Horizon 2020

Reduced Order Modelling, Simulation and Optimization of Coupled systems

Benchmark cases

Deliverable number: D5.3

Version 0.1

Funded by the European Union's Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie Grant Agreement No. 765374

Project Acronym: ROMSOC
Project Full Title: Reduced Order Modelling, Simulation and Optimization of Coupled systems
Call: H2020-MSCA-ITN-2017
Topic: Innovative Training Network
Type of Action: European Industrial Doctorates
Grant Number: 765374

Editors:	Andrés Prieto, Peregrina Quintela, ITMATI
Deliverable nature:	Report (R)
Dissemination level:	Public (PU)
Contractual Delivery Date:	30/06/2021
Actual Delivery Date	01/09/2021
Number of pages:	94
Keywords:	Benchmarks, Model hierarchies
Authors:	Marcus Bannenberg, BUW Andreas B´armann, FAU Patricia Barral, ITMATI-USC Jean-David Benamou, INRIA Federico Bianco, Danieli Andres Binder, MathConsult Guillaume Chazareix, INRIA Angelo Ciccazzo, STMicroelectronics Ricardo Conte, Danieli Sören Dittmer, U-HB Daniel Fernández Comesaña, Microflown Technologies Michele Girfoglio, SISSA M. Günther, BUW José Carlos Gutiérrez Pérez, U-HB Lena Hauberg-Lotte, U-HB Michael Hintermüller, WIAS Berlin Wilbert Ijzerman, Philips Onkar Jadhav, MathConsult Tobias Kluth, U-HB Karl Knall, MathTec Alejandro Lengomin, AMIII Peter Maass, U-HB Gianfranco Marconi, Danieli Alexander Martin, FAU Marco Martinolli, MOX, PoliMi Volker Mehrmann, TU Berlin Pier Paolo Monticone, CorWave SA Umberto Morelli, ITMATI Ashwin Nayak, ITMATI Daniel Otero Baguer, U-HB Luc Polverelli, CorWave SA Andrés Prieto, ITMATI-UDC Peregrina Quintela, ITMATI-USC Ronny Ramlau, Industrial Mathematics Institute JKU Conte Riccardo, Danieli Gianluigi Rozza, SISSA Giorgi Rukhaia, INRIA Nirav Shah, SISSA Giovanni Stabile, SISSA Bernadett Stadler, Industrial Mathematics Institute JKU Jonasz Staszek, FAU Christian Vergara, MOX, PoliMi
Peer review:	Andrés Prieto, ITMATI-UDC Peregrina Quintela, ITMATI-USC

Abstract

The description of the selected benchmark cases provide two main elements: (i) A document with a short step-by-step description of the selected benchmark cases to ease the verification, validation and reproduction of its input/output data. (ii) A GitHub repository associated with the selected benchmark cases. Both elements, which include datasets, sources files, implementation requirements and any other supplementary software information, should guarantee that a potential practitioner can run easily and reproduce accurately the provided numerical test cases in relevant real-life engineering and applied science scenarios. These selected benchmarks play an essential role in the development and validation of novel numerical methodologies analysed among the ROMSOC partners, since they ensure the numerical reproducibility of the reported numerical methods and guarantee the sustainability of its computer implementation much beyond the span and the lifetime of the consortium. All the GitHub respositories are available at https://github.com/ROMSOC.

Disclaimer & acknowledgment

This project has received funding from the European Union's Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie Grant Agreement No. 765374.

Contents

List of Acronyms

AO	Adaptive Optics
ELT	Extremely Large Telescope
FSI	Fluid-Structure Interaction
FWHM	Full Width at Half Maximum
ITMATI	Technological Institute of Industrial Mathematics
JKU	Johannes Kepler University Linz
LGS	Laser Guide Star
LTAO	Laser Tomography AO
MVM	Matrix-Vector-Multiplication
NGS	Natural Guide Star
SISSA	Scuola Internazionale Superiore di Studi Avanzati
UDC	University of A Coruña
USC	University of Santiago de Compostela
WFS	Wavefront sensor
XFEM	Extended Finite Element Method

Part I.
Benchmark for high-performance algorithms in adaptive optics control

Bernadett Stadler, Ronny Ramlau

Abstract

The new generation of ground-based extremely large telescopes requires highly efficient algorithms to achieve an excellent image quality in a large field of view. These systems rely on adaptive optics (AO), where one aims to compensate the rapidly changing optical distortions in the atmosphere in real-time. Many of these systems require the reconstruction of the turbulence layers, which is called atmospheric tomography. Mathematically, this problem is ill-posed, due to the small angle of separation. The dimension of the problem depends on the telescope size and has increased in the last years. Altogether, efficient solution methods are of great interest. Within this benchmark case we will use the standard, however, not most efficient method, called Matrix Vector Multiplication, to deal with the problem of atmospheric tomography. As a test example we will consider a reduced problem size, in order to be able to run the benchmark case on an off-the-shelf CPU.

