Development in code_aster
Finite elements – advanced

Code_Aster, Salome-Meca course material
GNU FDL licence (http://www.gnu.org/copyleft/fdl.html)
Prerequisites

• To understand this course:
  • code_aster skills - mandatory:
    • Create and execute a code_aster computation (Code_Aster and Salome-Meca course material – Basic training)
    • General organization of code_aster (Code_Aster and Salome-Meca course material – Development training)
    • How to compile code_aster (Code_Aster and Salome-Meca course material – Development training)
  • code_aster skills - optional:
    • The fields in code_aster (Code_Aster and Salome-Meca course material – Development training)
  • Computer science skills:
    • Python (elementary)
  • Science skills:
    • The finite element method: variational forms, theory of approximation
Goals

• Understand general organization for elementary computation in code_aster
• How to define a new finite element in code_aster
• How to modify some elements (integration schemes, ...)
• How to add new elementary computation
Planning

- Introduction
- Geometric elements
- Physical quantities
- Attributes
- Parameters
- Options
- Modelisations
- Finite element
Introduction

• Definition of a finite element in Python catalogs:
  • Geometric element: geometric cell, shape functions and integration scheme for geometric quantities
  • Finite element: select physics and unknowns, shape functions and integration schemes for physical quantities
Introduction

• Definition of a finite element in Python catalogs:
  • Commons catalogs ($ASTER_ROOT/catalo/cataelem/Commons/* .py)
    • Geometric elements
    • Physical quantities
    • Input/output parameters
    • Attributes
    • Modelisations
  • Variational form to compute ($ASTER_ROOT/catalo/cataelem/Options/* .py)
  • Finite element ($ASTER_ROOT/catalo/cataelem/Elements/* .py)
Introduction

Geometric elements

Finite elements

Physical quantities

- mesh_types.py
- finite_element.py
- Physical_quantities.py
- Located_components.py
- parameters.py
- options.py
- phenomenons_modelisations.py

\[ \int_{\Omega} \sigma(u) : \epsilon(\tilde{u}) \, d\Omega \]

\[ \int_{\Omega} f(u) : \tilde{u} \, d\Omega \]

8 - Code_Aster and Salome-Meca course material

GNU FDL Licence
GEOMETRIC ELEMENTS
Geometric elements – General definition

The `mesh_types.py` catalog

$ASTER_ROOT/catalo/cataelem/Commons/mesh_types.py

The diagram shows a geometric cell added to an integration scheme to form a geometric element.
Geometric elements – General definition

- Only one geometric cell but several geometric elements because of different integration schemes

<table>
<thead>
<tr>
<th>Geometric element (elrefe): 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Geometric cell</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Geometric element (elrefe): 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Geometric cell</td>
</tr>
</tbody>
</table>
Geometric elements – General definition

• The « real » geometric cell => must been generated by external mesh program (Salome/Mesh) and can been « post-processed » (Salome/Paravis) + MED format support
Geometric elements – General definition

- The « virtual » geometric cell => for internal use in Code_Aster’s command
Remarks:
• Don’t add a new « real » geometric cell except if you can read mesh and export mesh for post-processing!
• You can easily add new/modify integration schemes
Geometric elements – In Python catalogs

• How to define geometric element (mesh_types.py):

```python
TRIA7 = MeshType(nbno=7, dim=2, code='TR7')

! First geometric element (dedicated FPG1 scheme)
MEC3TR7H = Elrefe()
MEC3TR7H.addLocation('FPG1', 7)
TRIA7.addElrefe(MEC3TR7H)

! Second geometric element (dedicated FPG1 scheme)
TR7 = Elrefe()
TR7.addLocation('NOEU', 7)
TR7.addLocation('FPG1', 1)
...
TRIA7.addElrefe(TR7)
```

• Define geometric cell with general properties: MeshType()
• Instantiation of geometric element (elrefe): Elrefe()
• Add integration schemes (several are possible): addLocation()
• Define geometric element (elrefe): addElrefe()
Geometric elements – In Python catalogs

- How to define **geometric cell** (`mesh_types.py`):

```python
TRIA3 = MeshType(nbno=3, dim=2, code='TR3')
```

- Properties:
  - **Number of nodes**: `nbno`
  - **Topological dimension**: `dim (0, 1, 2, or 3)`
  - **Internal code**: `code`

All real geometric cells in code_aster:

Geometric elements – In Python catalogs

• How to define *integration scheme* (*mesh_types.py*):

```python
TR7.addLocation('NOEU', 7)
```

• Properties:
  • Name: ‘FPG1’
  • Number of integration points: 7

• Remarks:
  • All integration schemes are in : R3.01.01
  • Integration point, weights are defined in Fortran subroutine: see elraga.F90
PHYSICAL QUANTITIES
Physical quantities – Definition

