ExaQUte

Exascale Quantification of Uncertainties for Technology and Science Simulation

D2.3. Adjoint-based error estimation routines

Document information table

Contract number:	800898
Project acronym:	ExaQUte
Project Coordinator:	CIMNE
Document Responsible Partner:	TUM
Deliverable Type:	OTHER: Report & Software
Dissemination Level:	Public
Related WP & Task:	WP2, Task 2.3
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:				
Authors	Partner	Modified Page/Sections	Version	Comments
Brendan Keith	TUM			
Contributors				
Andreas Apostolatos	TUM			
Anoop Kodakkal	TUM			
Riccardo Rossi	CIMNE			
Riccardo Tosi	CIMNE			
Barbara Wohlmuth	TUM			

Change Log

Versions	Modified Page/Sections	Comments

Approval

Aproved by:				
	Name	Partner	Date	OK
Task leader	Barbara Wohlmuth	TUM	05/30/2019	OK
WP leader	Brendan Keith	TUM	05/30/2019	OK
Coordinator	Riccardo Rossi	CIMNE	05/30/2019	OK

Executive summary

This document presents a simple and efficient strategy for adaptive mesh refinement (AMR) and *a posteriori* error estimation for the transient incompressible Navier–Stokes equations. This strategy is informed by the work of Prudhomme and Oden [22, 23] as well as modern goal-oriented methods such as [5]. The methods described in this document have been implemented in the Kratos Multiphysics software and uploaded to `https://zenodo.org` [27].[1]

This document includes:

- A review of the state-of-the-art in solution-oriented and goal-oriented AMR.
- The description of a 2D benchmark model problem of immediate relevance to the objectives of the ExaQUte project.
- The definition and a brief mathematical summary of the error estimator(s).
- The results obtained.
- A description of the API.

[1]This release of the Kratos software with DOI (10.5281/zenodo.3235261) can be found at the following link: `https://zenodo.org/record/3235261`.

Table of contents

List of Figures

Nomenclature / Acronym list

Acronym	Meaning
AMR	Adaptive mesh refinement
API	Application Programming Interface
ExaQUte	EXAscale Quantification of Uncertainties for Technology and Science Simulation
QoI	Quantity of Interest
MC	Monte Carlo
MLMC	Multilevel Monte Carlo method
C-MLMC	Continuation Multilevel Monte Carlo method
HPC	High performance computing
PDE	Partial differential equation

1 Introduction

In this section, we give a brief overview of contemporary research on *a posteriori* error estimation and adaptive mesh refinement and their uses within the purview of the ExaQUte project.

1.1 *A posteriori* error estimation

The field of *a posteriori* error estimation in finite element methods began with the Ph.D. work of Ladevèze [17] and the research of Babuška et al. [2, 3]. These early ideas have prospered throughout the intervening decades and have been applied to a plethora of problems of engineering interest [1, 28].

There are two *classical* types of error estimators in computational engineering [1]: residual estimators [2] and recovery estimators [30–32]. Both of these are concerned with discerning the accuracy of numerical approximations in global "energy norms" endemic to the given problem. In addition, one may consider adjoint-based error estimators which result from an optimal control approach to *a posteriori* error estimation [4, 12] and attempt to estimate the error in quantities of interest to the engineer. We will refer to the former (energy-based) class as *solution-oriented* error estimators and the latter (adjoint-based) class as *goal-oriented* error estimators.

Recovery estimators have been used in many applications with Kratos and have a long history of use in *a posteriori* error estimation, in particular with the Navier–Stokes equations beginning with [29]. The support for residual estimators in Kratos is limited compared to its support for recovery estimators and the ExaQUte project presents a convenient opportunity to enlarge the support for this alternative. Therefore, in this document, we shall mainly a focus on the residual type when describing solution-oriented error estimation.

In the ExaQUte project, the objective of *a posteriori* error estimation is to two-fold: (1) to estimate and control the accuracy of individual numerical simulations performed throughout the project; and (2) to assist in the development of a hierarchy of adaptively refined meshes $\{\mathcal{T}_k\}$ for use in multi-level Monte Carlo (MLMC) algorithms. Although inevitably linked objectives, the former mainly manifests in asserting the stopping criteria for and certifying the accuracy of individual algorithms. Meanwhile, the latter is central to the composition of discretizations used in these algorithms. This is the subject of the next subsection.

1.2 Adaptive mesh refinement

Adaptive mesh refinement (AMR) strategies seek to generate optimized approximation spaces to improve the accuracy and efficiency of numerical methods. In mathematical literature, the most pervasive strategy for AMR can be described by the following simple loop [10, 11]:

$$\ldots\text{solve; estimate; mark; refine; }\ldots \tag{1}$$

This procedure generates a nested sequence of meshes wherein each new mesh is can be viewed as an enriched version of the previous mesh. An alternative, which is often seen in engineering applications, is also referred to as *adaptive remeshing*. Here, at every refinement step a completely new mesh is generated by reading a (scalar) weight or (tensor) *metric* which has been assigned to the entire computational domain. The standard adaptive remeshing procedure is similar to (1), however, instead of the marking step, the new *metric* must be computed.

In some contexts, nested mesh adaptivity is desirable because the generation of new meshes is usually faster and geometry changes at each refinement step are usually very small. Adversely, remeshing may may deliver an optimized mesh in a shorter series of iterations, especially when anisotropic meshes are necessary or when a good initial mesh is not known. Kratos only supports adaptive remeshing, therefore this will be our principal focus. Neverthess, for completeness, we now briefly review the state-of-the-art in nested mesh adaptivity.

1.2.1 Nested mesh adaptivity

Normally, at the first stage of the "estimate" step, a special dedicated estimate of the global error, η_Ω, is computed. If this estimate is below a specific tolerance `TOL`, the loop is broken. Otherwise, it continues until plenteous time or computational resources have been consumed. In some scenarios, η_Ω comes in the form of special dedicated bounds on the solution error [18, 21, 25, 26]. Other times, it is simply derived from the accumulation of element-wise error estimates, η_K, which drive the refinement process, otherwise known as *refinement indicators*. For simplicity, we will make the following simplifying assumption, holding to the latter scenario: $\eta_\Omega^2 = \sum_{K\in\mathcal{T}} \eta_K^2$, where $\mathcal{T}$ is the given finite element mesh.

The abstract AMR strategy we will consider is given in Algorithm 1. Note that, at least for the time being, we do not wish to consider anisotropic refinement or mesh coarsening. Therefore, in step (4), it is assumed that each marked element is isotropically refined in a consistent and stable way; see, e.g., [7]. This algorithm can be generalized in a straightforward way when multiple sets of refinement indicators $\{\eta_K^1\}_{T\in\mathcal{T}}$, $\{\eta_K^2\}_{T\in\mathcal{T}}, \ldots$ are used, as can appear naturally in the analysis of coupled PDEs (cf. Sections 2.2 and 2.3).

Algorithm 1 Adaptive mesh refinement

```
Input: initial mesh $\mathcal{T}_0$, tolerance TOL, a marking strategy.
  $i \leftarrow 0$.
  loop
      (1) Solve the discrete problem on $\mathcal{T}_i$.
      (2) Compute the refinement indicators $\{\eta_K\}_{K\in\mathcal{T}_i}$.
      if $\eta_\Omega <$ TOL then
          break
      (3) Mark a set of elements $\mathcal{M} \subset \mathcal{T}_i$, as dictated by the marking strategy.
      (4) Refine all marked elements $M \in \mathcal{M}$ and construct the next mesh $\mathcal{T}_{i+1}$.
      (5) $i \leftarrow i + 1$.
  return solution $u_h$.
```

1.2.2 Marking strategies

In step (3) of Algorithm 1, the construction of $\mathcal{M} \subset \mathcal{T}$ can be performed in numerous ways. Here, we outline three common strategies which we plan to investigate in the ExaQUte.

