ExaQUte

Exascale Quantification of Uncertainties for Technology and Science Simulation

D1.1 Solvers "stub" implementation of the capabilities to be delivered

Document information table

Contract number:	800898
Project acronym:	ExaQUte
Project Coordinator:	CIMNE
Document Responsible Partner:	CIMNE
Deliverable Type:	Report, Other
Dissemination Level:	CO
Related WP & Task:	WP 1 Task 1.1
Status:	Final version

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No **800898**

Authoring

Prepared by: Riccardo Rossi				
Authors	Partner	Modified Page/Sections	Version	Comments
Riccardo Rossi	CIMNE		V0	Creation of the document
Riccardo Rossi	CIMNE	Pages 1-8	V1	Added main body
Inigo Lopez	TUM	Pages 1-8	V2	Added some corrections

Change Log

Versions	Modified Page/Sections	Comments
V0	Creation of the document	
V1	Added main body	
V2	Added some corrections	

Approval

Aproved by:				
	Name	Partner	Date	OK
Task leader	Riccardo Rossi	CIMNE	30.7.18	OK
WP leader	Riccardo Rossi	CIMNE	30.7.18	OK
Coordinator	Riccardo Rossi	CIMNE	30.7.18	OK

Executive summary

The current deliverable describes the initial API available for the solvers. The API is intended to be based on the Kratos Multiphysics fraemework and will evolve during the project. The current deliverable describes the essential features of the interface and provides an initial working implementation to be used as a basis for the future developements.

The initial implementation described here is currently operative on the master branch of Kratos. As of the end of July 2018 (moment of handing in of current deliverable) the interface is operative and is being used “in production”. Nevertheless, it still does not fully support the model serialization capabilities that are needed for pyCompSS and HyperLoom.

The interface is documented in the project wiki page [wiki]. The same documentation is also presented in the current deliverable.

Table of contents

1 AnalysisStage

1.1 Introduction

Solving a problem with Kratos is divided into two Python-objects : The **AnalysisStage** and the **PythonSolver**.

The "PythonSolver" is responsible for everything related to the physics of the problem (e.g. how to setup the system of equations), whereas the "AnalysisStage" is related to everything that is not related to the physics (e.g. when and what output to write).

This means that coupling of physics is done on solver level. A coupled solver would contain the solvers of the involved physics as well as for example the coupling logic, the data exchange, etc. The coupling of things not related to physics is done in the "AnalysisStage", this includes for example the combined output.

The "PythonSolver" is design to act as a member of the "AnalysisStage", therefore the "AnalysisStage" can be seen as "outer" layer and the "PythonSolver" as "inner" layer.

1.2 AnalysisStage

1.2.1 AnalysisStage: Overview

The baseclass of the "AnalysisStage" is located in the KratosCore ([KratosKore]). It provides a set of functionalities needed to perform a simulation. Applications should derive from this object to implement things specific to the application.

In coupled simulations everything that is not related to the physics of a problem is done in the "AnalysisStage". This can be for example a special user scripting or a combined output.

These derived classes replace what was formerly done in "MainKratos.py". This means that also the user scripting should be in these classes. The idea is that instead of having a custom "MainKratos.py", the user derives a class from the "AnalysisStage" of the application to be used. Only the functions that require modifications are being overridden, the remaining implementation is used from the baseclass. In this way, updates to the baseclass are automatically being used in the users custom “AnalysisStage”.

1.2.2 AnalysisStage: Responsibilities and provided Functionalities

The “AnalysisStage” handles everything not related to the physics of the problem. This includes for instance:

- Managing and calling the “PythonSolver”
- Construction and handling of the Processes
- Managing the output (post-processing in GiD/h5, saving restart, ...)

The main **public** functions are listed together with a brief explanation in the following. For a more detailed explanation it is referred to the docstrings of the respective functions.

- **Run** this function executes the entire simulation
- **Initialize** this function initializes the ‘AnalyisStage‘, i.e. it performs all the operations necessary before the solution loop
- **RunSolutionLoop** this function runs the solution loop
- **Finalize** this function finalizes the ‘AnalyisStage‘, i.e. it performs all the operations necessary after the solution loop

  The main **protected** functions that are supposed to be used in derived classes are:
- **_GetSolver** : This function returns the “PythonSolver”. It also internally creates the “PythonSolver”, if it does not exist yet
- **_GetListOfProcesses** : This function returns the list of processes. It also internally creates it if it does not exist yet
- **_GetListOfOutputProcesses** : This function returns the list of output processes. It also internally creates it, if it does not exist yet

1.2.3 AnalysisStage: Usage

In order to use the “AnalysisStage” it has to be constructed with specific objects:

- ‘KratosMultiphysics.Model’: The model containing all the modelparts involved in a simulation ([WIP](Model))
- ‘KratosMultiphysics.Parameters’: The settings for the simulation. The following settings are required to be present:
  - ‘problem_data’ : general settings for the simulation
  - ‘solver_settings’ : settings for the “PythonSolver”
  - ‘processes’ : regular processes, e.g. for the boundary conditions

- ‘output_processes’ : processes that write the output

```
{
"problem_data" : {
    "echo_level"     : 0
    "parallel_type" : "OpenMP" # or "MPI"
    "start_time"     : 0.0,
    "end_time"       : 1.0
},
"solver_settings" : {
...
settings for the PythonSolver
...
},
"processes" : {
    "my_processes" : [
    list of Kratos Processes
    ],
    "list_initial_processes" : [
    list of Kratos Processes
    ],
    "list_boundary_processes" : [
    list of Kratos Processes
    ],
    "list_custom_processes" : [
    list of Kratos Processes
    ]
},
"output_processes" : {
    "all_output_processes" : [
    list of Kratos Output Processes
    ]
}
```

Objects deriving from the “AnalysisStage” have to implement the ‘_CreateSolver’ function which creates and returns the specific “PythonSolver”

Note: If the order in which the processes-blocks are initialized matters (if e.g. some processes would overwrite settings of other processes), then the function ‘_GetOrderOfProcessesInitialization‘ (resp. ‘_GetOrderOfOutputProcessesInitialization‘) has to be overridden in the derived class. This function returns a list with the order in which the processes will be initialized.