- The physical quantity define data on given support (node, element, Gauss point, etc.)
- The properties of physical quantity:
  - The name (less the 8 characters)
  - The general type:
    - Integer: ‘I’
    - Real: ‘R’
    - Complex: ‘C’
    - String (with length): ‘K8’, ‘K16’, ‘K24’
  - The names of components (less the 8 characters)
  - The storing:
    - A list: used for fields
    - An array: used for vector or matrix
  - The support:
    - On element: ‘ELEM’
    - On nodes for each element: ‘ELNO’
    - On integration point: ‘ELGA’
Physical quantities – Definition

• Definition in two steps:
  • Global definition (for all finite elements in code_aster) with:
    • Name
    • General type (real, integer, …)
    • List of components (complete)
    • Storing (list or array)
  • Local definition (for specific finite element) with:
    • List of components (restricted)
    • Given support (ELEM, ELNO, ELGA)

• Remark:
  • As some local definitions are the same for several finite elements, a generic catalog exists to define them (located_components.py)
Physical quantities – Catalogs

- **Catalog for *global* physical quantities:**
  - This catalog contains the set of **all possible physical quantities with all the components for all finite elements** in code_aster
  - $ASTER_ROOT/catalo/cataelem/Commons/physical_quantities.py

- **Catalog for *generic* physical quantities:**
  - This catalog contains **the most used dedicated physical quantities** to avoid redefinition in finite element catalog
  - $ASTER_ROOT/catalo/cataelem/Commons/located_components.py

- **Local dedicated physical quantities:**
  - $ASTER_ROOT/catalo/cataelem/Elements/*.py
Physical quantities – Catalogs

**Global definition:** `physical_quantities.py`

**Generic (restriction + support):** `located_components.py`

**Local (restriction + support):** `./Elements/*.py`
Physical quantities – Definition

• The physical quantity is known as `GRANDEUR` keyword in command file

• The physical quantity on given support is known as `NOM_CHAM` keyword in command file

• A component in physical quantity is known by `NOM_CMP` keyword in command file
Physical quantities – Definition

• Some examples:
  • Unknowns for mechanics at nodes are displacements and rotations:
    • `GRANDEUR = 'DEPL_R'`
    • `NOM_CMP = ('DX', 'DY', 'DZ', 'DRX', 'DRY', 'DRZ')`
    • `NOM_CHAM = 'DEPL' (and not (DEPL_NOEU !)`
  • Unknown for thermics at nodes is temperature:
    • `GRANDEUR = 'TEMP_R'`
    • `NOM_CMP = 'TEMP'`
    • `NOM_CHAM = 'TEMP' (and not (TEMP_NOEU !)`
  • Unknowns for mechanics at Gauss points are stresses:
    • `GRANDEUR = 'SIEF_R'`
    • `NOM_CMP = ('SIXX', 'SIYY', 'SIZZ', 'SIXZ', 'SIXY', 'SIZZ')`
    • `NOM_CHAM = 'SIEF_ELGA'`
Physical quantities – Definition

- Example of definition of physical quantity (*physical_quantities.py*)

```python
CACABL = PhysicalQuantity(
    type='R',
    components=(
        'SECT',
        'TENS',
    ),
    comment='"""" CACABL Type:R Caracteristiques des cables
    SECT : section du cable
    TENS : tension initiale
    """
)
```

- Name of physical quantity (*GRANDEUR*): CACABL
- Type for all components: `R` (real)
- It’s a *list* of two components (*NOM_CMP*): SECT and TENS
Physical quantities – Definition

- As components and comment are Python’s objects (list for example), we can defined them outside physical quantity object

```python
list_cmp = ('SECT', 'TENS',)
comm_cmp = """CACABL Type:R Caracteristiques des cables
SECT : section du cable
TENS : tension initiale""

CACABL = PhysicalQuantity(
    type = 'R',
    components = list_cmp,
    comment = comm_cmp)
```
Physical quantities – Special

- Special: *dynamic* physical quantity
  - Used when the number of components cannot been defined *a priori* in physical quantities catalog (dynamic construction, number is defined at execution)
  - Most used: internal variables for non-linear behaviour
  - The component has special name: ‘VARI’
  - The components (NOM_CMP) are automatically named V1, V2, V3, ...

```
VARI_R   = PhysicalQuantity(
    type      = 'R',
    components = ('VARI',),
    comment    = """Variables internes pour les lois de comportement""
)
```
Physical quantities – Special