Fixed proportion marking. In some applications, it is desirable to ensure a fixed rate of growth in computational complexity between each mesh $\mathcal{T}_k$ and $\mathcal{T}_{k+1}$. This objective readily manifests in the following simple marking strategy:

Marking strategy 1. Fixed proportion marking

Input: constant $0 < \theta < 1$.

Define $\mathcal{M}$ to be the largest subset of $\mathcal{T}$ such that

$$|\mathcal{M}| \leq \theta \cdot |\mathcal{T}| \quad \text{and} \quad \eta_M > \eta_{M'} \text{ for each } \eta_M \in \mathcal{M} \text{ and } \eta_{M'} \in \mathcal{T} \setminus \mathcal{M}\,.$$

The maximum strategy Of course, in some problems it may be a drawback if the same proportion of elements are added at each refinement step. For instance, if as the mesh is developed new features appear in the solution, which were not present at coarse scales, then naturally the proportional of new elements should change with the refinement level. This issue is partially avoided if the refinement criteria is based on the relative values of the refinement indicators, such as in the following strategy:

Marking strategy 2. Maximum value marking

Input: constant $0 < \theta < 1$.

Define

$$\eta_{\max} = \max_{K\in\mathcal{T}} \eta_K\,.$$

Define $\mathcal{M}$ to the largest subset of $\mathcal{T}$ such that

$$\eta_M \geq (1-\theta)\cdot \eta_{\max} \quad \text{for all } M \in \mathcal{M}\,.$$

Dörfler marking An important improvement on Marking Strategy 2 was made in [10] which, with certain PDEs, can be used to prove the recovery of optimal convergence rates in the presence of some singularities. This strategy is given as follows:

Marking strategy 3. Dörfler marking

Input: constant $0 < \theta < 1$.

Define

$$\eta_\Omega = \left(\sum_{K\in\mathcal{T}} \eta_K^2\right)^{1/2}.$$

Define $\mathcal{M}$ to be the smallest subset of $\mathcal{T}$ such that

$$\sum_{M\in\mathcal{M}} \eta_M^2 \geq \theta \cdot \eta_\Omega^2\,.$$

Remark 1. In general, every refinement indicator η_K will be non-negative. Therefore, notice that each of the Marking Strategies 1–3 reproduce or mimic uniform refinement in the limit $\theta \to 1$. Conversely, no refinements are usually performed in the limit $\theta \to 0$. When setting the bulk parameter $\theta \in (0, 1)$, these dependences should be kept in mind.

Remark 2 In order to construct the maximal or minimal sets $\mathcal{M} \subset \mathcal{T}$ in Marking Strategies 1 or 3, respectively, a global sort of the set of refinement indicators $\{\eta_K\}_{K\in\mathcal{T}}$ is usually necessary.

1.2.3 Adaptive remeshing

In a two-dimensional adaptive remeshing strategy, one often designs a metric $\mathbf{g} : \Omega \to \mathbb{R}^{2\times 2}_{\text{sym.}}$ encoding the desired node spacing $\delta(\boldsymbol{x})$, aspect ratio $r(\boldsymbol{x})$, and alignment direction $\boldsymbol{a}(\boldsymbol{x})$, which vary through each point $\boldsymbol{x}$ in the domain. For instance, let $\lambda_i(\boldsymbol{x})$, $i = 1, \ldots, 2$, taken in decreasing order, be the eigenvalues of $\mathbf{g}(\boldsymbol{x})$ and let $\boldsymbol{v}_i(\boldsymbol{x})$ be the associated eigenvector field. Generally, $\delta \propto 1/\sqrt{\lambda_1}$, $r \propto \sqrt{\frac{\lambda_1}{\lambda_2}}$, and $\boldsymbol{a}(\boldsymbol{x}) \propto \boldsymbol{v}_1(\boldsymbol{x})$. At each refinement step, instead marking in step (3) in Algorithm 1, the metric $\mathbf{g}(\boldsymbol{x})$ is updated. Generalization to three dimensions is straightforward.

Usually, the metric is constructed using second order information derived from an *a posteriori* error estimator. Let $\mathbf{I} \in \mathbb{R}^{2\times 2}$ be the identity matrix and $f(\boldsymbol{x})$ be some scalar-valued function. In this report, we use only first order information, which restricts the type of metric structure we may consider. Specifically, we only account for tensors of the form[2] $\mathbf{g}(\boldsymbol{x}) = f(\boldsymbol{x}) \cdot \mathbf{I}$. Therefore, all of the meshes we will be able to produce will be *isotropic*; i.e., $r(\boldsymbol{x}) = 1$.

Over the union of all elements in the mesh K, let $\tilde{\eta} : \bigcup \overline{K} \to \mathbb{R}$ be the linear interpolant of the piecewise constant function $K \ni \boldsymbol{x} \mapsto \eta_K$ and let N be the number of elements in the mesh. The function $f(\boldsymbol{x})$ is constructed as follows: First, fix $\phi \in (0, 1)$ and define the piece-wise linear mesh density factor

$$C(\boldsymbol{x}) = \frac{\phi \cdot \eta_\Omega}{N^{1/2} \cdot \tilde{\eta}}\,,$$

which is a scaling factor penalizing the deviation of the local element error estimate from mean element error. Here, the factor ϕ serves to make the average element size throughout the domain change by a factor of ϕ, after each refinement.

The function $f(\boldsymbol{x})$ could be set directly in terms of $h_{\text{curr}}(\boldsymbol{x}) \cdot C(\boldsymbol{x})$, where h_{curr} is the nodal element size function for the current mesh. However, if $C(\boldsymbol{x})$ is too large or small in some regions of the domain, this may induce too rapidly graded meshes. Therefore, we use an additional tolerance $C_0 \in \mathbb{R}$ to bound this density factor and, thereby, define the restricted mesh density factor $C_0(\boldsymbol{x}) = \min\{\max\{C(\boldsymbol{x}), 1/C_0\}, C_0\} \in [1/C_0, C_0]$.

Now define the mesh density function $h(\boldsymbol{x}) = h_{\text{curr}}(\boldsymbol{x}) \cdot C_0(\boldsymbol{x})$. It is also a potential issue if the element size becomes too small or large. Therefore, we must introduce limits on these values: $h_{\texttt{safe}}(\boldsymbol{x}) = \min\{\max\{h(\boldsymbol{x}), h_{\min}\}, h_{\max}\}$. Finally, set $f(\boldsymbol{x}) = h_{\texttt{safe}}(\boldsymbol{x})^{-2}$.

[2] In Euclidean coordinates.

1.3 Dynamic spatial meshes

In transient problems, one has the choice of whether or not to dynamically update the spatial mesh $\mathcal{T}$; i.e., reconstruct a new mesh after each (or a certain number) of time steps [29]. Although such dynamic AMR strategies ultimately optimize the mesh distribution over each spatial refinement window, they can easily become prohibitively expensive on modern computing systems [9]. This is due in part to the extra computational resources necessary for mesh generation and load balancing in distributed environments and, in turn, requires numerous considerations in order to control runtimes.

In taking this and ExaQUte objectives into account, one of the most important AMR studies to compare with is [5]. From their paper, we reprint their justification for avoiding dynamic AMR in the confined cylinder flow problem, which has many important similarities to the problems to be encountered in within the ExaQUte project:

> "We do not use dynamic spatial meshes in this example for two reasons. On the one hand, the use of dynamic meshes leads to wrong approximations of the drag coefficient if no additional projection steps are applied each time the spatial mesh is changed, which would be rather costly (see Besier and Wollner [6] for a discussion of this problem). The other reason becomes clear if we have a look at Figure 9...We observe that in order to precisely determine the mean drag coefficient, it is not necessary to resolve the whole van Kármán vortex street. Only a small recirculation zone behind the obstacle is strongly refined. As the vortices in this region develop relatively early, we may conclude that allowing dynamic meshes would not provide a notable reduction in the degrees of freedom needed to reach the same accuracy as with adaptively refined but fixed spatial meshes. In view of the additional effort on dynamic meshes due to more frequent matrix reassembling and the additional projection steps, we reach the conclusion that the use of dynamic spatial meshes does not make sense in this particular flow example."

