MAILx', candidat='CLxxx',
10 colorCandidat='BLEUx', behav=gapc1,
11 CorpsAntagoniste='RBDY2', antagoniste='JONCx',
12 colorAntagoniste='FLOOR', alert=0.001)
13 # ajout de la table de visibilite dans le conteneur
14 # de tables de visibilite
15 svs.addSeeTable(sv)
```

Deux blocs : écriture des fichiers de données

```
1  # ecriture des fichiers de donnees pour LMGC90
2  #   * les modeles
3  writeModels(mods, chemin='./DATBOX/')
4  #   * les materiaux
5  writeBulkBehav(mats, chemin='./DATBOX/', dim=dim)
6  #   * les corps
7  writeBodies(bodies, chemin='./DATBOX/')
8  #   * les conditions aux limites
9  writeDrvDof(bodies, chemin='./DATBOX/')
10 #   * les contacts envisages
11 writeTactBehav(tacts, svs, chemin='./DATBOX/')
12 #   * les conditions initiales (cinematique, valeurs
13 # aux points d'integration, contacts)
14 writeDofIni(bodies, chemin='./DATBOX/')
15 writeGPVIni(bodies, chemin='./DATBOX/')
16 writeVlocRlocIni(chemin='./DATBOX/')
```

Deux cubes



- deux cubes élastiques posés l'un sur l'autre.
- conditions limites :
  - non-pénétration de la fondation : $v_z = 0$ en $z = 0$.

Deux cubes : génération des maillages

```
1  # import des modules
2  import numpy as np
3  import copy
4  from pre_lmgc import *
5
6  # declarations (dimension, materiau)
7
8  # definition d'un modele elastique, lineaire, hpp
9  m3Dl = model(name='M3DLx', type='MECAx',
10     element='H8xxx', dimension=dim,
11     external_model='yes__', kinematic='small',
12     material='elas_', anisotropy='iso__',
13     mass_storage='lump')
14
15 # remplissages des conteneurs (modeles, materiaux)
```

Deux cubes : définition des corps pour les cubes

```
1  # generation du maillage du cube superieur
2  down_mesh=buildMeshH8(x0=0., y0=0., z0=0., lx=1.,
3      ly=1., lz=1., nb_elem_x=2, nb_elem_y=2, nb_elem_z=2)
4  # copie du maillage pour construire le cube inferieur
5  up_mesh=copy.deepcopy(down_mesh)
6  # construction d'un avatar maille a partir du
7  # maillage du cube inferieur
8  down=down_mesh.buildMeshedAvatar(model=m3DI,
9      material=stone)
10 # contacteurs antagonistes sur le dessus du cube
11 down.addContactors(group='up', type='ASpx4',
12     color='BLEUx')
13 # condition de non penetration de la fondation
14 down.imposeDrivenDof(group='down', component=3,
15     dofty='vlocy')
16 # ajout du bloc dans le conteneur de corps
17 bodies.addAvatar(down)
```

Deux cubes : définition des corps pour les cubes (suite)