• Special: *compact syntax* for list
  • Compact syntax of list: \( X[nb] \)
  • Compact definition of list, \( X[30] \) is same as components=(\( X_1, X_2, X_3, \ldots, X_{30} \))

```python
CADISK = PhysicalQuantity(
    type = 'R',
    components = ('K[144]',),
    comment = 'Rigidity matrix for DIS_* elements'
)
```
Physical quantities – Special

• Special: **anonymized**
  • Used when it’s not necessary to defined a special physical quantity
  • Exists for integer, real and strings

```python
NEUT_R = PhysicalQuantity(
    type = 'R',
    components = ('X[30]',),
    comment = """
)
```
Physical quantities – Special

- Special: JEVEUX adress
  - Used for material parameters access
  - Required special preparation in Fortran: only for core team!

```python
ADRSJEVE = PhysicalQuantity(
    type       = 'I',
    components = ('I1',),
)

ADRSJEVN = PhysicalQuantity(
    type       = 'I',
    components = ('I[5]',),
)```

30 - Code_Aster and Salome-Meca course material  GNU FDL Licence
Physical quantities – Array

• Array of physical quantity:
  • Allow to define elementary quantities to assemble (finite element method)
  • Assemble with dedicated subroutines (assvec.F90 for vectors and asmatr.F90 for matrix)
  • Need to define numbering (with numero.F90)
  • Based on physical quantity real or complex only
  • Support must been ‘ELNO’ in finite element catalog
Physical quantities – Array

- Physical quantities is an elementary vector
  - Example: vector based on DEPL_R

```
VDEP_R = ArrayOfQuantities(elem='V', phys= DEPL_R)
```
Physical quantities – Array

- Physical quantity is an elementary matrix:
  - Example: symmetric matrix MS based on DEPL_C

MDEP_C = ArrayOfQuantities(elem='MS', phys= DEPL_C)

- Example: unsymmetric matrix MR based on DEPL_R

MDNS_R = ArrayOfQuantities(elem='MR', phys= DEPL_R)
Physical quantities – Support

- Restriction of physical quantities on given support:
  - From global physical quantities
  - Restricted list of components
  - On given support: ELEM, ELNO or ELGA

- ELEM
  - Constant by element

- ELNO
  - On nodes in element

- ELGA
  - On Gauss points
Physical quantities – Support

- For ‘ELEM’ support

```python
import cataelem.Commons.physical_quantities as PHY

EDCEL_I = LocatedComponents(phs=PHY.DCEL_I, type='ELEM',
                           components=('NPG_DYN', 'NCMP_DYN',))
```
Physical quantities – Support

- For ‘ELGA’ support: you must define integration scheme (location)

```python
import cataelem.Commons.physical_quantities as PHY

G27NEUTR = LocatedComponents(phys=PHY.NEUT_R,
                            type='ELGA', location='RIGI',
                            components=('X[27]',))
```
Physical quantities – Support

- For ‘ELNO’ support: you can define **differentiated** physical quantities

```python
DDL_MECA = LocatedComponents(phys = PHY.DEPL_R,
                          type = 'ELNO',
                          diff = True,
                          components=(
                          ['EN1',('DX','DY','TEMP',)),
                          ['EN2',('DX','DY',))])
```

**Definition of components on specific set of nodes: EN1 and EN2**

**You must define set of nodes EN1 and EN2: only in finite elements catalogs!**
Physical quantities – Support ‘ELNO’ and nodal field

- Warning: ‘ELNO’ support is not the same as ‘NODE’ one!
- Defined on element: nodes for each element
- calcul.F90 automatically convert nodal field in ELNO one

Field defined on nodes

texxxx.F90: each element is INDEPENDENT but underground is the same value
Physical quantities – Support ‘ELNO’ and nodal field

- The ‘ELNO’ support is not the same as ‘NODE’ one
  - Support ‘ELNO’ is defined for all nodes one each element
  - Support ‘NOEU’ cannot exists in texxxx.F90 because all elements are independent:
    - Subroutine calcul.F90 automatically convert input nodal field in ‘ELNO’ one
    - No nodal field is possible for output of calcul.F90
    - To have nodal field, you should use array and assemble after calcul.F90
Physical quantities – Support ‘ELNO’ and nodal field

• Subroutine `calcul.F90` automatically convert nodal field in ‘ELNO’

Field defined on nodes

Input for `texxxx.F90`: each element is independent but node has *same* value
Physical quantities – Support ‘ELNO’ and nodal field

- To have output nodal field you must assemble ELNO fields

Output for texxxx.F90: each element is independent and node has different value

Field defined on nodes
Physical quantities – Inheritance

- Restriction of physical quantities on given support:

```python
DCEL_I = PhysicalQuantity(
    type='I',
    components=('NPG_DYN', 'NCMP_DYN'),)
```

**EDCEL_I inherits from DCEL_I**