For similar reasons, we choose to neglect dynamic spatial remeshing at this stage of the ExaQUte project.

1.4 Adjoint-based strategies and other extensions

The vast majority of early work on *a posteriori* error estimation was focused on estimates of the global solution error in deterministic problems. However, in the late 90s, an adjoint-based (goal-oriented) error estimation theory devoted to the error in functional outputs of computer simulations—so-called quantities of interest (QoI)—was developed by Oden and Prudhomme [19, 24], Becker and Rannacher [4], Patera and Peraire [21], and Giles and Süli [12]. More recently, sophisticated *a posteriori* error estimation techniques have also been developed for stochastic problems with parametric and domain uncertainties [13, 14, 20]. However, this theory is still limited in nonlinear or transient settings.

In general, goal-oriented AMR strategies deliver far superior performance in problems where the error in the QoI has outsize sensitivity to localized solution features which are not prioritized by traditional solution-oriented AMR algorithms. Conversely, in some other scenarios, there is little advantage in goal-oriented AMR over solution-oriented AMR or, in some cases, even uniform refinement [4]. For the ExaQUte objectives, one

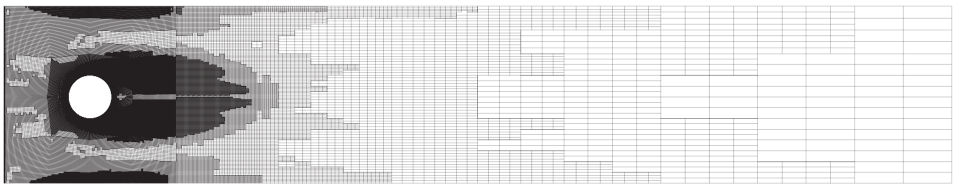


Figure 1: An adaptively generated mesh from a goal-oriented AMR algorithm for the Navier–Stokes equations. Reproduced from [5].

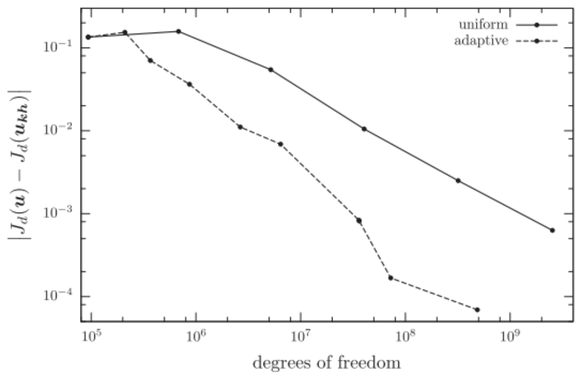


Figure 2: Goal-oriented AMR versus uniform refinement for the confined cylinder problem. Reproduced from [5].

of the most important studies into adjoint-based strategies is [5] (cf. Section 1.3). Here, a sophisticated dual-weighted residual (DWR) method [4, Section 5] is proposed for the transient Navier–Stokes equations in 2D. Moreover, it is analyzed on the canonical confined cylinder problem where the QoI is the drag coefficient on the cylinder.

In Figure 1, we have reproduced an adaptively generated mesh from one of the confined cylinder experiments in [5]. This mesh was generating by a goal-oriented AMR strategy *without dynamic remeshing* where the quantity of interest was the drag coefficient on the cylinder. Although their results were not compared with a solution-oriented approach, we clearly see several notable features in the mesh pattern which often appear in solution-oriented AMR with this class of problems; see, e.g., [15, 16, 24]. For instance, one sees the finest mesh scales near the cylinder; namely, in a region around the boundary of the object, in its immediate wake, and along the nearest regions of the channel wall.

In Figure 2, we reproduce the corresponding convergence plot comparing the goal-oriented AMR strategy with uniform refinements. Here, we see only about a one order of magnitude gain in efficiency at the 1% error mark and nearly a two order of magnitude gain at the 0.1% error mark. Even though solution-oriented AMR was never compared with these results in this study, given the mesh pattern depicted in Figure 1, one would obviously expect the accuracy of solution-oriented AMR to be far more similar to results from goal-oriented refinement rather than uniform refinement.

At best, the results in [5] are not persuasive that goal-oriented AMR will be a benefit to the objectives of the ExaQUte project. Therefore, we have decided to avoid this investing in this feature.

We now offer a partial heuristic justification for the drawback of goal-oriented refinement in this particular example. In such problems, consider that large errors in the

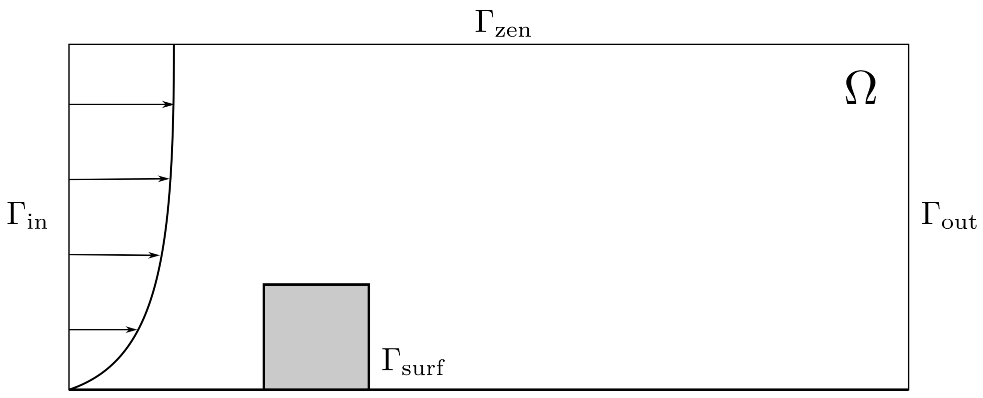


Figure 3: Benchmark domain Ω and boundaries.

downstream solution usually have little influence on the accuracy of quantities measured upstream. On the contrary, large upstream errors can easily propagate in a way which ruins the accuracy of downstream quantities. This may be a great concern when trying to construct an optimal mesh for a flow over of sequence of bluff bodies when a quantity such as the drag on a single body is of interest. However, when only a single object is impeding the flow, the influences are much simpler to discern because every boundary layer is generated near the object in question. In this case, the boundary layers will be resolved with similar accuracy with a solution-oriented strategy or a goal-oriented strategy (at least one based on a quantity measured in a vicinity of the object) because both will emphasize every boundary layer present in the solution.

2 Problem statement

In the ExaQUte project, the PDEs of principle concern are the incompressible Navier–Stokes equations, given as follows:

$$\frac{\partial \mathbf{u}}{\partial t} + (\mathbf{u} \cdot \nabla)\mathbf{u} = -\nabla p + \nu \nabla^2 \mathbf{u}\,,$$
$$\nabla \cdot \mathbf{u} = 0\,.$$

In this section, we will use these equations to define a simple benchmark problem and propose a straightforward error estimator for use in AMR during its analysis.

2.1 Set-up

In the ExaQUte project, we will be simulating wind flow past large complicated structures under calibrated physical conditions. Figure 3 is an extremely simplified 2D representation of this scenario. Here, the impeding object is simply a square, which leaves behind the computational domain Ω.