Keywords: Adaptive optics, atmospheric tomography, benchmark, MVM.

Latest release: https://doi.org/10.5281/zenodo.5171804

GitHub repository: https://github.com/ROMSOC/benchmark-adaptive-optics

1.1. Introduction

The new generation of planned earthbound Extremely Large Telescopes (ELT) require highly efficient algorithms to achieve an excellent image quality in a large field of view. These systems rely on a technique called Adaptive Optics (AO) with the task to compensate in real-time the rapidly changing optical distortions caused by atmospheric turbulences. To achieve such a correction, the deformations of optical wavefronts emitted by natural or artificial guide stars are measured via wavefront sensors and, subsequently, corrected using deformable mirrors.

Many of those systems require the reconstruction of the turbulence profile in the atmosphere, which is called atmospheric tomography. Mathematically, this problem is ill-posed, i.e., there is an unstable relation between measurement data and the solution. Hence, regularization techniques must be applied. A common way of dealing with this problem is the Bayesian formulation, where the statistical information regarding the turbulence model and sensor noise can be incorporated. The dimension of the atmospheric tomography problem depends on the number of subapertures of the used wavefront sensors and on the number of degrees of freedom of the correcting mirrors, which are in general higher for bigger telescopes. Moreover, the solution has to be computed in real-time. Altogether, efficient solution methods are of great interest for these kind of problems.

So far, the standard method for atmospheric tomography is the matrix-vector-multiplication (MVM). The computational cost of the MVM scales at $\mathcal{O}(n^2)$, where n is the dimension of the AO system. This dimension is increasing drastically in the next generation of ground-based telescopes, as e.g., the Extremely Large Telescope.

Within this report, the MVM method will serve as benchmark algorithm. We will not consider an ELT-sized problem, but a reduced one. In fact, we will drastically reduce the telescope diameter, the number of subapertures of the wavefront sensors, the number of actuators of the deformable mirror and the number of atmospheric layers. This enables us to still be able to run the benchmark case on an off-the-shelf CPU. This report consists of a description of the input and output data in Section 1.2 and Section 1.4, respectively. Moreover, we provide

a detailed step-by-step description on how to run the benchmark test in Section 1.3.

1.2. Description of input data

The parameters for the test configuration of the benchmark case are summarized in Table 1. Note that we drastically reduce the dimension of the problem compared to the ELT. For a detailed description of the parameter in the table below we refer to [1]. The telescope diameter is only 5 m. We use a reduced number of wavefront sensors and guide stars. If we would define the benchmark case for an ELT-sized problem, the matrices involved in the computation of MVM method would require about 50 GB of memory. Solving such a demanding problem on an off-the-shelf CPU is not feasible. However, to understand the step-by-step procedure to solve the atmospheric tomography this small benchmark case is sufficient.

Description	**Value**
Operating mode	LTAO
Telescope diameter D	5 m
Type of WFS	Shack-Hartmann
Number of WFS	4
Number of layers L	2
Layer heights h_ℓ	[0, 5000]
Layer strength c_n^2	[0.65, 0.35]
Discretization spacing on layer δ_ℓ	[1, 1]
Number of subapertures n_s	10×10
Number of actuators n_a	11×11
Number of photons $n_{photons}$	500
Number of LGS G_{LGS}	3
LGS positions	$(3.75, 0), (3.75/2, 3.75 \cdot \sqrt{3}/2), (-3.75/2, 3.75 \cdot \sqrt{3}/2)$
LGS wavelength λ_{LGS}	589 nm
LGS FWHM	11.4 km
LGS height H	90 km
Laser launch positions (x_i^{LL}, x_j^{LL})	$(16.26, -16.26), (16.26, 16.26), (-16.26, 16.26)$
Number of NGS G_{NGS}	1
NGS positions	$(-5, 0)$
NGS wavelength λ_{NGS}	500 nm
FWHM of non-elongated spot f	1.1
Outer scale L_0	25 m
Fine-tuning parameter α_η	0.4
Fried parameter r_0	0.129

Table 1: Test configuration of benchmark case.

Based on these input parameters the matrices involved in the computation of the MVM algorithm can be

calculated. These matrices are

- the Shack Hartmann operator Γ,
- the bilinear interpolation operator P,
- the turbulence covariance matrix C_ϕ^{-1},
- the noise covariance matrix C_η^{-1},
- and the mirror fitting operator F.