```
# construction d'un avatar maille a partir du
# maillage du cube superieur
up=up_mesh.buildMeshedAvatar(model=m3Dl,
    material=stone)
# contacteurs candidats sur le dessous du cube
#    * coordonnees des Points de Gauss (regle a 4 PG)
GP_coor=numpy.array( \
    [[-1./math.sqrt(3), -1./math.sqrt(3)], \
     [+1./math.sqrt(3), -1./math.sqrt(3)], \
     [+1./math.sqrt(3), +1./math.sqrt(3)], \
     [-1./math.sqrt(3), +1./math.sqrt(3)]])
#    * fonction de forme du Q4 de reference
def N(kez):
    ksi = kez[0]; eta = kez[1]
    return np.array([0.25*(1. - ksi)*(1. - eta), \
                     0.25*(1. + ksi)*(1. - eta), \
                     0.25*(1. + ksi)*(1. + eta), \
                     0.25*(1. - ksi)*(1. + eta)])
```

Deux cubes : définition des corps pour les cubes (suite et fin)

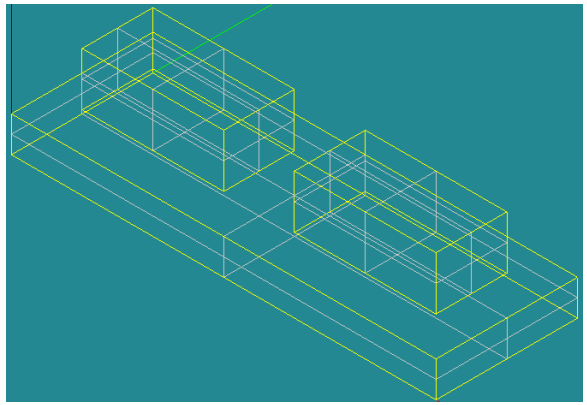
```
#     * poids associes aux points de contact
weights = numpy.zeros([4, 4], 'd')
weights[0, 0 : 4] = N(GP_coor[0])
weights[1, 0 : 4] = N(GP_coor[1])
weights[2, 0 : 4] = N(GP_coor[2])
weights[3, 0 : 4] = N(GP_coor[3])
#     * affectation des points de contact
up.addContactors(group='down', type='CSxx4',
    weights=weights, color='BLEUx')

# positionnement du cube
up.translate(dz=1.)

# ajout du cube dans le conteneur de corps
bodies.addAvatar(up)
```

Deux cubes : définition des contacts envisagés

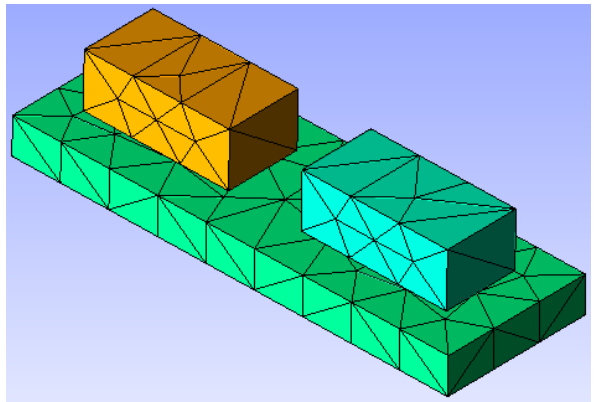
```
1  # definition de la loi d'interaction (contact-frottant)
2
3  # definition d'une table de visibilite pour le
4  # contact surface-surface (i.e. entre cubes)
5  sv = see_table(CorpsCandidat='MAILx', candidat='CSxxx',
6  colorCandidat='BLEUx', behav=gapc0,
7  CorpsAntagoniste='MAILx', antagoniste='ASpxx',
8  colorAntagoniste='BLEUx', alert=0.1, halo=0.5)
9  # ajout de la table de visibilite dans le conteneur
10 # de tables de visibilite
11 svs.addSeeTable(sv)
12
13 # remplissages des conteneurs (lois d'interactions,
14 # tables de visibilite) + ecriture des fichiers
```

Deux briques : une rigide et une déformable

- géométrie (GEOM) :
  - brique : primitive "Box" ("Dx"=0.4, "Dy"=0.2, "Dz"=0.15),
  - fondation : primitive "Box" ("Dx"=1.2, "Dy"=0.4, "Dz"=0.1),
  - $\Rightarrow$ fichier "2_briques.brep",



- maillage (Gmsh) :
  - commande : "gmsh -3 -algo netgen -optimize -clmin 0.1 -clmax 0.1 2_briques.brep",
  - $\Rightarrow$ fichier "2_briques.msh".



Deux briques : initialisation

