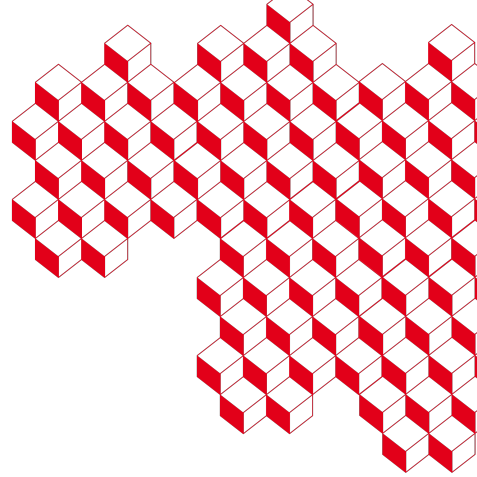

MEDCoupling Introduction and overview

08-03-2024

Aymeric SONOLET, Guillaume BROOKING

Outline

Agenda

Training overview & objectives

Introduction to functionalities

BASICS

Library and code Structure

Note for C++ developers

Agenda

- Training overview & objectives
- Introduction to the MED world
- Functionalities
- Basic concepts
- DataArrays
- Meshes
- Fields
- Library and code structure
- Some practical details for the exercises
- API changes in V8+
- Notes for C++ developers

Outline

Agenda

Training overview & objectives

Introduction to functionalities

BASICS

Library and code Structure

Note for C++ developers

Training overview

Agenda

1. General introduction to the SALOME MED module
   - Exercises
2. MEDLoader: reading and writing MED files
   - Exercises
3. Interpolation: what MED do for you
   - Exercises

Objectives

- Understand what MED offers
- Discover the main concepts used in MED (meshes, fields, indexing techniques)
- Be able to use via Python on simple (but representative!) use cases

Outline

Agenda

Training overview & objectives

Introduction to functionalities

BASICS

Library and code Structure

Note for C++ developers

MEDCoupling Introduction, Mesh, fields, . . . what for?

- Simulation studies require manipulations of
  - Mesh = spatial discretization of a geometric domain
  - Fields = physical data / boundary conditions on the above discretization
  - They often form the main input of a numerical solver
    - And can be used to store the simulation results too
- Building, accessing and interacting with these elements demands suited tools
- Many possible routes
  - Ad-hoc libraries : still often the case in dedicated codes
    - Problem: interaction with the rest of the world (other codes, post-processing tools, etc . . .)
  - SALOME's philosophy: common building block
    - A shared tool to work on meshes and fields
    - But MEDCoupling is much more than just a way to standardize interaction!!
- A powerful library to perform all this: MEDCoupling
- Difference with SMESH??
  - In SMESH, the geometry is the starting point. No field.
  - MED acts more as an interface with the numerical code

MEDCoupling Functionalities, A short tour

- Mesh manipulation
  - Build from scratch
  - Refinement & sub-splitting
  - Intersection (some spatial configurations only!)
- Fields
  - Projection from one mesh to another
  - Taking into account physical nature of the field
- Parallel codes – for field projection mainly
  - Work splitting between groups of processors
    - MEDPartitioner
  - Serialization, data exchange and projection between groups of procs
    - ParaMEDMEM
- Most of the functionalities accessible with Python

MEDCoupling Functionalities, And many more

- Convex hull computation
- Duplicate nodes identification
- Degenerated cells reduction, typically a flat triangle
- Point localization
- Algebraic volume, area, length computation
- Mesh concatenation
- Eigen values computation
- Gauss points management
- And many more
- What you need is not there yet? Ask for it!

What do we understand under MED in SALOME?

- Core structures (arrays, meshes and algorithms)
- Projection (interpolation and field projection)
- Parallelism (DEC - Data Exchange Channel)
- File I/O (MEDFiles)
- MEDReader, Paraview plugin to visualize MEDFiles
- MED GUI (aka Fields)

History, an ongoing effort

- 1996: study about the standardization of data exchange between EdF R&D simulation codes
- 1997: first version of the data model
- 1998-1999: first version of the MED-file library at EdF and joint work with CEA (ELAN)
- 2001: including MED into the SALOME platform for data exchange (meshes and fields)
- 2003: first version of MED in memory (MEDMEM): exchanging objects between processes directly in memory
- 2010: first version of MEDCoupling
- 2013: removing MEDMEM from SALOME 7
- 2014: MEDCoupling/MEDLoader: engine behind the new MED reader in ParaView / PARAVIS

Outline

Agenda

Training overview & objectives

Introduction to functionalities

BASICS

Library and code Structure

Note for C++ developers

DataArrays (1/2)

- To do all this, surely, we need
  - Arrays of points (mesh nodes coordinates, centers of mass, . . .)
  - Arrays of “data” (fields, etc . . .)
- First building block: DataArray
  - “Similar” to a list with several columns
    - Usually much more rows than columns!
  - Example: array of 2D points = DataArrayDouble
    - coo = mc.DataArrayDouble([(1.0,2.0), (2.0,3.5), (1.5,3.4)])
    - Array of 3 tuples and 2 components
- Main sub-types: DataArray’s of int or double
- All indices start at zero
- In Python, DataArray-s are manipulated very much like NumPy arrays
- Important point for I/O: DataArray-s have a name, and so have their components

DataArrays (2/2)