All these matrices are stored in the directory `benchmark` in the GitHub repository. For details on how to compute them we refer to the software based representation of the benchmark cases in [1]. Note that all matrices except the turbulence covariance matrix C_ϕ have a sparse structure and are stored as triplet format $(row, col, value)$ in the respective `txt` file. For the dense matrix C_ϕ we store the entire matrix. Besides these matrices the vector of sensor measurements s is required as input for the MVM method. This vector is also contained within the `benchmark` directory.

1.3. Step-by-step procedure

The GitHub repository contains the shell script `run_test.sh` (see Listing 1), which compiles and runs the benchmark case. In particular, it executes the `CMakeLists.txt` (see Listing 2) in the `benchmark` directory, compiles the source files and executes the created file. The executable file reads the input matrices from the directory `benchmark`, performs the MVM algorithm and stores the output again to the `benchmark` directory. In order to be able to successfully run the shell script a C++ compiler, which supports the C++ 14 standard, and a CMake version greater than 3.6 is required. The only library needed for the benchmark case is `Eigen`, which is included into the repository.

```
echo "Run MVM benchmark test"
cd source
cmake .
make release
benchmark
```

Code Listing 1: run_test.txt

```
cmake_minimum_required(VERSION 3.6)
project(benchmarks)
set(CMAKE_CXX_STANDARD 14)
include_directories(Eigen)
add_executable(benchmarks main.cpp)
```

Code Listing 2: CMakeLists.txt

The MVM algorithm programmed in the `main.cpp` source file starts with reading the input matrices and the measurement vector from the `benchmark` directory. Afterwards, the reconstruction matrix R and the right-hand side b are computed via matrix-matrix and matrix-vector multiplications

$$R = (\Gamma P)^T C_\eta^{-1} (\Gamma P) + C_\phi^{-1}$$
$$b = (\Gamma P)^T C_\eta^{-1} s.$$

Note that all matrices involved in the computation above have a sparse representation except C_ϕ.

Then we can set-up the problem of atmospheric tomography and reconstruct the turbulence layers ϕ from the sensor measurements s by solving the system

$$R\phi = b.$$

For more details on the atmospheric tomography problem we refer again to [1].

The inversion is performed in two steps. First, we calculate the Householder rank-revealing QR decomposition of the matrix R via `colPivHouseholderQr` provided by `Eigen` library. Afterwards, we use the `Eigen` function `solve` to obtain the desired solution ϕ. Finally, the turbulence layers are fitted to actuator commands by applying the matrix F. The output vector is stored in the file `out.txt` in the `benchmark` directory.

This benchmark case, which has a significant lower dimension than the ELT, already shows the drawbacks of the matrix-based MVM algorithm. The dense input matrix C_ϕ for this toy example requires more than 300 MB of memory. Hence, using matrix-free methods, such as the Finite Element Wavelet Hybrid Algorithm (FEWHA), provide significant advantages here. For details about FEWHA and related topics we refer to [2, 3, 4, 5].

1.4. Description of output data

The output of the MVM algorithm are the actuator commands, with which the deformable mirror can be adjusted such that atmospheric distortions are corrected. For our specific benchmark case the resulting DM commands are a vector of size 100. This vector is stored in the file `out.txt`, which is contained in the directory `benchmark`.

Part II.
A benchmark for coupled models for acoustic propagation through multilayer systems for particle-velocity sensors

Ashwin Nayak, Andrés Prieto, Daniel Fernández

Abstract

Some benchmark cases are described in detail focused on the acoustic scattering simulation of a rigid exterior domain enclosed in a porous layer. Details of the mathematical model used are highlighted alongside numerical procedures implemented in obtaining an approximate solution. An elaborate end-to-end strategy using open-source software tools to compute the solution is also provided. The provided benchmarks involves not only academic test involving simplified geometries (such as spherical computational domains) but also realistic CAD geometries based on engineering PU probes designs.

Keywords: Scattering, aeroacoustics, porous materials.

Latest release: https://doi.org/10.5281/zenodo.5171815

GitHub repository: https://github.com/ROMSOC/benchmarks-acoustic-propagation

2.1. Introduction

This benchmark repository contains scripts for generating the relevant meshes and computing the scattering of a plane wave by the Microflown PU Regular Probe and other simple geometries such as a three-dimensional spheres. More precisely, the GitHub repository includes all the data and Python scripts to run from scratch the following three coupled problems:

- A three-dimensional coupled fluid-PML acoustic scattering problem with an spherical obstacle.
- A three-dimensional coupled fluid-porous-PML acoustic scattering problem with an spherical obstacle coated with a spherical porous layer.
- A three-dimensional coupled fluid-porous-PML acoustic scattering problem, where the obstacle is a realistic design of a PU probe coated with a cylindrical porous layer.