```python
import cataelem.Commons.physical_quantities as PHY
EDCEL_I = LocatedComponents(phys=PHY.DCEL_I, type='ELEM',
                            components=('NPG_DYN', 'NCMP_DYN'))
```

On support **ELEM** (constant by element)

```python
EDCEL_I = LocatedComponents(phys=PHY.DCEL_I, type='ELEM',
                            components=('NPG_DYN', 'NCMP_DYN'))
```
Physical quantities – Inheritance

- Restriction of physical quantities on given support, with selected components:

```
SIEF_R = PhysicalQuantity(type='R',
                          components=(
                            'SIXX','SIYY','SIZZ',..., 'VMIS','TRESCA', ... ),)
```

**ECOEQNO partially inherits from SIEF_R**

```
import cataelem.Commons.physical_quantities as PHY

ECOEQNO = LocatedComponents(phys=PHY.SIEF_R, type='ELNO',
                             components=('VMIS', 'TRESCA'))
```

On support **ELNO**

```
ECOEQNO

ECOEQNO
```

```
ECOEQNO = LocatedComponents(phys=PHY.SIEF_R, type='ELNO',
                             components=('VMIS', 'TRESCA'))
```
Physical quantities

- Array of components (vector and matrix): from local finite element catalog

```python
DDL_MECA = LocatedComponents(phys=PHY.DEPL_R, type='ELNO',
                             components=('DX', 'DY', 'DZ',))

VECTUR = ArrayOfComponents(phys=PHY.VDEP_R,
                            locatedComponents=(DDL_MECA,))

- Array of components (vector and matrix): from located_components.py catalog

```python
import cataelem.Commons.located_components as LC

MVECZZR = ArrayOfComponents(phys=PHY.VSIZ_R,
                             locatedComponents=(LC.DDL_NOZ1,))
```
ATTRIBUTES
Attributes

• To identify some properties on finite elements: topological size, modelization options, etc.

• Attributes can been used in finite elements catalogs:
  • $ASTER_ROOT/catalo/cataelem/Commons/phenomenons_modelisations
  • For options: $ASTER_ROOT/catalo/cataelem/Options/*.py
  • For finite elements: $ASTER_ROOT/catalo/cataelem/Elements/*.py

• Attributes can been ask in Fortran:
  • Subroutines: teattr.F90 and lteatt.F90

• Definition:
  • Automatic from some other properties (dimension of element, name of modelisations,...)
  • Defined by developers
  • In $ASTER_ROOT/catalo/cataelem/Commons/attributes.py
Attributes

• Define automatic attributes:

```python
DIM_TOPO_MAILLE = Attribute(auto=True,
    value=('0', '1', '2', '3'),
    comment="""
"
"
"
"
```

• To construct automatic attributes => program in:
  $ASTER_ROOT/catalo/cataelem/Tools/build_jeveux.py

• Don’t forget to `comment` attributes (no “paper” documents !)
Attributes

• Some automatic attributes:
  • `DIM_TOPO_MAILLE`: topological dimension of cell (apex, lineic, surfacic, volumic)
  • `PHENO`: short string for phenomenon
    *Mechanique/Thermique/Acoustique/Presentation*
  • `MODEL`: short string for modelisation (in `phenomenons_modelisations.py`)
  • `TYPMA`: short string for geometric cell code (PO1, SE2, HE8, ...)
  • `ALIAS8`: concatenated string from `PHENO/MODEL/TYPMA`
  • `DIM_COOR_MODEL`: general dimension of space (2 or 3)
  • `DIM_TOPO_MODEL`: topological dimension of finite element (-1 for `DIS_*` elements)

Example: plate element (DKT)
```
DIM_TOPO_MAILLE = 2  (triangle/quadrangle)
DIM_COOR_MODEL = 3   (global space is 3D)
DIM_TOPO_MODEL = 2   (plate is locally 2D)
```

Example: isoparametric 2D element (plane stress)
```
DIM_TOPO_MAILLE = 2  (triangle/quadrangle)
DIM_COOR_MODEL = 2   (global space is 3D)
DIM_TOPO_MODEL = 2
```
Attributes

• Some automatic attributes:
  • PRINCIPAL: for principal elements (not BORD)
  • BORD: for boundary elements (not PRINCIPAL)

BORD = 0 ⇔ DIM_TOPO_MAILLE = DIM_TOPO_MODELI ⇔ PRINCIPAL = ‘OUI’

BORD = 0  BORD = -1  BORD = -2  BORD = -3
PRINCIPAL=‘OUI’  PRINCIPAL=‘NON’  PRINCIPAL=‘NON’  PRINCIPAL=‘NON’