- All standard operators directly accessible using operators:
  - `da = mc.DataArrayInt([1,2,3])`
  - `da *= 2`
  - `da = (1-da)`
  - Similarity with NumPy: by default, operations are done at the tuple level (no matrix-like operations)
- Advanced operations: intersection (for DAInt), min/max extraction, ...
- Link with NumPy (need to have a MED compiled with NumPy support!)
  - `DataArrayInt` and `DataArrayDouble` can be converted to NumPy array
    - Efficient in both ways (no copy, just ownership transfer)
  - Advantages
    - mmap, serialize/deserialize with multiprocessing
    - Usage with arrays of bool (where() clauses)
    - Syntactic sugar for dimension management (translate, newaxis, ...)
    - Link with SciPy for linear algebra
    - Link with pyCUDA
  - Restrictions
    - Some services missing: `getMinMaxPerComponent()`, `findCommonTuples()`, ... or to be invoked with another approach
    - No names associated to arrays/components

Renumbering, Easy index manipulation

- Powerful indexing methods – some examples
  - `da[[1,3,4]]` : gets the 2nd, 4th and 5th elements
  - `da[:, 0]` : extract the first component of all tuples
- Another classical indexing technique: array renumbering. You might want to re-organize an array according to a surjection (a mapping works too, obviously)

```
>>> da = mc.DataArrayInt([2,3,4,5,6])
>>> surj = mc.DataArrayInt([0,2,4,1,4])
>>> surj
[2,4,6,3,6]
>>> result = da[surj]
```

- The point: a lot of MEDCoupling functions work with (or return) arrays in the format of surj above.
- See exercises for more on this and section Array Renumbering in the doc.

MESHES, Some geometry

- In the MED world, a mesh is
  - The spatial discretization of a continuous geometrical domain
  - Associated to only one array of underlying point coordinates (a DataArrayDouble, called the nodes of the mesh)
  - With only one single mesh dimension
    - You can NOT mix a mesh representing 3D volumes (say cubes) with 2D areas (triangles)
  - A mesh also has a name (important for I/O) and a time-step ID
- Do not confuse
  - Mesh dimension: dimension of the cells
  - Spatial dimension: number of components in the array of coordinates
  - Example: an helix-shaped curve (= a wire shaped like a corkscrew) has
    - Mesh dim = 1 (this is a simple wire made of segment cells)
    - Spatial dimension = 3
  - A curved surface has mesh dimension = 2, space dimension = 3
- A mesh is made of cells: segments in 1D, surfaces in 2D, volumes in 3D.
- Main types of mesh in MEDCoupling: structured, unstructured and extruded

Cell representation (unstructured)

- A cell is described by the list of point identifiers (not point coordinates) delimiting it
  - [0,1,2,3,4,5,6,7]
  - “cell” is somewhat of a misnomer – can be a segment in 1D
- Need conventions!
  - E.g., more than one way to index a cube’s vertices:
- No explicit notion of “edges” for a 2D cell, no notion of “faces” for a 3D volume
  - Only points
  - But you can re-compute this if needed: buildDescendingConnectivity()
- You will see those code names
  - HEXA8, HEXA20: e.g. hexahedron with 8 points = a “cube” really. The one with 20 points represent a “quadratic” element (“cube with curved faces”).
  - TETRA4, TETRA10
  - Etc . . .
- The MED file documentation has them all
- Note: in 2D, the reverse trigonometric convention is used

Connectivity representation, focus on a typical way to

- Internal representation of the cell connectivity
  - By no means mandatory to memorize(!)
  - Just gives a good example of the indirect index format

Fields, just an array of values?

- A field represents some physical quantity associated with the spatial domain
  - No continuous description of the physical quantity over the domain
  - A finite set of values, associated to some constituents of the mesh
  - At a low level, a DataArray associated to a given mesh
    - Can have more than one component!
- A field can be supported by nodes (vertices) ON_NODES, cells (elements) of the mesh ON_CELLS, or more complex items
- A field has a temporal discretization
  - NO_TIME
  - ONE_TIME
  - CONST_ON_TIME_INTERVAL
- For some operations (interpolation), one need to define the physical nature of the field (extensive field or intensive field)
- More on all this with interpolation . . .
- Examples
  - Magnetic field: tensor field of double values: 3 components (Bx, By, Bz)
  - Temperature field: scalar field of double values: 1 component

Illustration

Outline

Agenda

Training overview & objectives

Introduction to functionalities

BASICS

Library and code Structure

Note for C++ developers

Dependency Structure

- Several sub-parts each dedicated to a specific task
  - MEDCoupling: memory model and general processing
  - MEDLoader: persistence
  - ParaMEDMEM: parallelism
- A big effort to have little dependencies
- Parts are:
  - Swigged
  - Swigged and wrapped with CORBA
  - System dependencies

A few words about the code

- Code
  - 138k+ lines of C++ code (35k+ for tests purposes)
  - 41k+ lines of Python (35k+ for test purposes)
  - More than 1600 unit tests
  - Valgrind 0 on all unit tests
  - More than 80% test coverage
  - Everything in C++ with a thin layer of Python to wrap it
- Source control via Git
- Configuration and build with CMake
  - Help with portability (compiling on Win64 for example)
- A very dynamic library
  - Regular improvements and bug fixes
  - SGLS team at CEA, and Anthony GEAY (initial author) at EdF supporting the dev

Outline

Agenda

Training overview & objectives

Introduction to functionalities

BASICS

Library and code Structure

Note for C++ developers

Base classes

RefCountObject abstract class

- Similarities with VTK code structure
  - Ease the interaction with VTK (ParaView plugins)
  - Historically, there's been a reflection about using VTK directly
- All (significant) MEDCoupling classes inherit from
  - RefCountObject
    - A pointer and a counter
    - Memory management philosophy: someone owns the object after its creation
    - incrRef() to take ownership of the object / decrRef() to release it
    - Template class MCAuto is here to help
    - Smart pointer behavior (no Boost dependency)
    - No need to decrRef() if used
    - valgrind is your friend
  - Careful, some MEDCoupling functions
    - give you pointer ownership: e.g. mergeNodes()
    - Some don't: getCoords()