Data management
Kratos Workshop 2019

Vicente Mataix Ferrándiz [1]
vmataix@cimne.upc.edu

[1] CIMNE. International Center for Numerical Methods in Engineering, Technical University of Catalonia (UPC). Barcelona. Spain

March 27, 2019

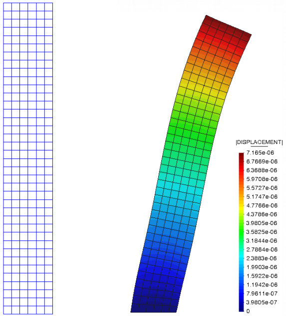
Overview

Section 1

Introduction

Introduction

In the following presentation we will in first place introduce the *Kratos* data structures related with **Data management**. In the second part of this presentation we will present different data manipulations with a simple 2D structural example.



(a) Mesh (b) Deformation

Figure 1: Data management example

Kratos structure classes

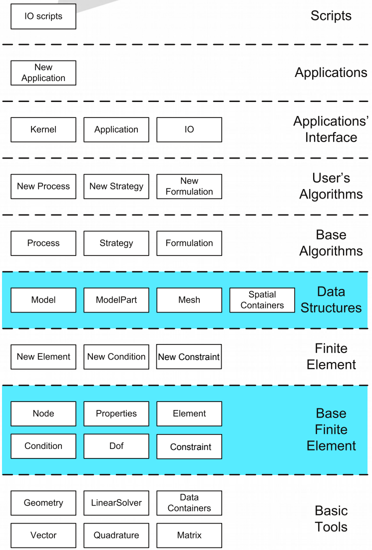


Figure 2: Kratos structure classes

Groups

- *Scripts:* Simple scripted programs created in order to reduce the workload and simplify run problems
- *Applications:* This is the base of the modularity of **Kratos**. Each application can be defined o solve an specific problem and couple them later
- *App interface (core):* Communicate each components and define the framework behaviour
- *Algorithms:* Operations that are used to solve the problem (strategies, time schemes, algorithms, etc...)
- **Data structure: Contains the information of the problem (geometries, elements, etc...)**
- **Finite element: The base components necessaries to define a FE problem (DoF, elements, nodes, etc...)**
- *Basic tools:* Algebraic and mathematic components

The groups and classes in **cyan** will be detailed later for being more related with examples to be run

Data structures classes

`Model`

`Model` stores the whole model to be analyzed. All `Nodes`, `Properties`, `Elements`, `Conditions` and solution data

`ModelPart`

`ModelPart` holds all data related to an arbitrary part of model. It stores all existing components and data like `Nodes`, `Properties`, `Elements`, `Conditions` and solution data related to a part of model

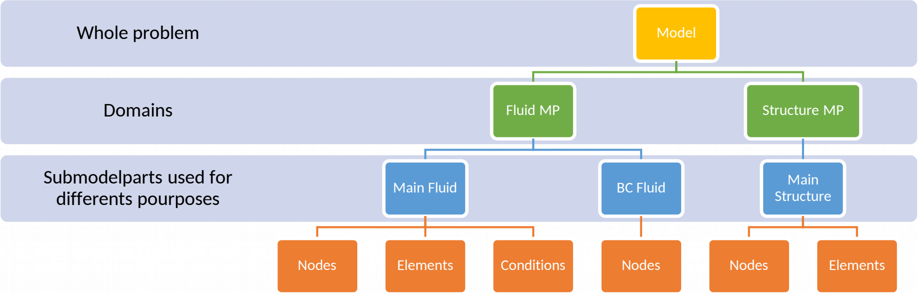


Figure 3: Example

Finite element classes

`Node`

`Node` It is a point with additional facilities. Stores the nodal data, historical nodal data, and list of *DoF*

`Condition`

`Condition` encapsulates data and operations necessary for calculating the local contributions of `Condition` to the global system of equations. *Neumann* conditions are example

`Elements`

`Element` encapsulates the elemental formulation in one object and provides an interface for calculating the local matrices and vectors necessary for assembling the global system of equations. It holds its geometry that meanwhile is its array of `Nodes`. Also stores the elemental data

`Properties`

`Properties` encapsulates data shared by different `Elements` or `Conditions`. It can store any type of data

`DoF`

`DoF` represents a degree of freedom (*DoF*). This class enables the system to work with different set of *DoFs* and also represents the *Dirichlet* condition assigned to each *DoF*

Submodel part concept

The concept of submodelpart is important to understand, because it is the structure that is usually used from an interface point of view in order to assign and set **BC**, or properfies affecting a particular region of the problem, etc... without affecting the rest of the problem.