These three benchmarks share a common mathematical modelling and an analogous variational formulation. All of them have been discretized with the same kind of finite element method. In what follows, the mathematical formulation and a detailed description of its discretization is provided.

2.2. Mathematical Formulation

The implementation considered is one of the benchmark stages of the project i.e. to solve for the acoustic scattering effect of a rigid object represented by the external domain Ω_{S}, enclosed entirely by a porous layer Ω_{P}. The setup is placed in an acoustic field represented by the unbounded domain Ω_{F}, as shown in the schematic Fig.2.1a. To be more generic with possible configurations - a fluid-filled gap is considered between the structure and the porous enclosure. An acoustic wave of a certain kind (plane wave, spherical etc.) is assumed to be incident on the setup and a model is sought to compute the scattering of the incident wave due to the object.

The problem can be mathematically formulated in various physically-relevant variables e.g scalar fields like pressure, displacement potential or velocity potential; or vector fields like displacement or velocity, the choice often being the vector fields for coupled systems [6]. In this particular implementation, the acoustic oscillations are chosen to be represented by the displacement vector field.

A series of assumptions are considered to arrive at a feasible mathematical model for the problem. The acoustic fluid is assumed to be homogeneous, non-viscous, compressible, isotropic and isentropic. Also, the porous layer is considered to be made of homogeneous, isotropic and isothermal material. The acoustic fields are

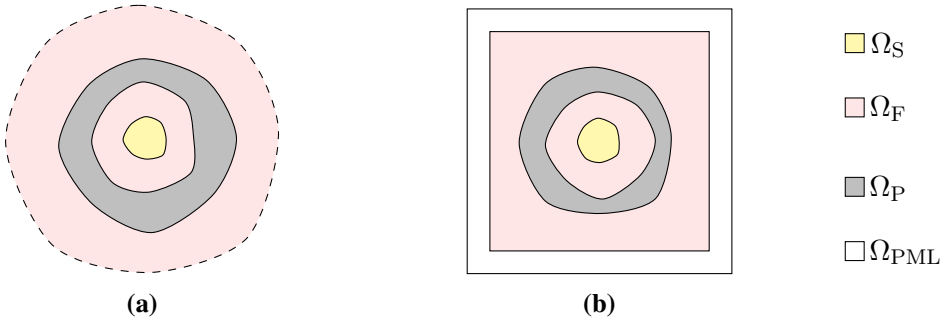


Figure 2.1: Schematic of the original problem configuration on unbounded domain (a), and the model configuration with perfectly matched layers (b).

assumed to be time-harmonic. The problem configuration is also posed in an unbounded domain which ensures a complete dissipation of all outgoing waves. Pragmatically, this is mimicked by a model configuration with a finite truncation of the domain and an artificial boundary enclosing it with absorbing properties, known in literature as the perfectly matched layers (PML) technique [7, 8]. It is represented by the Cartesian box Ω_{PML} in Fig.2.1b.

The mathematical formulation for the coupled problem may be surmised as the following system of equations: for a particular frequency, ω,

$$-\nabla(\rho_{\mathrm{F}} c_{\mathrm{F}}^2 \operatorname{div} \boldsymbol{u}_{\mathrm{F}}) - \rho_{\mathrm{F}}\omega^2 \boldsymbol{u}_{\mathrm{F}} = \boldsymbol{f}_{\mathrm{F}} \quad \text{in } \Omega_{\mathrm{F}}, \tag{2.1}$$

$$-\nabla(K_{\mathrm{P}}(\omega) \operatorname{div} \boldsymbol{u}_{\mathrm{P}}) - \rho_{\mathrm{P}}(\omega)\omega^2 \boldsymbol{u}_{\mathrm{P}} = \boldsymbol{f}_{\mathrm{P}} \quad \text{in } \Omega_{\mathrm{P}},$$

$$-\operatorname{div}(\rho_{\mathrm{F}} c_{\mathrm{F}}^2 \widetilde{\mathsf{C}}(\nabla \boldsymbol{u}_{\mathrm{PML}})) - \rho_{\mathrm{F}}\omega^2 \widetilde{\boldsymbol{M}} \boldsymbol{u}_{\mathrm{PML}} = \boldsymbol{f}_{\mathrm{PML}} \quad \text{in } \Omega_{\mathrm{PML}}, \tag{2.2}$$