On the inflow boundary, Γ_{in}, the following temporal average velocity is prescribed:

$$\mathbf{u}(t, \Gamma_{\text{in}}) \cdot \boldsymbol{n} = \overline{\mathrm{u}}\left(\frac{z}{z_0}\right)^{\alpha}, \qquad \mathbf{u}(t, \Gamma_{\text{in}}) \cdot \boldsymbol{n}^{\perp} = 0\,, \tag{2}$$

where $\boldsymbol{n}$ is the unit normal vector field on $\partial\Omega$ and $\boldsymbol{n}^{\perp}$ is any unit length vector field orthogonal to $\boldsymbol{n}$. In our experiments, we have set $\alpha = 0.12$ and $\mathrm{u} = 10\,\mathrm{ms}^{-1}$. The remaining boundaries, Γ_{surf}, Γ_{zen}, and Γ_{out} have wall, free slip, and zero flux boundary conditions, repectively; i.e.,

$$\begin{aligned} \mathbf{u}(t,\Gamma_{\text{surf}}) &= \mathbf{0}\,, & \boldsymbol{\sigma}(t,\Gamma_{\text{out}})\boldsymbol{n} &= \mathbf{0}\,, \\ \mathbf{u}(t,\Gamma_{\text{zen}})\cdot\boldsymbol{n} &= 0\,, & \boldsymbol{\sigma}(t,\Gamma_{\text{zen}})\boldsymbol{n}\cdot\boldsymbol{n}^{\perp} &= 0\,, \end{aligned} \tag{3}$$

where $\boldsymbol{\sigma} = -p\mathrm{I} + \nu\big(\nabla\mathbf{u} + \nabla\mathbf{u}^{\top}\big)$ is the Cauchy stress tensor.

2.2 A mass conservation error estimator

Solution-oriented *a posteriori* error estimation for the transient Navier–Stokes equations was considered in rigorous detail in [22, 25]. In those works, the finite element residuals of the momentum balance and mass conservation equations are analyzed separately and two separate *a posteriori* error estimators are derived out of these residuals.

From the mass conservation equation residual, for each time step n and element $K \in \mathcal{T}$, they arrive at the following error estimator:

$$\eta^{\text{mass}}_{K,n} = \|\mathrm{div}(\mathbf{u}^n_h)\|_{L^2(K)}\,.$$

Taking the root mean square of this estimator over the time interval $[0,T]$, delivers the following averaged residual estimator:

$$\eta^{\text{mass}}_{K} = \left(\frac{1}{T}\int_0^T \|\mathrm{div}(\mathbf{u}_h(s,\cdot))\|^2_{L^2(K)}\,\mathrm{d}s\right)^{1/2}\,, \tag{4}$$

and its corresponding global error estimator,

$$\eta^{\text{mass}}_{\Omega} = \left(\frac{1}{T}\int_0^T \|\mathrm{div}(\mathbf{u}_h(s,\cdot))\|^2_{L^2(\Omega)}\,\mathrm{d}s\right)^{1/2}\,. \tag{5}$$

2.3 A momentum balance law error estimator

The error estimator above is extremely convenient as it has an immediate physical meaning and can be efficiently computed at the element level. Nevertheless, it is agnostic to all errors in the discrete pressure field p^n_h and errors in the solenoidal part of the velocity field $\mathbf{u}^n_h$. In order to rigorously control these errors, a separate error estimator may also be computed.

By analyzing the momentum balance law, Prudhomme and Oden arrive at the implicit error estimator

$$\eta^{\text{mom.}}_{K,n} = \left(\frac{1}{\Delta t}\|\boldsymbol{\varphi}_h\|^2_{L^2(K)} + \frac{1}{2}\mathrm{Re}^{-1}|\boldsymbol{\varphi}_h|_{H^1(K)}\right)^{1/2}\,,$$

where $\boldsymbol{\varphi}_h$ is the unique solution of an auxiliary problem on an enriched approximation space $\boldsymbol{W}_h \subset \boldsymbol{H}^1(\Omega)$. To define this auxiliary problem, first denote the discrete residual of the momentum balance law by $\mathscr{R}^{\text{mom.}}_h$. The Riesz projection of the residual, $\boldsymbol{\varphi}_h \in \boldsymbol{W}_h$, is then defined as the unique solution to

$$\frac{1}{\Delta t}(\boldsymbol{\varphi}_h,\mathbf{v}) + \frac{1}{2}\mathrm{Re}^{-1}(\boldsymbol{\nabla}\boldsymbol{\varphi}_h,\boldsymbol{\nabla}\mathbf{v}) = \mathscr{R}^{\text{mom.}}_h(\mathbf{v})\,,\quad \text{for all } \mathbf{v}\in\boldsymbol{W}_h\,. \tag{6}$$

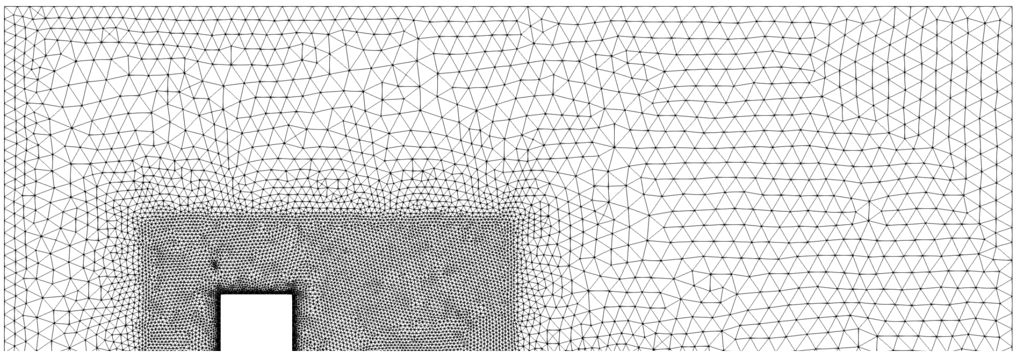


Figure 4: Initial mesh.

The solution of (6) can be made computationally feasible by forming $\boldsymbol{W}_h$ out of a localized approximation space using bubble functions. Otherwise, one may formally consider the limit $\mathrm{Re} \to \infty$ and observe that the difficult $(\boldsymbol{\nabla}\boldsymbol{\varphi}_h, \boldsymbol{\nabla}\mathbf{v})$ vanishes. In this case, one may also consider constructing a different Riesz representation $\tilde{\boldsymbol{\varphi}}_h \in \tilde{\boldsymbol{W}}_h$ solving

$$\frac{1}{\Delta t}(\tilde{\boldsymbol{\varphi}}_h, \mathbf{v}) = \mathscr{R}_h^{\text{mom.}}(\mathbf{v})\,, \quad \text{for all } \mathbf{v} \in \tilde{\boldsymbol{W}}_h\,,$$

where $\tilde{\boldsymbol{W}}_h \subsetneq \prod_{K\in\mathcal{T}} \boldsymbol{H}^1(K)$ is a nonconforming enriched approximation space. This second alternative is convenient for the ExaQUte objectives since the Reynolds number may eventually grow to eight order of magnitude, $\mathrm{Re} \sim 10^8$.