This can be seen in the following example Figure 4:

Processes and submodelparts relationship

```
"python_module" : "apply_inlet_process",
"kratos_module" : "KratosMultiphysics.FluidDynamicsApplication",
"help"          : [],
"process_name"  : "ApplyInletProcess",
"Parameters"    : {
    "model_part_name" : "AutomaticInlet2D_Inlet",
    "variable_name"   : "VELOCITY",
    "modulus"         : 1.0,
    "direction"       : "automatic_inwards_normal",
    "interval"        : [0,"End"]
}
```

Figure 4: Example of BC in *json* format

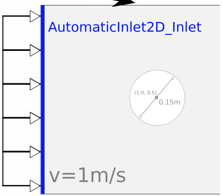


Figure 5: Subm. BC

Variables List Data Value Container

`VariablesListDataValueContainer`: A shared variable list gives the position of each variable in the containers sharing it. The mechanism is very simple. There is an array whichstores the local offset for each variable in the container and assigns the value −1 for the rest of the variables. **AKA historical values**

Data containers

`Data containers` A data value container is a heterogeneous container with a variable base interface designed to hold the value for any type of variable. **AKA non-historical values**

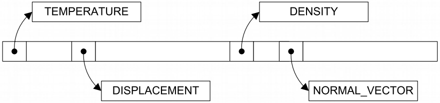


Figure 6: Variables List Data Value Container

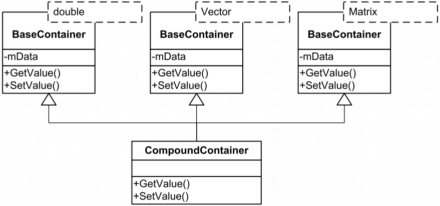


Figure 7: Data value container

Retrieve the example problem

The files for this tutorial can be found in:

https://github.com/KratosMultiphysics/Documentation/tree/master/Workshops_files/Kratos_Workshop_2019/Sources

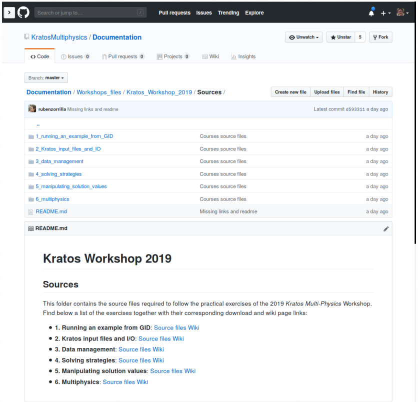


Figure 8: Download

Tutorials can be found in:
https://github.com/KratosMultiphysics/Kratos/wiki/Data-management
https://github.com/KratosMultiphysics/Kratos/wiki/Python-Tutorials

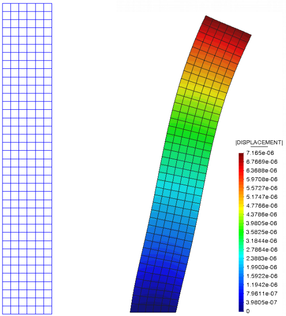
Section 2

Data management tutorial

Introduction

Before starting with the manipulation of the example problem (Figure 9) we will show some *Python* operations from scratch in order to introduce the most basic conceps.
The operations the *Kratos* classes that are accesible via *Python* can be found in in:

https://github.com/KratosMultiphysics/Kratos/wiki/Kratos-classes-accesible-via-python



(a) Mesh (b) Deformation

Figure 9: Data management example

Generating New Modelparts (I)

We need to create the `Model`, which will be the resposible to manage the different `ModelParts` that we will create.

```
import KratosMultiphysics
this_model = KratosMultiphysics.Model()
```