As example we consider the settings above: We want the processes “list_initial_processes” to be constructed first and “list_custom_processes” to be constructed second. The order in which the other processes are initialized does not matter. In this case we have to override the ‘_GetOrderOfProcessesInitialization’ function to return ‘[“list_initial_processes”, “list_custom_processes”]’. With this we achieve the desired behavior.

2 PythonSolver

2.1 PythonSolver: Overview

The baseclass of the “PythonSolver” is located in the KratosCore ([KratosKore]). It provides a set of functionalities that are needed for solving a physical problem. Applications should derive from this object to implement the application-specific tasks.

If physics are being coupled (e.g. for Fluid-Structure Interaction) then this should be implemented on solver-level. The coupled solver used the solvers of the involved physics and does also other tasks such as coupling logic or data exchange. E.g. an FSISolver would have a fluid and a structural solver.

2.2 PythonSolver: Responsibilities and provided Functionalities

The “PythonSolver” is responsible for everything related to the physics of a problem. This includes for instance:

- Setting up and solving of the system of equations
- Importing and preparing the ModelPart
- Advancing in time

The main **public** functions are listed together with a brief explanation in the following. For a more detailed explanation it is referred to the docstrings of the respective functions.

- **AddVariables** : this function adds the variables needed in the solution to the ModelPart
- **AddDofs** : this function adds the dofs needed in the solution to the ModelPart
- **ImportModelPart** : this function imports the ModelPart used by the solver (e.g. form an mdpa- or a restart-file)
- **PrepareModelPart** : this function prepares the ModelPart to be used by the solver (e.g. create SubModelParts necessary for the solution)
- **AdvanceInTime**: this function advances the “PythonSolver” in time
- **Initialize**: this function initializes the “PythonSolver”
- **Predict**: this function predicts the new solution
- **InitializeSolutionStep**: this function prepares solving a solutionstep
- **SolveSolutionStep**: this function solves a solutionstep
- **FinalizeSolutionStep**: this function finalizes solving a solutionstep
- **Finalize**: this function finalizes the “PythonSolver”

2.3 PythonSolver: Usage

In order to use the "PythonSolver" it has to be constructed with specific objects:

- 'KratosMultiphysics.Model': The model to be used by the "PythonSolver"
- 'KratosMultiphysics.Parameters': The settings for the "PythonSolver". They are expecting that the following settings are present:
  - 'echo_level' : echo_level for printing informations
  - 'model_import_settings' : settings for importing the modelpart

```
{
"echo_level" : 0,
"model_import_settings" : {
    "input_type"       : "mdpa" # or "rest"
    "input_filename"   : "input_file_name"
}
```

For importing the ModelPart it can also be necessary (depending on the details of the solver) to pass the name of the ModelPart such that it can interact correctly with the Model.

2.4 Outlook (Kratos-Project, Multi-Stage Simulation)

Note: This is a collection of ideas, to be done AFTER AnalysisStage and Solver are implemented in a first version. Please note that the following is in a very early design phase.

In the future the objects presented here can be used in a larger context, e.g. a Multi-Stage Analysis. This means that e.g. a FormFinding Analysis can be performed with doing a FSI-simulation afterwards. The above mentioned objects are already designed for this, e.g. a ModelPart can be passed from outside to the AnalysisStage, this means that it can be used in severals AnalysisStages.

The idea is that in the beginning all AnalysisStages are constructed (i.e. all necessary Variables are added to the ModelPart), then the ModelPart is being read. This can be done e.g. by a global ModelManager. For this to work, the 'Model' has to be enhanced, therefore it should be done later.

This could look like this:

```
import KratosMultiphysics

##construct all the stages

Model = KratosMultiphysics.Model()
list_of_analysis_stages = GenerateStages(Model,
                                          "ProjectParameters.json")
#internally loads the applications needed

for stage in list_of_analysis_stages:
    stage.Initialize()
    stage.Run()
    stage.Finalize()
```

3 Kratos “Model” Object

The “Model” is a new kratos entity that is designed as a container (and memory manager) of all of the kratos “model_parts”. It essentially includes all of the finite element data needed to run a complex example.

An example of use of the model can be found [here].

A fundamental feature of the “Model” is to be **serializeable**. Since the model is one of the input parameters of the stage, this constitutites a vital feature for the ExaQUte application. It is planned to have the feature working by the end of september 2018.

4 Kratos “Parameters” Object

The “Parameters” object is in charge of passing the input parameters for the analsysis. It can also be modified dynamically thus allowing to return simulation data. The syntax of the parameters is based on the “json” format and the object **allows serialization**. The Parameter object also has validation capabilities. The user interface is described [here].