No rigidity – For boundary conditions
Attributes

• Warning ! Geometric cell is not finite element
  • Same geometric cell, different finite element

![Diagram showing geometric cells with different modelisations and boundary conditions](image)

- **Geometric cell** = QUAD4
- **Modelisation** = 3D
- **Modelisation** = DKT
- **BORD** = 0
- **BORD** = -1
- **PRINCIPAL** = ‘OUI’
- **PRINCIPAL** = ‘NON’

GNU FDL Licence
Attributes

• Some other attributes:
  • `POUTRE/D_PLAN/C_PLAN/TUYAU`: define specific finite elements (beams, pipes ...)
  • `TYPMOD/TYPMOD2/XFEM/MODTHM`: more precise definition of modelisation
  • `NBSIGM`: number of components for stress tensor (4 or 6)
  • `SIGM=‘OUI’`: if element can compute stress (not structural element with generalized forces `EFGE`)
  • ... see `attributes.py`
PARAMETERS
Parameters – Definition

• To compute something on finite element we have to define *parameters*:
  • Input fields
  • Output fields
Parameters – Definition

• The properties of parameter:
  • Physical quantity
  • Status: input or output
  • Specific definition for post-treatment
Parameters – Definition

- Where define parameters:
  - For same parameters in several finite elements: common definition in $ASTER_ROOT/catalo/cataelem/Commons/parameters.py
  - Adapted parameters for specific option: definition in $ASTER_ROOT/catalo/cataelem/Options/*/*.py
  - Adapted parameters for specific finite element: definition in $ASTER_ROOT/catalo/cataelem/Elements/*/*.py
Parameters

- Status of parameter: input (*parameters.py*):

```python
import cataelem.Commons.physical_quantities as PHY

PDEPLR = InputParameter(phys=PHY.DEPL_R,
    container='RESU!DEPL!N',
    comment=""" Previous displacements""")
```

- Based on physical quantity: `phys=PHY.DEPL_R`
- With comment to define the input parameter: `comment`
- Special information for post-treatment: `container`
Parameters

• Status of parameter: output (`parameters.py`):

```python
import cataelem.Commons.physical_quantities as PHY

PCODRET = OutputParameter(
    phys=PHY.CODE_I,
    type='ELEM',
    comment="""Output code for constitutive law"""
)
```

• Based on physical quantity: `phys=PHY.CODE_I`
• With comment to define the input parameter: `comment`
• Must define localization: `type='ELEM'`
Parameters – The container property

• Parameter definition: the container
  • To compute option for post-treatment (CALC_CHAMP), we required input fields
  • Container: where we can find these fields?

• The different containers are constructed in several parts separated by !:
  • First part: general
  • Second part: name of object
  • Third part: other
Parameters – The container property

- List of first parts for container:
  - `container = 'VOLA!'`: object is constructed automatically in `CALC_CHAMP` (complete list in `ccpara.F90`)
  - `container = 'RESU!'`: object can be found in result datastructure
  - `container = 'MAIL!'`: object can be found in mesh
  - `container = 'MODL!'`: object can be found in model
  - `container = 'CARA!'`: object can be found in elementary characteristics (`AFFE_CARA_ELEM`)
  - `container = 'CHMA!'`: object can be found in material characteristics (`AFFE_MATERIAU`)
Parameters – The container property

• Other parts for container:
  • container = ‘VOLA!xxxx’: object xxxx is constructed automatically in 
    CALC_CHAMP (complete list in ccpara.F90)
  • container = ‘RESU!xxxx!y’: object xxxx can been found in result datastructure, at 
    current time step \( y=N \) or previous time step \( y=NM1 \)
  • For container = ‘MAIL!xxxx’, ‘MODL!xxxx’, ‘CARA!xxxx’, ‘CHMA!xxxx’: 
    where xxxx is the name of JEVEUX object (see D4 documentation)

```plaintext
container = ‘RESU!VARI_ELGA!NM1’
⇒ VARI_ELGA in result datastructures for previous step time (NM1)  
container = ‘RESU!VARI_ELGA!N’
⇒ VARI_ELGA in result datastructures for previous step time (N)  
container = ‘MODL!.TOPOSE.CNS’
⇒ MODEL .TOPOSE.CNS (object for XFEM)
```
OPTIONS
Options

- A option define what to compute:
  - Input fields
  - Output fields
  - Name of option
  - When can we compute this option

\[ \int_{\Omega} \sigma(u) : \varepsilon(\tilde{u}) \, d\Omega \]

\[ \int_{\Omega} f(u) : \tilde{u} \, d\Omega \]

options.py
Options – Definition

• Properties of option:
  • Name of option
  • List of input parameters: para_in
  • List of output parameters: para_out
  • Condition to compute option: condition
  • Comments with comment