```
1  # import du module
2  from pre_lmgc import *
3
4  # definition des conteneurs:
5  #   * de corps
6  bodies = avatars()
7  #   * de modeles
8  mods = models()
9  #   * de materiaux
10 mats = materials()
11 #   * de tables de visibilite
12 svs = see_tables()
13 #   * de lois de contact
14 tacts = tact_behavs()
15 #   * de commandes de post-traitement
16 post = postpro_commands()
```

Deux briques : définition des modèles

```
1  # exemple 3D
2  dim = 3
3
4  # definition d'un modele elastique, lineaire, hpp
5  m3Dl = model(name='M3DLx', type='MECAx',
6      element='TE4xx', dimension=dim,
7      external_model='yes__', kinematic='small',
8      material='elas_', anisotropy='iso__',
9      mass_storage='lump_')
10 # ajout du modele au conteneur de modeles
11 mods.addModel(m3Dl)
12
13 # definition d'un modele rigide
14 mR3D = model(name='rigid', type='MECAx',
15     element='Rxx3D', dimension=dim)
```

Deux briques : définition des matériaux

```
1  # definition d'un materiau elastique
2  stone = material(name='stone', type='ELAS',
3      density=1800., elas='standard',
4      anisotropy='isotropic', young=1.e10, nu=0.15)
5
6  # definition d'un materiau rigide
7  tdur = material(name='TDURx', type='RIGID',
8      density=2500.)
9
10 # remplissage du conteneur de materiaux
11 mats.addMaterial(stone)
12 mats.addMaterial(tdur)
```

Deux briques : lecture du maillage

```
1  # lecture du maillage des trois objets
2  complete_mesh = lecture(name='gmsh/2_briques.msh',
3      dim=dim)
4
5  # separation des maillages volumiques des trois
6  # solides : les deux briques et la fondation
7  entity2mesh=complete_mesh.separateMeshes(dim=dim,
8      entity_type="physicalEntity", keep_all_elements=True)
9
10 # recuperation du maillage
11 #     * de la fondation
12 mesh_floor=entity2mesh["dalle"]
13 #     * de la brique rigide
14 mesh_rigid_brick=entity2mesh["brique_rigide"]
15 #     * de la brique deformable
16 mesh_deformable_brick=entity2mesh["brique_deformable"]
```

Deux briques : définition des corps pour les briques

```
1  # construction d'un corps deformable a partir du
2  # maillage de la brique deformable
3  deformable_brick = \
4      mesh_deformable_brick.buildMeshedAvatar(model=m3DI,
5      material=stone)
6  # contacteur candidat sur le dessous de la brique
7  deformable_brick.addContactors(group='10',
8      type='CSxx3', color='REDxx')
9
10 # construction d'un corps rigide a partir du maillage
11 # de la brique rigide
12 rigid_brick = volumicMeshToRigid3D(volumic_mesh= \
13     mesh_rigid_brick, model=mR3D, material=stone,
14     color='REDxx')
```

Deux briques : définition d'un corps pour la fondation

```
1  # construction d'un corps rigide a partir du maillage
2  # de la fondation
3  floor = volumicMeshToRigid3D(volumic_mesh=mesh_floor,
4      model=mR3D, material=tdur, color='FLOOR')
5
6  # conditions limites : fondation bloquee
7  floor.imposeDrivenDof(component=[1, 2, 3, 4, 5, 6],
8      dofty='vlocy')
9
10 # rempissage du conteneur de coprs
11 bodies.addAvatar(rigid_brick)
12 bodies.addAvatar(deformable_brick)
13 bodies.addAvatar(floor)
```

Deux briques : définition des lois de contact

```
1  # definition de deux lois de contact frottant :
2  #     * pour le contact "brique deformable"-fondation
3  gapG0=tact_behav(name='gapG0', type='GAP_SGR_CLB_g0',
4      fric=0.5)
5  #     * pour le contact "brique rigide"-fondation
6  iqsG0=tact_behav(name='iqsG0', type='IQS_CLB_g0',
7      fric=0.3)
8
9  # ajout des lois dans le conteneur de lois
10 tacts.addBehav(gapG0)
11 tacts.addBehav(iqsG0)
```

Deux briques : définition des tables de visibilité