Obviously, (4) can also be extended to averaged local and global residual estimators. Namely,

$$\eta_K^{\text{mom.}} = \left(\frac{1}{T}\int_0^T \frac{1}{\Delta t}\|\boldsymbol{\varphi}_h\|^2_{L^2(K)} + \frac{1}{2}\mathrm{Re}^{-1}|\boldsymbol{\varphi}_h|_{H^1(K)}\,\mathrm{d}s\right)^{1/2},$$

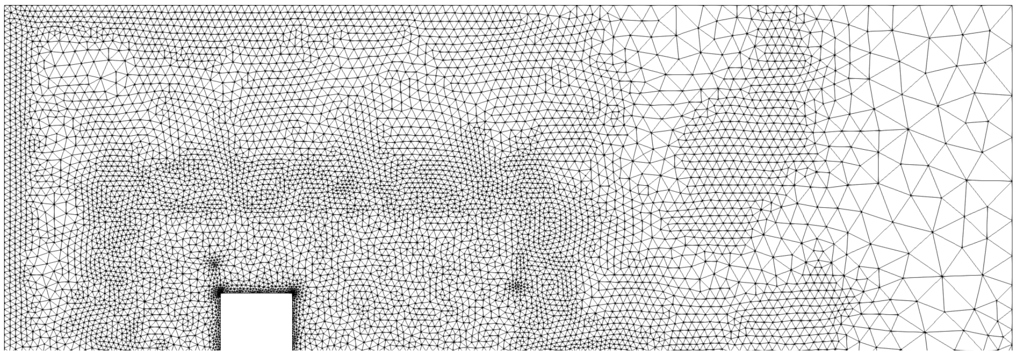
and

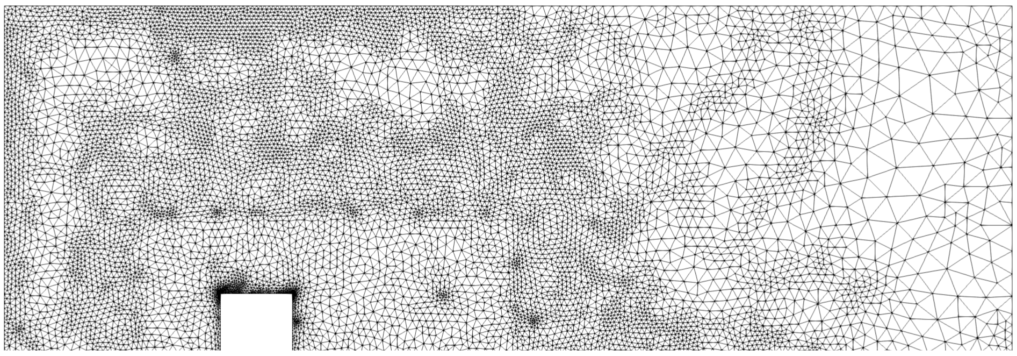
$$\eta_\Omega^{\text{mom.}} = \left(\frac{1}{T}\int_0^T \frac{1}{\Delta t}\|\boldsymbol{\varphi}_h\|^2_{L^2(\Omega)} + \frac{1}{2}\mathrm{Re}^{-1}|\boldsymbol{\varphi}_h|_{H^1(\Omega)}\,\mathrm{d}s\right)^{1/2}.$$

3 Results

In this section, we present results from our adaptive remeshing strategy with a wind field simulation over a five second time interval. Here we start the simulation with a zero initial velocity field and use $\nu = 1.846 \cdot 10^{-5}\,\mathrm{kg/(m\cdot s)}$. Additionally, we begin the simulation with the two-scale initial mesh depicted in Figure 4.

With the parameters $\phi = 0.9$, $C_0 = 2$, $h_{\max} = 5$, and $h_{\min} = 0.01$, the first three successively adapted meshes are depicted in Figure 5. Note that, in order to avoid early transients, the time averages in (4) and (5) did not include the first 20% of the simulation. It is clear from these figures that the boundary layer region near the block is targeted for refinement. However, there clearly appear larger scale spurious refinements in the originally coarsely meshed part of the domain. It is unclear whether these mesh artifacts are a direct result of the adaptive scheme because they do not appear in what was, in the initial mesh, the finely meshed region. *This is an immediate concern which will need to be addressed going forward.*


(a) First mesh.


(b) Second mesh.

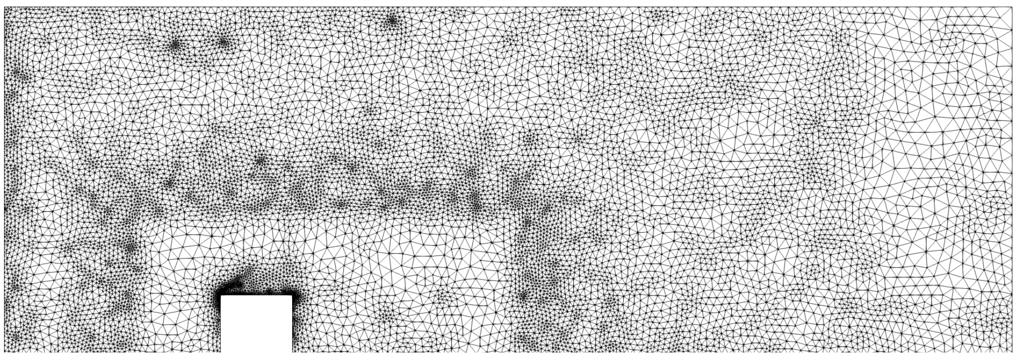

(c) Third mesh.

Figure 5: Adaptively refined meshes.

Snapshots of the solution at $t = 2.5$ are provided in Figure 6 to verify that the persistent boundary layer is indeed captured by the adaptive algorithm.

A API definition and usage

This appendix provides a brief documentation and explanation of the API for the adaptive mesh refinement and how it may be used [27]. As in most other standard Kratos Multiphysics software ®, a simulation invoking adaptive mesh refinement is controlled via a python layer. In Kratos, the remeshing facility is provided by the MMG software library [8]. We will use the file `ProblemZero_ForwardOnly.py`, which runs the Navier–Stokes problem defined in Section 2, in order to demonstrate the API.

The Kratos meshing application may be loaded using the following python import command:

```python
import KratosMultiphysics.MeshingApplication as KratosMeshing
```

After solving the problem, the user may call the divergence error estimator and adaptive mesh remeshing utility in the following way:

```python
    # Calculate divergence metric of the variable
    metric_parameters = KratosMultiphysics.Parameters("""
        {
            "minimal_size": 0.01,
            "maximal_size": 5.0,
            "refinement_strategy": "mean_distribution_strategy",
            "mean_distribution_strategy": {
                "target_refinement_coefficient" : 0.9,
                "refinement_bound"              : 2.0
            }
        }    """)
    local_divergencefree_metric =
    ↪  KratosMeshing.MetricDivergenceFreeProcess2D(fluid_solver._GetSolver().main_model_part,metric_parameters)
    local_divergencefree_metric.Execute()
    print("divergence free metric computed")
```

Currently, only the refinement strategy `"refinement_strategy": "MeanDistributor"` is supported. Clearly, the parameters `minimal_size` and `maximal_size` fix the minimial and maximal sizes of the elements, h_{min} and h_{max}, respectively. Meanwhile, the parameters `target_refinement_coefficient` and `refinement_bound` are, respectively, ϕ and C_0 from Section 1.2.3.

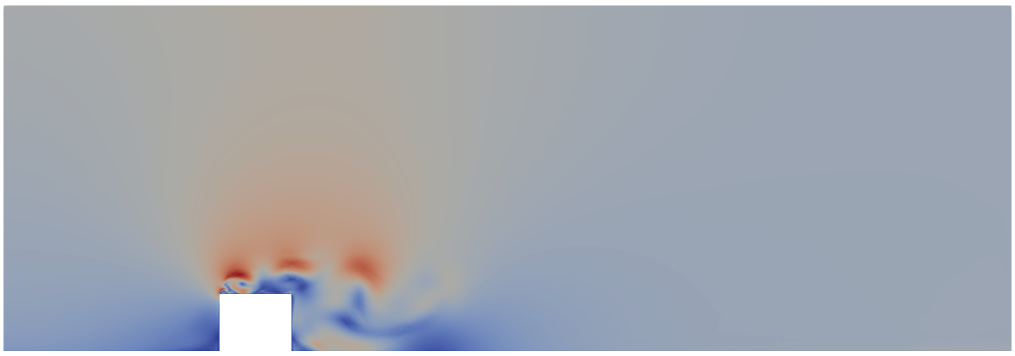
After the metric parameters have been set, the remeshing process is executed in the next sequence of lines:

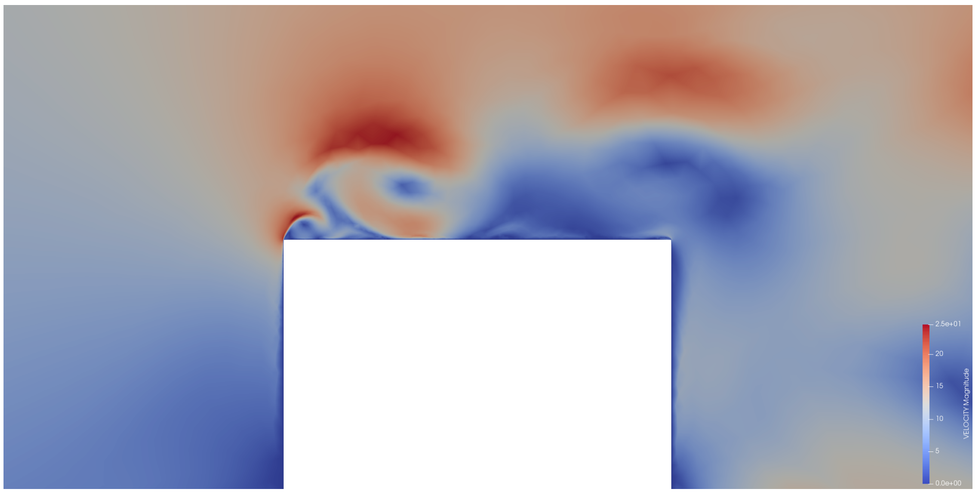
```python
    # Execute remeshing process
    remesh_parameters = KratosMultiphysics.Parameters()
    MmgProcess = KratosMeshing.MmgProcess2D(fluid_solver._GetSolver().main_model_part,remesh_parameters)
    MmgProcess.Execute()
    print("MMG refinement computed")
```

after which, the new mesh and problem information is output to `.gid` and `.mdpa` file using the commands

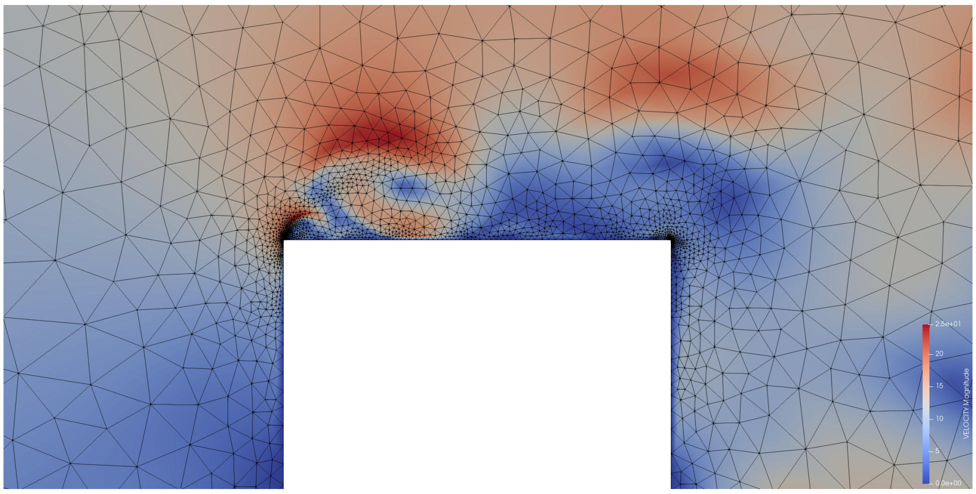
```python
    computational_name = fluid_solver._GetSolver().GetComputingModelPart().Name
    print(computational_name)
    fluid_solver._GetSolver().main_model_part.RemoveSubModelPart(computational_name)

```


(a) Velocity field.



(b) Close-up (without mesh).



(c) Close-up (with mesh).

Figure 6: Velocity field at $t = 2.5$.

```
    # PRINT MDPA AND GID
    CreateGidControlOutput(fluid_solver._GetSolver().main_model_part,"ProblemZero_finaltime")
    KratosMultiphysics.ModelPartIO("ProblemZero_refined", KratosMultiphysics.IO.WRITE |
    ↪  KratosMultiphysics.IO.MESH_ONLY).WriteModelPart(fluid_solver._GetSolver().main_model_part)
```

In most applications, a sequence of meshes will need to be generated. In this case, it is necessary to rerun the simulation the necessary number of times after changing input filename in the .json parameter file `ProblemZeroParameters.json` to `ProblemZero_refined`. Note that in order to run MLMC algorithms, a sequence of refined meshes may be required. The technology to read only once and at coarsest level the model part has already been developed (see deliverable 4.2 and 5.2). The implementation of this benchmark problem into those MLMC routines will be a future concern.

A copy of the complete file `ProblemZero_ForwardOnly.py` now follows.

```
from __future__ import absolute_import, division #makes KratosMultiphysics backward compatible with python
↪  2.6 and 2.7

# Importing the Kratos Library
import KratosMultiphysics

# Import applications
import KratosMultiphysics.FluidDynamicsApplication as KratosFluidDynamics
from fluid_dynamics_analysis import FluidDynamicsAnalysis
import KratosMultiphysics.MultilevelMonteCarloApplication as KratosMLMC
import KratosMultiphysics.MeshingApplication as KratosMeshing

# Avoid printing of Kratos informations
KratosMultiphysics.Logger.GetDefaultOutput().SetSeverity(KratosMultiphysics.Logger.Severity.WARNING) #
↪  avoid printing of Kratos things

# Importing the base class
from analysis_stage import AnalysisStage

# Import packages
import numpy as np

# Import Monte Carlo library
import mc_utilities as mc

# Import cpickle to pickle the serializer
try:
    import cpickle as pickle  # Use cPickle on Python 2.7
except ImportError:
    import pickle

# Import exaqute
from exaqute.ExaquteTaskPyCOMPSs import *   # to execute with pycompss
# from exaqute.ExaquteTaskHyperLoom import *  # to execute with the IT4 scheduler
# from exaqute.ExaquteTaskLocal import *      # to execute with python3
'''
get_value_from_remote is the equivalent of compss_wait_on: a synchronization point
in future, when everything is integrated with the it4i team, importing exaqute.ExaquteTaskHyperLoom you
↪  can launch your code with their scheduler instead of BSC
'''

'''
function evaluating the QoI of the problem: int_{domain} TEMPERATURE(x,y) dx dy
right now we are using the midpoint rule to evaluate the integral: improve!
'''
def EvaluateQuantityOfInterest():
    # TODO AK: Look for a better way to do this
    """here we evaluate the QoI of the problem: """

    fluid_model = KratosMultiphysics.Model()