• Catalogs for options:
  • $ASTER_ROOT/catalo/cataelem/Options/*.py
Options – Catalog

- **Example** *(full_meca.py)*:

```python
import cataelem.Commons.physical_quantities as PHY
import cataelem.Commons.parameters as SP

PCOMPOR = InputParameter(phys=PHY.COMPOR)

FULL_MECA = Option(
    para_in=(...),
    para_out=(...),
    condition=(...),
    comment=""""""",
)
```

- Local definition of parameters: PCOMPOR, ...
- Name of option: FULL_MECA
- List of input parameters: para_in
- List of output parameters: para_out
- Condition to compute option: condition
- Comments with comment
Options – Catalog

• Define input/output fields (*full_meca.py*):

```python
import cataelem.Commons.physical_quantities as PHY
import cataelem.Commons.parameters as SP

PCOMPOR = InputParameter(phys=PHY.COMPOR)

FULL_MECA = Option(
    para_in=(PCOMPOR, SP.PDDEPLA,),
    para_out=(SP.PVECTUR),
    condition=(...),
    comment="""""",)
```

• From generic definition of parameters: in
  
  `$/ASTER_ROOT/catalo/cataelem/Commons/parameters.py`

• Local definition in *full_meca.py*
Options – CondCalcul

• **Conditions (CondCalcul properties):**
  • To define which finite element can compute the option

• **General algorithm:**
  • Option defined in finite element catalog (number of texxxxx.F90 is given) => the option can been computed *(OK)*
  • Option is not defined in finite element catalog (no number for texxxxx.F90) => the option cannot been computed *(Fatal error for user)*

• **Using CondCalcul:**
  • Define explicitly set of finite elements for which option has no sense *(with no error for user)*
Options – CondCalcul

• Example (full_meca.py):

```python
import cataelem.Commons.attributes as AT

FULL_MECA = Option(
    para_in=(...),
    para_out=(...),
    condition=(
        CondCalcul('+', ((AT.PHENO, 'ME'), (AT.BORD, '0'))),
        CondCalcul('-', ((AT.PHENO, 'ME'), (AT.ABSO, 'OUI'))),)
```

• Adding a class of finite element using attributes: `CondCalcul('+', ...)`
• Suppress a class of finite element using attributes: `CondCalcul('-', ...)`
• Use parameters to define `CondCalcul`
Options – CondCalcul

- **Example** *(full_meca.py)*:

```python
import cataelem.Commons.attributes as AT

FULL_MECA = Option(
    para_in=(...),
    para_out=(...),
    condition=(
        CondCalcul('+', ((AT.PHENO,'ME'),(AT.BORD,'0'),)),
        CondCalcul('-', ((AT.PHENO,'ME'),(AT.ABSO,'OUI'),)),),
    comment="""
    """
)
```

- **Option** `full_meca` compute tangent matrix for non-linear mechanics
- **CondCalcul('+', ((AT.PHENO,'ME'),(AT.BORD,'0'),))**: option has sense in mechanics *(AT.PHENO, 'ME')* but only for principal element *(AT.BORD, '0')*

**And:**
- **CondCalcul('-', ((AT.PHENO,'ME'),(AT.ABSO,'OUI'),))**: option has no-sense for *ABS0* elements
MODELISATION
Modelisation – Definition

• **Definition:**
  - For given physic (**PHENOMENON**): mechanic, thermic, acoustic
  - Is a set of finite elements: same dof on different geometric cells

• **Catalog for all modelisations:**
  $ASTER_ROOT/catalo/cataelem/ Commons/phenomenons_modelisations.py$

• **Using modelisations:**
  - Most of them are used in **AFFE_MODELE** command
  - Some are used in other commands (**DEFI_CONTACT** for instance)
Modelisation – Definition

• **Definition** (*phenomenons_modelisations.py*):

```python
MECANIQUE = Phenomenon(code='ME')
phen = MECANIQUE
phen.add('3D',
    Modelisation(dim=(3,3),
        code='3D_',
        attrs=(...),
        elements=(...),)
)
```

• **Select physics**: PHENOMENON

• **Add modelisation**:
  • Name (for `AFFE_MODELE`): here ‘3D’
  • Code for modelisation: here ‘3D_’
  • Dimensions: `dim(a,b)`
  • Attributes: `attrs`
  • List of all finite elements: `elements` (for each geometric cell => finite element)
Modelisation – Definition

- Define **dimensions**: \( \text{dim}(a, b) \)

```python
phen.add('3D',
    Modelisation(dim=(3, 3), ...)
)
```