Now we can create a `ModelPart`. The `ModelPart` is the object containing `Element`, `Conditions`, `Nodes` and `Properties`. For now we create the *Main* model part, which will store the successive submodelparts.

```
main_model_part = this_model.CreateModelPart("Main")
```

We can create a new model part with a certain *buffer size* using the following (we need to delete the model part to avoid errors):

```
this_model.DeleteModelPart("Main")
main_model_part = this_model.CreateModelPart("Main", 2)
```

We can now execute different operations with the `Model`:

```
print(this_model.HasModelPart("Main")) # It will return True
print(this_model.GetModelPartNames()) # It will return ['Main']
main_model_part_again = this_model.GetModelPart("Main") # Getting again
```

Let's output what is there:

```
print(main_model_part)
```

```
-Main- model part
    Buffer Size : 2
    Number of tables : 0
    Number of sub model parts : 0
    Current solution step index : 0

    Mesh 0 :
        Number of Nodes       : 0
        Number of Properties  : 0
        Number of Elements    : 0
        Number of Conditions  : 0
        Number of Constraints : 0
```

Some other operations we can do are:

```
print(this_model.NumberOfNodes()) # It will return 0
print(this_model.NumberOfElements()) # It will return 0
print(this_model.NumberOfConditions()) # It will return 0
print(this_model.NumberOfMasterSlaveConstraints()) # It will return 0
print(this_model.NumberOfProperties()) # It will return 0
print(this_model.NumberOfMeshes()) # It will return 1
print(this_model.GetBufferSize()) # It will return 2
this_model.SetBufferSize(3) # Set the buffer size to 3 instead of 2
```

Generating New Modelparts (II)

Now we can create a `ModelPart`. The `ModelPart` is the object containing `Element`, `Conditions`, `Nodes` and `Properties`. For now we create the *Main* model part, which will store the successive submodelparts.

```
bc_model_part = main_model_part.CreateSubModelPart("BC")
```

Let's output what is there:

```
print(main_model_part)
```

```
-Main- model part
    Buffer Size : 3
    Number of tables : 0
    Number of sub model parts : 1
    Current solution step index : 0

    Mesh 0 :
        Number of Nodes       : 0
        Number of Properties  : 0
        Number of Elements    : 0
        Number of Conditions  : 0
        Number of Constraints : 0

    -BC- model part
        Number of tables : 0
        Number of sub model parts : 0

        Mesh 0 :
            Number of Nodes       : 0
            Number of Properties  : 0
            Number of Elements    : 0
            Number of Conditions  : 0
            Number of Constraints : 0
```

Now we can do several operations with this:

```
print(main_model_part.HasSubModelPart("BC")) #returns True
print(main_model_part.NumberOfSubModelParts()) #returns 1
print(main_model_part.GetSubModelPart("BC").Name) #returns the name --> BC
```

Data Ownership

The parent-son relation is such that anything that belongs to a given **SubModelPart also belongs to the parent ModelPart**. This implies that the ultimate owner of any `Node`, `Element`, etc, will be the **root** `ModelPart`. The consistency of the tree is ensured by the `ModelPart` **API**, which provides the tools needed for creating or removing anything any of the contained objects.

You can access in any moment to the *root* owner part and the model with the following:

```
main_model_part = bc_model_part.GetRootModelPart()
this_model = main_model_part.GetModel()
```

Entities creation from scratch (I)

We can create a node by doing. If we Try to create a new node with the same Id and different coordinates we would get an error.

```
main_model_part.CreateNewNode(1, 1.0,0.0,0.0)
#main_model_part.CreateNewNode(1, 0.0,0.0,0.0)  # Here an error is thrown
```

However if we try to create a node with the same coordinates twice nothing is actually done (and no error is thrown)

```
main_model_part.CreateNewNode(1, 1.00,0.00,0.00)
print(main_model_part.NumberOfNodes()) # This still returns 1!!
```

We can now access the node as needed, for example:

```
print(main_model_part.GetNode(1).Id) # Gives 1
print(main_model_part.GetNode(1,0).X) # Gives 1.0
```