```

```python
    fluid_project_params_file_name = 'ProblemZeroParameters.json'
    with open(fluid_project_params_file_name,'r') as parameter_file:
        parameters_fluid = KratosMultiphysics.Parameters(parameter_file.read())

    '''
    # --------------------------------------------------------
    # ----- Setting up and initializing the Fluid Solver -----
    # --------------------------------------------------------
    '''
    fluid_solver = FluidDynamicsAnalysis(fluid_model, parameters_fluid)

    fluid_solver.Initialize()

    fluid_model_part = fluid_model["MainModelPart"]
    fluid_inlet_model_part = fluid_model["MainModelPart.AutomaticInlet2D_inlet"]

    print("======================================================================")
    print("|||||||||||||||||||||| SETTING UP FLUID DONE |||||||||||||||||||||||||")
    print("======================================================================")

    # compute the drag
    def CalcDrag(time):
        structure = fluid_model_part.GetSubModelPart('NoSlip2D_structure')
        nodes = structure.Nodes
        rx = 0.0
        ry = 0.0
        rz = 0.0
        for node in nodes:
            reaction = node.GetSolutionStepValue(KratosMultiphysics.REACTION, 0)
            #print(node, reaction)
            rx += -1 * reaction[0]
            ry += -1 * reaction[1]
            rz += -1 * reaction[2]
            # AK : Reaction is negative of the action on the structure
        drag_force = [time,rx,ry,rz]
        return drag_force


    # ----- Solving the problem (time integration) -----
    drag_force_vector = np.zeros([0,4])
    while(fluid_solver.time <= fluid_solver.end_time):

        fluid_solver.time = fluid_solver._GetSolver().AdvanceInTime(fluid_solver.time)

        fluid_solver.InitializeSolutionStep()

        #ApplyInletVelocity(fluid_inlet_model_part, gust_velocity,fluid_solver.time )

        fluid_solver._GetSolver().Predict()
        fluid_solver._GetSolver().SolveSolutionStep()

        fluid_solver.FinalizeSolutionStep()

        drag_force = CalcDrag(fluid_solver.time)
        drag_force_vector = np.vstack((drag_force_vector,drag_force))
        fluid_solver.OutputSolutionStep()

        # disp = tip_node.GetSolutionStepValue(KratosMultiphysics.DISPLACEMENT)
        # file_writer.WriteToFile([time, disp[0], disp[1], disp[2], num_inner_iter])

        ↪  KratosFluidDynamics.WeightedDivergenceCalculationProcess(fluid_solver._GetSolver().main_model_part).Execute()
        print("divergence weighted average computed")
        # for elem in fluid_solver.model.GetModelPart("MainModelPart").Elements:
        #     if (elem.GetValue(KratosFluidDynamics.DIVERGENCE) > 10.0):
        #         print(elem.GetValue(KratosFluidDynamics.DIVERGENCE))


    # Calculate NODAL_H
    find_nodal_h =
    ↪  KratosMultiphysics.FindNodalHNonHistoricalProcess(fluid_solver._GetSolver().main_model_part)
    find_nodal_h.Execute()
    print("nodal h computed")

    # Calculate divergence metric of the variable
    metric_parameters = KratosMultiphysics.Parameters("""
        {
            "minimal_size": 0.01,
            "maximal_size": 5.0,
            "refinement_strategy": "mean_distribution_strategy",
            "mean_distribution_strategy": {
                "target_refinement_coefficient" : 0.9,
                "refinement_bound"              : 2.0
            }
        }    """)
    local_divergencefree_metric =
    ↪  KratosMeshing.MetricDivergenceFreeProcess2D(fluid_solver._GetSolver().main_model_part,metric_parameters)
    local_divergencefree_metric.Execute()
    print("divergence free metric computed")

    # Execute remeshing process
    remesh_parameters = KratosMultiphysics.Parameters()
    MmgProcess = KratosMeshing.MmgProcess2D(fluid_solver._GetSolver().main_model_part,remesh_parameters)
    MmgProcess.Execute()
    print("MMG refinement computed")

    # Remove model part created by the solver
    # TODO: find a smarter way for this removing, but at least working
    computational_name = fluid_solver._GetSolver().GetComputingModelPart().Name
    print(computational_name)
    fluid_solver._GetSolver().main_model_part.RemoveSubModelPart(computational_name)

    # PRINT MDPA AND GID
    CreateGidControlOutput(fluid_solver._GetSolver().main_model_part,"ProblemZero_finaltime")
    KratosMultiphysics.ModelPartIO("ProblemZero_refined", KratosMultiphysics.IO.WRITE |
    ↪  KratosMultiphysics.IO.MESH_ONLY).WriteModelPart(fluid_solver._GetSolver().main_model_part)


    # TIME LOOP END

    fluid_solver.Finalize()

    time_stable= int(0.2 * len(drag_force_vector))
    mean_force_x = np.mean(drag_force_vector[time_stable:, 1])
    print(mean_force_x)

    return mean_force_x


'''
function called in the main returning a future object (the result class) and an integer (the finer level)
input:
        pickled_coarse_model      : pickled model
        pickled_coarse_parameters : pickled parameters
output:
        MonteCarloResults class   : class of the simulation results
        current_MC_level          : level of the current MLMC simulation
'''
def ExecuteMonteCarloAnalysis(pickled_model, pickled_parameters):
    current_MC_level = 0 # MC has only level 0
    return (ExecuteMonteCarloAnalysis_Task(pickled_model, pickled_parameters),current_MC_level)


'''
function executing the problem
input:
        model       : serialization of the model
        parameters  : serialization of the Project Parameters
output:
        QoI         : Quantity of Interest
'''
@ExaquteTask(returns=1)
def ExecuteMonteCarloAnalysis_Task(pickled_model, pickled_parameters):
    '''overwrite the old model serializer with the unpickled one'''
    model_serializer = pickle.loads(pickled_model)
    current_model = KratosMultiphysics.Model()
    model_serializer.Load("ModelSerialization",current_model)
    del(model_serializer)
    '''overwrite the old parameters serializer with the unpickled one'''
    serialized_parameters = pickle.loads(pickled_parameters)
    current_parameters = KratosMultiphysics.Parameters()
    serialized_parameters.Load("ParametersSerialization",current_parameters)
    del(serialized_parameters)
    '''initialize the MonteCarloResults class'''
    current_level = 0 # always 0 for MC
    mc_results_class = mc.MonteCarloResults(current_level)
    QoI = EvaluateQuantityOfInterest()
    mc_results_class.QoI[current_level].append(QoI) # saving results in the corresponding list, for MC
    ↪  only list of level 0
    return mc_results_class


'''
function serializing and pickling the model and the parameters of the problem
the idea is the following:
i)   from Model/Parameters Kratos object to StreamSerializer Kratos object
ii)  from StreamSerializer Kratos object to pickle string
iii) from pickle string to StreamSerializer Kratos object
iv)  from StreamSerializer Kratos object to Model/Parameters Kratos object
input:
        parameter_file_name   : path of the Project Parameters file
output:
        pickled_model      : model serializaton
        pickled_parameters : project parameters serialization
'''
@ExaquteTask(parameter_file_name=FILE_IN,returns=2)
def SerializeModelParameters_Task(parameter_file_name):
    with open(parameter_file_name,'r') as parameter_file:
        parameters = KratosMultiphysics.Parameters(parameter_file.read())
    local_parameters = parameters
    model = KratosMultiphysics.Model()
    serialized_model = KratosMultiphysics.StreamSerializer()
    serialized_model.Save("ModelSerialization",model)
    serialized_parameters = KratosMultiphysics.StreamSerializer()
    serialized_parameters.Save("ParametersSerialization",parameters)
    # pickle dataserialized_data
    pickled_model = pickle.dumps(serialized_model, 2) # second argument is the protocol and is NECESSARY
    ↪  (according to pybind11 docs)
    pickled_parameters = pickle.dumps(serialized_parameters, 2)
    print("\n","#"*50," SERIALIZATION COMPLETED ","#"*50,"\n")
    return pickled_model,pickled_parameters

def CreateGidControlOutput( model_part, output_name ):
        from gid_output_process import GiDOutputProcess
        gid_output = GiDOutputProcess(
                model_part,
                output_name,
                KratosMultiphysics.Parameters("""
                    {
                        "result_file_configuration" : {
                            "gidpost_flags": {
                                "GiDPostMode": "GiD_PostBinary",
                                "MultiFileFlag": "SingleFile"
                            },
                            "nodal_results"       : ["VELOCITY","PRESSURE"],
                            "nodal_nonhistorical_results": ["DIVERGENCE","NODAL_H","METRIC_TENSOR_2D"],
                            "nodal_flags_results": []
                        }
                    }
                    """)
                )
        gid_output.ExecuteInitialize()
        gid_output.ExecuteBeforeSolutionLoop()
        gid_output.ExecuteInitializeSolutionStep()
        gid_output.PrintOutput()
        gid_output.ExecuteFinalizeSolutionStep()
        gid_output.ExecuteFinalize()


if __name__ == '__main__':

    '''set the json parameters path'''
    parameter_file_name = "ProblemZeroParameters.json"
    # parameter_file_name = "ProblemZeroParameters.json"
    '''create a serialization of the model and of the project parameters'''
    pickled_model,pickled_parameters = SerializeModelParameters_Task(parameter_file_name)
    '''customize setting parameters of the ML simulation'''
    settings_MC_simulation = KratosMultiphysics.Parameters("""
    {
        "tolerance" : 1,
        "cphi" : 5e-1,
        "batch_size" : 5,
        "convergence_criteria" : "MC_higher_moments_sequential_stopping_rule"
    }
    """)
    '''contruct MonteCarlo class'''
    mc_class = mc.MonteCarlo(settings_MC_simulation)
    '''start MC algorithm'''
    while mc_class.convergence is not True:
        mc_class.InitializeMCPhase()
        mc_class.ScreeningInfoInitializeMCPhase()
        for instance in range (mc_class.difference_number_samples[0]):
            mc_class.AddResults(ExecuteMonteCarloAnalysis(pickled_model,pickled_parameters))
            break
        break
        mc_class.FinalizeMCPhase()
        mc_class.ScreeningInfoFinalizeMCPhase()

    #mc_class.QoI.mean = get_value_from_remote(mc_class.QoI.mean)
    print("\nMC mean = ",mc_class.QoI.mean)
```