```
# definition des tables de visibilite pour le
# contact "brique deformable"-fondation
svdr = see_table(CorpsCandidat='MAILx',candidat='CSxxx',
colorCandidat='REDxx', behav=gapG0,
CorpsAntagoniste='RBDY3', antagoniste='POLYR',
colorAntagoniste='FLOOR', alert=2.5e-2, halo=1.e0)
# definition des tables de visibilite pour le
# contact "brique rigide"-fondation
svrr = see_table(CorpsCandidat='RBDY3',candidat='POLYR',
colorCandidat='REDxx', behav=iqsG0,
CorpsAntagoniste='RBDY3', antagoniste='POLYR',
colorAntagoniste='FLOOR', alert=2.5e-2)
# ajout des tables de visibilite dans le conteneur
# de tables de visibilite
svs.addSeeTable(svrr); svs.addSeeTable(svdr)
```

Deux briques : définition des commandes de post-traitement

```
1  # definition du set pour suivre la brique deformable :
2  #   * defintion du set
3  mecax_set = [(body_deformable_brick, "coin")]
4  #   * creation de la commande
5  deformable_brick_set = \
6      postpro_command(type='NEW_MECAx_SETS',
7      mecax_sets=[mecax_set])
8
9  # ajout de la commande au conteneur de commandes
10 post.addCommand(deformable_brick_set)
```

Deux briques : définition des commandes de post-traitement (suite)

```
1  # suivi de la brique deformable :
2  #   * cinematique de la brique
3  deformable_brick_disp = \
4      postpro_command(type='Dep_EVOLUTION',
5      step=1)
6  #   * efforts subis par la brique
7  deformable_brick_fint = \
8      postpro_command(type='Fint_EVOLUTION',
9      step=1)
10
11 # ajout des commandes au conteneur de commandes
12 post.addCommand(deformable_brick_disp)
13 post.addCommand(deformable_brick_fint)
```

Deux briques : définition des commandes de post-traitement (suite et fin)

```
1  # suivi de la brique rigide :
2  #   * cinematique de la brique
3  rigid_brick_disp = \
4      postpro_command(type='BODY_TRACKING', step=1,
5      rigid_set=[rigid_brick])
6  #   * efforts subis par la brique
7  rigid_brick_torque = \
8      postpro_command(type='TORQUE_EVOLUTION', step=1,
9      rigid_set=[rigid_brick])
10
11 # ajout des commandes au conteneur de commandes
12 post.addCommand(rigid_brick_disp)
13 post.addCommand(rigid_brick_torque)
```

Deux briques : écriture des fichiers

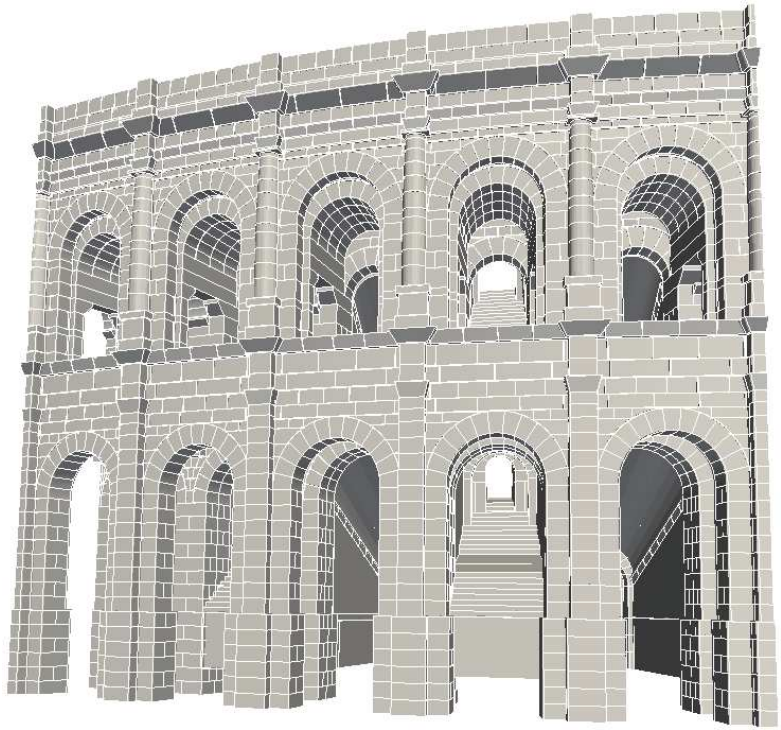
```
1 # ecriture des fichiers de donnees pour LMGC90
2 #   * les modeles
3 writeModels(mods, chemin='./DATBOX/')
4 #   * les materiaux
5 writeBulkBehav(mats, chemin='./DATBOX/', dim=dim)
6 #   * les corps
7 writeBodies(bodies, chemin='./DATBOX/')
8 #   * les contacts envisages
9 writeTactBehav(tacts, svs, chemin='./DATBOX/')
```

Deux briques : écriture des fichiers (suite et fin)