- **Dimensions**: \( \text{dim}(a, b) \)
  - \( \text{dim}(., b) \) for global dimension (2 or 3)
  - \( \text{dim}(a, .) \) for topological dimension (from -1 to 3)

- **Modelisation**
  - \( a=3 \) \( b=3 \)  
    Modelisation = ‘3D’  
    3D isoparametric elements
  - \( a=2 \) \( b=3 \)  
    Modelisation = ‘DKT’  
    Plate elements
  - \( a=2 \) \( b=2 \)  
    Modelisation = ‘D_PLAN’  
    Plane strain elements

- \( b=0 \) for POI1 elements
- \( b=-1 \) for mixed POI1 and SEG2 (special for DIS_*)
Modelisation – Definition

• Define **dimensions**: \( \text{dim}(a,b) \)

```python
phen.add('3D',
    Modelisation(dim=(3,3), ... ) )
```

• When define \( \text{dim}(a,b) \) => **automatic attribute**:
  • \( a \) is `DIM_COOR_MODEL`i: general dimension of space (2 or 3)
  • \( b \) is `DIM_TOPO_MODEL`i: topological dimension of finite element (-1 for `DIS_*` elements)

**Example: isoparametric 3D element**

```plaintext
DIM_COOR_MODEL = 3  (global space is 3D)
DIM_TOPO_MODEL = 3
```

**Example: plate element (DKT)**

```plaintext
DIM_COOR_MODEL = 3  (global space is 3D)
DIM_TOPO_MODEL = 2  (plate is locally 2D)
```

**Example: isoparametric 2D element (plane strain)**

```plaintext
DIM_COOR_MODEL = 2  (global space is 3D)
DIM_TOPO_MODEL = 2
```
Modelisation – Definition

• Define **attributes**: `attrs`

```python
import cataelem.Commons.attributes as AT

phen.add('3D',
    Modelisation(
        attrs=(
            (AT.NBSIGM, '6'),
            (AT.TYPMOD, 'COMP3D'),
        ),
    ),
)
```

• Using attributes defined from
  `$ASTER_ROOT/catalo/cataelem/Commons/attributes.py`

• Some attributes are automatic (not defined in `attrs` !)
• Define finite **elements**: elements

```python
import cataelem.Comments.mesh_types as MT
from cataelem.Elements.elements import EL

phen.add(elements=(
    (MT.HEXA8, EL.MECA_HEXA8),
    (MT.PENTA6, EL.MECA_PENTA6),
...
    (MT.TRIA6, EL.MECA_FACE6),
    (MT.SEG3, EL.MECA_ARETE3),
))
```

• A finite element is a pair of
  • One geometric cell from `mesh_types.py`
  • The name of finite element

• You must define all finite elements: volumic and skin ones for boundary conditions

• One geometric cell = one finite element
Finite element – Definition

- The finite elements are defined (by modelisation) in
  \$\text{ASTER\_ROOT/catalo/cataelem/Elements/\ast.py}
- From list of finite elements, get name of finite element

```python
phenomenon_modelisation.py
phen.add(elements=(
    (MT.HEXA20, EL.MECA_HEXAO20),)
```
Finite element – Definition

• **Definition** ($ASTER_ROOT/catalo/cataelem/Elements/*.py):

```python
located components
class MECA_HEX20(Element)
```

• In each finite element file, we have two parts to define:
  • Located components
  • Class for finite element
Finite element – Definition

• Define **located components**:

```python
import cataelem.Commons.physical_quantities as PHY

DDL_MECA = LocatedComponents(phys=PHY.DEPL_R, type='ELNO',
                               components=('DX','DY','DZ',))

class MECA.HEXA20(Element)
```

• The located components **DDL_MECA** is local from physical quantity **DEPL_R**, on support **ELNO** and restricted on three components (**DX**, **DY**, **DZ**).
Finite element – Definition

- Define **class** of finite element:

```python
located components

class MECA_HEX20 (Element):
    meshType = MT.HEXA20
    nodes = (...)
    elrefe = (...)
    calculs = (...)
```

- For each finite element
  - Geometric cell: meshType
  - Sets of nodes: nodes
  - Integration scheme: elrefe
  - Option to compute: calculs
Finite element – Definition

- Define **class** of finite element: *inheriting process*

```python
class MECA_HEXA20(Element):
    meshType = MT.HEXA20
    nodes   = (...)
    elrefe  = (...)
    calculs = (...)

class MECA_HEXA8(MECA_HEXA20):
    meshType = MT.HEXA8
    nodes   = (...)
    elrefe  = (...)
    calculs = (...)
```