References

[1] M. Ainsworth and J. T. Oden. *A posteriori error estimation in finite element analysis*, volume 37. John Wiley & Sons, 2000.

[2] I. Babuška and W. C. Rheinboldt. A-posteriori error estimates for the finite element method. *Int. J. Numer. Meth. Eng.*, 12(10):1597–1615, 1978.

[3] I. Babuška, R. B. Kellogg, and J. Pitkäranta. Direct and inverse error estimates for finite elements with mesh refinements. *Numer. Math.*, 33(4):447–471, 1979.

[4] R. Becker and R. Rannacher. An optimal control approach to a posteriori error estimation in finite element methods. *Acta Numer.*, 10:1–102, 2001.

[5] M. Besier and R. Rannacher. Goal-oriented space-time adaptivity in the finite element Galerkin method for the computation of nonstationary incompressible flow. *International Journal for Numerical Methods in Fluids*, 70(9):1139–1166, 2012.

[6] M. Besier and W. Wollner. On the pressure approximation in nonstationary incompressible flow simulations on dynamically varying spatial meshes. *International Journal for Numerical Methods in Fluids*, 69(6):1045–1064, 2012.

[7] J. Bey. Tetrahedral grid refinement. *Computing*, 55(4):355–378, 1995.

[8] C. Dapogny, C. Dobrzynski, and P. Frey. Three-dimensional adaptive domain remeshing, implicit domain meshing, and applications to free and moving boundary problems. *Journal of computational physics*, 262:358–378, 2014.

[9] T. Dickopf, D. Krause, R. Krause, and M. Potse. Design and analysis of a lightweight parallel adaptive scheme for the solution of the monodomain equation. *SIAM Journal on Scientific Computing*, 36(2):C163–C189, 2014.

[10] W. Dörfler. A convergent adaptive algorithm for Poisson's equation. *SIAM J. Numer. Anal.*, 33(3):1106–1124, 1996.

[11] K. Eriksson, D. Estep, P. Hansbo, and C. Johnson. Introduction to adaptive methods for differential equations. *Acta Numer.*, 4:105–158, 1995.

[12] M. B. Giles and E. Süli. Adjoint methods for PDEs: a posteriori error analysis and postprocessing by duality. *Acta Numer.*, 11:145–236, 2002.

[13] D. Guignard, F. Nobile, and M. Picasso. A posteriori error estimation for elliptic partial differential equations with small uncertainties. *Numerical Methods for Partial Differential Equations*, 32(1):175–212, 2016.

[14] D. Guignard, F. Nobile, and M. Picasso. A posteriori error estimation for the steady Navier–Stokes equations in random domains. *Computer Methods in Applied Mechanics and Engineering*, 313:483 – 511, 2017.

[15] B. Keith. *New ideas in adjoint methods for PDEs: A saddle-point paradigm for finite element analysis and its role in the DPG methodology*. PhD thesis, University of Texas at Austin, Austin, Texas, 2018.

[16] B. Keith, P. Knechtges, N. V. Roberts, S. Elgeti, M. Behr, and L. Demkowicz. An ultraweak DPG method for viscoelastic fluids. *Journal of Non-Newtonian Fluid Mechanics*, 247:107–122, 2017.

[17] P. Ladevèze. *Comparaison de modeles de milieux continus*. PhD thesis, Université P. et M. Curie, Paris, France, 1975.

[18] P. Ladevèze, F. Pled, and L. Chamoin. New bounding techniques for goal-oriented error estimation applied to linear problems. *Int. J. Numer. Meth. Eng.*, 93(13): 1345–1380, 2013.

[19] J. T. Oden and S. Prudhomme. Goal-oriented error estimation and adaptivity for the finite element method. *Comput. Math. Appl.*, 41(5-6):735–756, 2001.

[20] J. T. Oden, I. Babuška, F. Nobile, Y. Feng, and R. Tempone. Theory and methodology for estimation and control of errors due to modeling, approximation, and uncertainty. *Computer Methods in Applied Mechanics and Engineering*, 194(2-5):195–204, 2005.

[21] A. T. Patera and J. Peraire. A general Lagrangian formulation for the computation of a posteriori finite element bounds. In *Error Estimation and Adaptive Discretization Methods in Computational Fluid Dynamics*, pages 159–206. Springer, 2003.

[22] S. Prudhomme and J. Oden. A posteriori error estimation and error control for finite element approximations of the time-dependent Navier–Stokes equations. *Finite elements in analysis and design*, 33(4):247–262, 1999.

[23] S. Prudhomme and J. Oden. Numerical stability and error analysis for the incompressible Navier–Stokes equations. *Communications in numerical methods in engineering*, 18(11):779–787, 2002.

[24] S. Prudhomme and J. T. Oden. On goal-oriented error estimation for elliptic problems: application to the control of pointwise errors. *Comput. Methods Appl. Mech. Eng.*, 176(1-4):313–331, 1999.

[25] S. Prudhomme and J. T. Oden. Computable error estimators and adaptive techniques for fluid flow problems. In *Error estimation and adaptive discretization methods in computational fluid dynamics*, pages 207–268. Springer, 2003.

[26] S. Prudhomme, J. T. Oden, T. Westermann, J. Bass, and M. E. Botkin. Practical methods for a posteriori error estimation in engineering applications. *Int. J. Numer. Meth. Eng.*, 56(8):1193–1224, 2003.

[27] R. Rossi, C. Roig, P. Dadvand, M. Núñez, R. Tosi, R. M. Badia, R. Amela, and B. Keith. Kratosmultiphysics/kratos: Exaqute m12, May 2019. URL `https://doi.org/10.5281/zenodo.3235261`.

[28] R. Verfürth. *A posteriori error estimation techniques for finite element methods*. Oxford University Press, Oxford, 2013.

[29] J. Wu, J. Zhu, J. Szmelter, and O. Zienkiewicz. Error estimation and adaptivity in Navier–Stokes incompressible flows. *Computational mechanics*, 6(4):259–270, 1990.

[30] O. C. Zienkiewicz and J. Z. Zhu. A simple error estimator and adaptive procedure for practical engineerng analysis. *International journal for numerical methods in engineering*, 24(2):337–357, 1987.

[31] O. C. Zienkiewicz and J. Z. Zhu. The superconvergent patch recovery and a posteriori error estimates. Part 2: Error estimates and adaptivity. *International Journal for Numerical Methods in Engineering*, 33(7):1365–1382, 1992.

[32] O. C. Zienkiewicz and J. Z. Zhu. The superconvergent patch recovery and a posteriori error estimates. Part 1: The recovery technique. *International Journal for Numerical Methods in Engineering*, 33(7):1331–1364, 1992.