```
1 #   * les deplacements et vitesses initiaux
2 writeDofIni(bodies, chemin='./DATBOX/')
3 #   * les valeurs aux points de d'integration initiales
4 writeGPVIni(bodies, chemin='./DATBOX/')
5 #   * les conditions aux limites
6 writeDrvDof(bodies, chemin='./DATBOX/')
7 #   * les commandes de post-traitement
8 writePostpro(post, bodies, path='DATBOX/')
```

Outils utilisant le module Python de Salome

- module fournissant des fonctions d'export direct des solides en corps rigides : volume et inertie calculés par le modeleur CAO + enveloppe discrétisée par Gmsh,
- import de fichiers DXF d'AutoCAD (maillages d'enveloppes de solides),
- représentation de formes paramétrées pFormes (M. Bagneris).



Meshes

1. generation
   1.1 lecture
   1.2 buildMesh2D
   1.3 buildMeshH8
2. *mesh* objects
   2.1 constructor
   2.2 addNode
   2.3 addBulk
   2.4 buildMeshedAvatar
   2.5 separateMeshes
3. manipulation of *mesh* objects
   3.1 extractFreeSurface
   3.2 reorientSurfacicElement

Meshes generation

lecture

Read a mesh in a file. Supported file format are *gmsh* and *sysweld*

$$my\_mesh = lecture(name, dim, keep\_elements = [], scale\_factor = None)$$

where :

- name (input string), name of file to read
- dim (input integer), dimension
- keep_elements (optional string list), to keep only a subset of element types
- scale_factor (optional double), to rescale the read mesh
- my_mesh (returned mesh object), the read mesh

Meshes generation

buildMesh2D

Mesh a rectangle with a group for each boundary line (*'left'*, *'down'*, *'right'*, *'up'*)

$$my\_mesh = buildMesh2D(type\_mesh, x0, y0, lx, ly, nb\_elem\_x, nb\_elem\_y, vertices = [], number = None)$$

where :

- type_mesh (input string '2T3', '4T3', 'Q4', 'Q8'), the type of element to use to mesh the rectangular
- x0 (input double), left corner position
- y0 (input double), lower corner position
- lx (input double), dimension of the rectangle along x-axis
- ly (input double), dimension of the rectangle along y-axis
- nb_elem_x (input integer), number of elements along x-axis
- nb_elem_y (input integer), number of elements along y-axis
- vertices (optional double array), a list of x,y-coordinates following a suitable Q4 mesh node ordering
- number (optional input integer), number of avatar
- my_mesh (returned mesh object), the generated mesh

Meshes generation

buildMeshH8

Mesh a parallelepiped using hexaedric element. Group for each boundary line (*'left'*, *'down'*, *'right'*, *'up'*, *front*, *rear*) are also defined.

$$my\_mesh = buildMeshH8(x0, y0, z0, lx, ly, lz, nb\_elem\_x, nb\_elem\_y, nb\_elem\_z, surfacic\_mesh\_type = 'Q4')$$

where :

- x0 (input double), left corner position
- y0 (input double), lower corner position
- z0 (input double), rear corner position
- lx (input double), dimension of the rectangle along x-axis
- ly (input double), dimension of the rectangle along y-axis
- lz (input double), dimension of the rectangle along z-axis
- nb_elem_x (input integer), number of elements along x-axis
- nb_elem_y (input integer), number of elements along y-axis
- nb_elem_z (input integer), number of elements along z-axis
- surfacic_mesh_type (optional string 'Q4' or '2T3'), the element type to use on surfaces
- my_mesh (returned mesh object), the generated mesh

mesh objects

Constructor

Create an empty mesh object

$$my\_mesh = mesh(dimension)$$

where :