`Nodes` can be created in every order:

```
main_model_part.CreateNewNode(2000, 2.00,0.00,0.00)
main_model_part.CreateNewNode(2, 2.00,0.00,0.00)
```

We could then loop over all the nodes:

```
for node in main_model_part.Nodes:
    print(node.Id, node.X, node.Y, node.Z)
```

Or eventually remove nodes one by one by doing:

```
main_model_part.RemoveNode(2000)
```

Let's now see what happens if we add a node to a submodelpart. Here the node will be both in root `ModelPart` and "BC", but for example not in derived submodelparts from "BC" or root `ModelPart`.

```
bc_model_part = main_model_part.GetSubModelPart("BC")
bc_model_part.CreateNewNode(3, 3.00,0.00,0.00)
```

Multiple nodes can be removed at once (and from all levels) by flagging them:

```
for node in main_model_part.Nodes:
    if node.Id < 3:
        node.Set(KratosMultiphysics.TO_ERASE,True)
main_model_part.RemoveNodesFromAllLevels(KratosMultiphysics.TO_ERASE)
```

One could call simply the function RemoveNodes and remove them from the current level down.

Entities creation from scratch (II)

`Elements` and `Conditions` can be created from the *Python* interface by providing their connectivity as well as the `Properties` to be employed in the creation. The string to be provided is the name by which the element is registered in *Kratos*. An error is thrown if i try to create an element with the same Id

```
main_model_part.AddProperties(KratosMultiphysics.Properties(1))
main_model_part.CreateNewElement("Element2D3N", 1, [1,2,3], main_model_part.GetPropertie
s()[1])
#main_model_part.CreateNewElement("Element2D3N", 1, [1,2,3], main_model_part.GetProperti
es()[1])
```

An identical interface is provided for Conditions, as well as functions equivalent to the nodes for removing from one level or from all the levels.

```
mp = this_model.CreateModelPart("constraint_example")

mp.AddNodalSolutionStepVariable(KratosMultiphysics.DISPLACEMENT)
mp.AddNodalSolutionStepVariable(KratosMultiphysics.REACTION)

mp.CreateNewNode(1, 0.00000, 0.00000, 0.00000)
mp.CreateNewNode(2, 0.00000, 1.00000, 0.00000)

KratosMultiphysics.VariableUtils().AddDof(KratosMultiphysics.DISPLACEMENT_X, KratosMulti
physics.REACTION_X, mp)

mp.CreateNewMasterSlaveConstraint("LinearMasterSlaveConstraint", 1, mp.Nodes[1], KratosM
ultiphysics.DISPLACEMENT_X, mp.Nodes[2], KratosMultiphysics.DISPLACEMENT_X, 1.0, 0)
```

ModelPart entities

First of all we need to create a *Python* file with following code to import the *Kratos*, create a `ModelPart` and read it from input as described in the here:

```
import KratosMultiphysics
import KratosMultiphysics.StructuralMechanicsApplication

this_model = KratosMultiphysics.Model()
structural_model_part = this_model.CreateModelPart("StructuralPart", 3)

structural_model_part.AddNodalSolutionStepVariable(KratosMultiphysics.DISPLACEMENT)
structural_model_part.AddNodalSolutionStepVariable(KratosMultiphysics.REACTION)

structural_model_part_io = KratosMultiphysics.ModelPartIO("KratosWorkshop2019_high_
rise_building_CSM")
structural_model_part_io.ReadModelPart(structural_model_part)
```

The elements stored in the `ModelPart` can be accessed using the `Elements` parameter:

```
model_part_elements = structural_model_part.Elements
```

Iteration over all elements in a model part is very similar to the nodes. For example writing the ID elements in a model part can be done as follow:

```
for element in model_part_elements:
    print(element.Id)
```

Additionally we can access for example the geometry of the element and ask the area of each element:

```
for element in structural_model_part.Elements:
    print("ID", element.Id, " AREA: ", element.GetGeometry().Area())
```

Conditions parameter of model part provides access to the conditions it stores:

```
model_part_conditions = structural_model_part.Conditions
```

Iteration over conditions is very similar to the elements. In the same way printing the ID conditions is as follow:

```
for condition in model_part_conditions:
    print(condition.Id)
```