- **MECA_HEXA8** inherits from **MECA_HEXA20**
- Inheriting process from one finite element to another: it’s not necessary to redefine all properties except `meshType` and `nodes`
Finite element – Definition

• **Definition** ($ASTER_ROOT$/catalo/cataelem/Elements/*.py):

```python
DDL_MECA = LocatedComponents(phys=PHY.DEPL_R, type='ELNO',
                             components=('DX','DY',))

class MECA_QUAD8(Element):
    meshType = MT.QUAD8
    nodes = SetOfNodes('EN1', (1,2,3,4,5,6,7,8),),
```

• Same components on all nodes: define **EN1**
Finite element – Definition

• Define **set of nodes** in class: nodes

```python
DDL_MECA = LocatedComponents(phys=PHY.DEPL_R, type='ELNO', diff=True,
                           components=((‘EN1’, (‘DX’, ‘DY’)),
                           (‘EN2’, (‘DRZ’,))))

class MECA_QUAD8(Element):
    meshType = MT.QUAD8
    nodes = SetOfNodes(‘EN1’, (1, 2, 3, 4),
                       ‘EN2’, (5, 6, 7, 8),)
```

• Define several set of nodes (**EN1**, **EN2**, ...) to describe different components on nodes
Finite element – Definition

• Define *integration schemes families* in class: `elrefe`

```python
class MECA_HEXAX20(Element):
    meshType = MT.HEXAX20
    elrefe =
    ElrefeLoc(MT.H20,
        gauss = ('RIGI=FPG27', 'FPG1=FPG1', 'MASS=FPG27', 'NOEU=NOEU', ),
        mater = ('RIGI', 'MASS', 'NOEU', 'FPG1', ), ),
    ElrefeLoc(MT.QU8,
        gauss = ('RIGI=FPG9', 'MASS=FPG9', 'NOEU=NOEU', ), ),
)
```

• On mesh type (volumic and surfacic), we define two kind of integration schemes:
  
  • `gauss`: use for all elementary computation defined on ELGA support
  • `mater`: especially for material parameters (using `ADRSJEVE` physical quantity)
Finite element – Definition

• Define integration schemes families \textit{gauss} in class

```python
class MECA_HEXAA0(Element):
    meshType = MT.HEXA20
    elrefe = (ElrefeLoc(MT.H20,
        gauss = ('RIGI=FPG27','FPG1=FPG1','MASS=FPG27','NOEU=NOEU',),),
        mater = ('RIGI','MASS','NOEU','FPG1',),),
    ElrefeLoc(MT.QU8,
        gauss = ('RIGI=FPG9','MASS=FPG9','NOEU=NOEU',),),
    )
```

• Family definition: for \textit{RIGI gauss family}, we use \textit{FPG27} integration scheme as defined on \textit{HEXA20} mesh type:
  • List of integration schemes on every mesh type is defined in
    \texttt{$ASTER\_ROOT/catalo/cataelem/Commons/mesh\_types.py$}
  • List of integration points, coordinates and weight in Fortran subroutine \texttt{elraga.F90}
  • \textit{RIGI} is the local name of integration scheme for \text{MECA\_HEXA20} finite element
  • \textit{FPG27} is the general name of integration scheme for \textit{HEXA20} mesh types
Finite element – Definition

• Define integration schemes families \textit{mater} in class

```python
class MECA_HEXAX20(Element):
    meshType = MT.HEXA20
    elrefe = (ElrefeLoc(MT.H20,
        gauss = ('RIGI=FPG27', 'FPG1=FPG1', 'MASS=FPG27', 'NOEU=NOEU',),
        mater = ('RIGI', 'MASS', 'NOEU', 'FPG1'),),
    ElrefeLoc(MT.QU8,
        gauss = ('RIGI=FPG9', 'MASS=FPG9', 'NOEU=NOEU'),),
)
```

• Family definition: for mater family, list of gauss family point we can use to access material parameters)
Finite element – Definition

• Define **options** in class: `calculs`

```python
class MECA_HEX20(Element):
    meshType = MT.HEXA20
    calculs = (OP.RIGI_MECA(te=11,
        para_in = ((PCAMASS, CCAMASS), (PGEOMER, NGEOMER),
                   (PMATERC, CMATERC), (PVARCPR, ZVARCPG)),
        para_out = ((PMATUUR, MMATUUR), ),
    ),
```

• The finite element can compute option `RIGI_MECA`:
  • Input fields: type and name
  • Output fields: type and name
  • Index of texxxx.F90 subroutine
End of presentation

Is something missing or unclear in this document?
Or feeling happy to have read such a clear tutorial?

Please, we welcome any feedbacks about Code_Aster training materials.
Do not hesitate to share with us your comments on the Code_Aster forum dedicated thread.