- dimension (input integer), the dimension of the mesh
- my_mesh (returned mesh object), the new mesh empty mesh object

mesh objects

addNode

Add a node to a mesh.

$$body = my\_mesh.addNode(noeud)$$

where :

- my_mesh, is the mesh object in which to add the node
- noeud (input node object), the node to add to my_mesh

mesh objects

addBulk

Add an element to a mesh.

$$body = my\_mesh.addBulk(ele)$$

where :

- my_mesh, is the mesh object in which to add the node
- ele (input bulk object), the bulk to add to my_mesh

mesh objects

buildMeshedAvatar

Building a meshed avatar. Contactors have to be added afterward to the body.

$$body = my\_mesh.buildMeshedAvatar(model, material)$$

where :

- my_mesh, is the mesh object to build the avatar from
- model (input model object), the model of the elements
- material (input material object), the material of the elements
- body (returned avatar object), the generated avatar

mesh objects

separateMeshes

When reading a file, only one mesh object is returned, even if there are several meshes in the file. This function allow to get back the mesh objects corresponding to the submeshes (in a Python dictionnary).

$$meshes = my\_mesh.separateMeshes(dim, entity\_type = 'geometricalEntity', keep\_all\_elements = True)$$

where :

- my_mesh, is the mesh object to build the avatar from
- dim (input integer), dimension depending on the type of mesh (surfacic or volumic)
- entity_type (optional input string 'geometricalEntity' or 'physicalEntity'), select according which type to separate the meshes
- keep_all_elements (optional input boolean), if all elements are to be kept or just those of the input dimension (dim)
- meshes (returned dictionnary), a dictionnary of mesh objects where keys are the entity_type names

manipulation of *mesh* objects

extractFreeSurface

Create a surfacic mesh corresponding to the free surface of a volumic mesh. The elements of the volumic mesh must be tetrahdra.

$$surfacic\_mesh = extractFreeSurface(volumic\_mesh)$$

where :

- volumic_mesh (input mesh object), a mesh object in 3D
- surfacic_mesh (returned mesh object), a mesh object in 2D

manipulation of *mesh* objects

reorientSurfacicElements

Reorient the surfacic elements of surfacic elements of a 3D mesh using the orientation of its volumic elements. The elements of the volumic mesh must be tetrahdra.

$$reorientSurfacicElements(volumic\_mesh)$$

where :

- volumic_mesh (input mesh object), the 3D mesh object to reorient

Meshed bodies generation

1. 2D avatars from mesh
   1.1 rigidFromMesh2D
   1.2 rigidsFromMesh2D
2. 2D avatars from meshed avatar
   2.1 explodeMeshedAvatar2D
3. 3D avatars from mesh
   3.1 volumicMeshToRigid3D
   3.2 surfacicMeshToRigid3D
   3.3 surfacicMeshesToRigid3D

2D avatars from mesh

rigidFromMesh2D

Create a rigid avatar using a mesh to define its geometric boundary.

$$body = rigidFromMesh2D(surfacic_mesh, model, material, color = 'BLEUx', reverse = False)$$

where :

- surfacic_mesh (input mesh object), the 2D mesh object to use to generate the rigid avatar
- model (input model object), rigid model for the particle
- material (input material object), material of the particle
- color (optional input string), color of the POLYG contactors (5 characters string)
- reverse (optional input boolean), to specify if the elements need to be reversed
- body (returned avatar object), the rigid avatar with each element of surfacic_mesh being a POLYG contactor