Nodes and Nodal Data (I)

First of all we need to create a python file with following code to import the *Kratos*, create a `ModelPart` and read it from input as described in the here :

```
import KratosMultiphysics
import KratosMultiphysics.StructuralMechanicsApplication

this_model = KratosMultiphysics.Model()
structural_model_part = this_model.CreateModelPart("StructuralPart", 3)

structural_model_part.AddNodalSolutionStepVariable(KratosMultiphysics.DISPLACEMENT)
structural_model_part.AddNodalSolutionStepVariable(KratosMultiphysics.REACTION)

structural_model_part_io = KratosMultiphysics.ModelPartIO("KratosWorkshop2019_high_
rise_building_CSM")
structural_model_part_io.ReadModelPart(structural_model_part)
```

The nodes stored in the `ModelPart` can be accessed using the Nodes parameter:

```
model_part_nodes = structural_model_part.Nodes
```

Having access to the nodes make iteration over all nodes very easy. For example to print all nodes in the model part:

```
for node in model_part_nodes:
    print(node)
```

Here is a loop over all of the nodes in a model part, which prints the ID for all of the nodes:

```
for node in model_part_nodes:
    print(node.Id)
```

The coordinates can be accessed by `X`,`Y`,`Z` parameters of the node:

```
node_x = node.X
node_y = node.Y
node_z = node.Z
```

Or we can extend the previous example writing also the coordinates of all the nodes in the `ModelPart`:

```
for node in model_part_nodes:
    print(node.Id, node.X, node.Y, node.Z)
```

This access is very useful in order to classify the nodes due to their position. For example we can extend the previous loop to write node information exclusively on the nodes with positive X:

Nodes and Nodal Data (II)

```
for node in model_part_nodes:
    if(node.X > 0.0): # Printing the ID of all of the nodes with positive X
        print(node.Id, node.X, node.Y)
```

The *Python* interface provides full access to the nodal database. The access to the historical variables is given by `GetSolutionStepValue` and `SetSolutionStepValue` passing the variable you want:

```
node_displacement = node.GetSolutionStepValue(DISPLACEMENT) # node's displacement at the current time step
```

We can write the displacements of all the nodes:

```
for node in model_part_nodes:
    node_displacement = node.GetSolutionStepValue(DISPLACEMENT) # node's displacement at the current time step
    print(node_displacement)
```

you can also get a value for n time step ago, where n is the buffer size:

```
node_previous_displacement = node.GetSolutionStepValue(DISPLACEMENT, 1) # node's displacement at 1 time step ago
node_earlier_displacement = node.GetSolutionStepValue(DISPLACEMENT, 2) # node's displacement at 2 time step ago
```

For getting the previous time step displacements of all the nodes:

```
for node in model_part_nodes:
   print(node.GetSolutionStepValue(DISPLACEMENT, 1)) # node's displacement at 1 time step ago
```

To set the historical value for a variable in a node we can use the `SetSolutionStepValue`. To make an example let's assume that we want to set the variable `DISPLACEMENT_X` to the value of `1.0e-6` on the nodes in our `ModelPart`. This is obtained immediately by typing

```
for node in model_part_nodes:
    node.SetSolutionStepValue(DISPLACEMENT_X,0,1.0e-6)
```

To set the non-historical value for a variable in a node we can use the `SetValue`. Later this can be accessed with `GetValue`. For example:

```
for node in model_part_nodes:
    node.SetValue(TEMPERATURE,100.0)
    print(node.GetValue(TEMPERATURE))
```

References

- *P. Dadvand, R. Rossi, E. Oñate*: *An Object-oriented Environment for Developing Finite Element Codes for Multi-disciplinary Applications.* Computational Methods in Engineering. 2010
- *Python Tutorials*: https://github.com/KratosMultiphysics/Kratos/wiki/Python-Tutorials
- *Data Management*: https://github.com/KratosMultiphysics/Kratos/wiki/Data-management
- *Kratos API*: https://github.com/KratosMultiphysics/Kratos/wiki/Kratos-classes-accesible-via-python

Thank you very much for your attention