2D avatars from mesh

rigidsFromMesh2D

Create an avatar containers of rigid avatars from the elements of a mesh.

$$bodies = rigidFromMesh2D(surfacic_mesh, model, material, color = 'BLEUx', reverse = False)$$

where :

- surfacic_mesh (input mesh object), the 2D mesh object to use to generate the rigid avatar
- model (input model object), rigid model for the particle
- material (input material object), material of the particle
- color (optional input string), color of the POLYG contactors (5 characters string)
- reverse (optional input boolean), to specify if the elements need to be reversed
- bodies (returned avatar container), the rigid avatars corresponding to each element of surfacic_mesh with a POLYG contactor

2D avatars from meshed avatar

expledMeshedAvatar2D

Create an avatar container of meshed avatars each corresponding to one element of a 2D avatar meshed with elements of the first order.

$$bodies = explodeMeshedAvatar2D(body, nbPoints = 2, color = 'BLEUx') :$$

where :

- body (input avatar object), a 2D avatar of a meshed body
- nbPoints (optional input integer), number of points to put on candidat contactor line
- color (optional input string), color of the contactors (5 characters string)
- bodies (returned avatar container), the meshed avatars each corresponding to an element of the input meshed avatar *body*

3D avatars from mesh

volumicMeshToRigid3D

Create a 3D rigid avatar from a mesh object by extracting its surfacic mesh and computing its mass and inertia. The elements of the mesh must be tetrahedra.

$$body = volumicMeshToRigid3D(volumic\_mesh, model, material, color = 'BLEUx')$$

where :

- volumic_mesh (input mesh object), a volumic mesh object
- model (input model object), a rigid model
- material (input material object), the material of the avatar
- color (optional input string), the color of the POLYR contactor (5 characters string)
- body (returned avatar object), a mesh rigid object with a POLYR contactor corresponding to the surfacic mesh of the input volumic_mesh

3D avatars from mesh

surfacicMeshToRigid3D

Create a 3D rigid avatar from a surfacic mesh object in 3D by computing its mass and inertia. The elements of the mesh must be triangles.

$$body = surfacicMeshToRigid3D(surfacic\_mesh, model, material, color = 'BLEUx')$$

where :

- surfacic_mesh (input mesh object), a surfacic mesh object
- model (input model object), a rigid model
- material (input material object), the material of the avatar
- color (optional input string), the color of the POLYR contactor (5 characters string)
- body (returned avatar object), a mesh rigid object with a POLYR contactor corresponding to the input surfacic_mesh

3D avatars from mesh

surfacicMeshesToRigid3D

Create a 3D rigid avatar from a list of surfacic mesh object in 3D. The elements of the meshes must be triangles.

$$body = surfacicMeshesToRigid3D(surfacic\_meshes, model, material, color = 'BLEUx')$$

where :

- surfacic_mesh (input list of mesh objects), a list of surfacic mesh objects
- model (input model object), a rigid model
- material (input material object), the material of the build avatar
- color (optional input string), the color of the POLYR contactor (5 characters string)
- body (returned avatar object), a mesh rigid object with POLYF contactors corresponding to the input surfacic_meshes