$$\boldsymbol{u}_{\mathrm{F}} \cdot \boldsymbol{n} = g \quad \text{on } \Gamma_{\mathrm{S}}, \tag{2.3}$$

$$\boldsymbol{u}_{\mathrm{F}} \cdot \boldsymbol{n} - \boldsymbol{u}_{\mathrm{P}} \cdot \boldsymbol{n} = 0 \quad \text{on } \Gamma_{\mathrm{C}}, \tag{2.4}$$

$$\rho_{\mathrm{F}} c_{\mathrm{F}}^2 \operatorname{div} \boldsymbol{u}_{\mathrm{F}} - K_{\mathrm{P}}(\omega) \operatorname{div} \boldsymbol{u}_{\mathrm{P}} = 0 \quad \text{on } \Gamma_{\mathrm{C}},$$

$$\boldsymbol{u}_{\mathrm{F}} \cdot \boldsymbol{n} - \boldsymbol{u}_{\mathrm{PML}} \cdot \boldsymbol{n} = 0 \quad \text{on } \Gamma_{\mathrm{PML}},$$

$$\operatorname{div} \boldsymbol{u}_{\mathrm{F}} - \operatorname{div} \boldsymbol{u}_{\mathrm{P}} = 0 \quad \text{on } \Gamma_{\mathrm{PML}}. \tag{2.5}$$

Here, $\boldsymbol{u}_{\mathrm{F}}$, $\boldsymbol{u}_{\mathrm{P}}$ and $\boldsymbol{u}_{\mathrm{PML}}$ are the displacement vector fields in the fluid, porous and PML domains respectively. Γ_{S}, Γ_{C} and Γ_{PML} represent the boundaries making up the interfaces between structure-fluid, fluid-porous and fluid-PML domains with outward facing normals, $\boldsymbol{n}$. The model includes material properties like fluid mass density ρ_{F}, sound speed in the fluid c_{F}, the dynamic porous mass density ρ_{P} and the dynamic porous bulk modulus K_{P}. Equations (2.1)-(2.2) represent the Helmholtz-like equations in each of the domains. Equation (2.3) is a boundary condition at the object boundary and (2.4)-(2.5) represent the pressure and displacement continuity conditions on the interfaces. The source-terms $\boldsymbol{f}_{\mathrm{F}}$, $\boldsymbol{f}_{\mathrm{P}}$, $\boldsymbol{f}_{\mathrm{PML}}$ and function g appear according to initial sources of disturbances and are explained later in this document.

The porous material properties are determined either through experiments conducted *apriori* or through suitable models. A wide range of porous material models provide the material response along a range of frequencies e.g the Zwikker-Kosten model, Miki model, Johnson-Champoux-Allard-Lafarge Model, the Johnson-Champoux-Allard-Pride-Lafarge model among others [8, 9]. The fairly detailed six-parameter Johnson-Champoux-Allard-Lafarge (JCAL) model is chosen in the current article to obtain the dynamic porous mass density and bulk modulus, given by equations,

$$\rho_{\mathrm{P}}(\omega) = \frac{\rho_{\mathrm{F}}}{\phi}\alpha_\infty \left(1 - \mathrm{i}\frac{\sigma\phi}{\omega\rho_{\mathrm{F}}\alpha_\infty}\sqrt{1 + \mathrm{i}\frac{4\alpha_\infty^2\eta\rho_{\mathrm{F}}\omega}{\sigma^2\Lambda^2\phi^2}}\right),$$

$$K_{\mathrm{P}}(\omega) = \frac{\gamma P_{\mathrm{F}}/\phi}{\gamma - (\gamma-1)\left(1 - \mathrm{i}\dfrac{\eta\phi}{\rho_{\mathrm{F}} k_0' \omega \mathrm{Pr}}\sqrt{1 + \mathrm{i}\dfrac{4{k_0'}^2 \rho_{\mathrm{F}} \omega \mathrm{Pr}}{\eta \Lambda'^2 \phi^2}}\right)^{-1}}.$$

The JCAL model is reliable for porous materials with arbitrarily shaped pores. The parameters in the model: porosity ϕ, flow resistivity σ, tortuosity α_∞, viscous characteristic length Λ, thermal characteristic length Λ' and static thermal permeability ${k_0}'$; effectively capture the macroscopic thermal, viscous and inertial characteristics of the porous material. The model also requires the fluid state properties like density ρ_{F}, specific heat ratio γ, Prandtl Number Pr, and equilibrium fluid pressure P_{F}.

The Helmholtz-like PML governing Equation (2.2) ensures absorption of outgoing waves. This is achieved by a complex stretching of spatial variables[10] by the fourth-order tensor $\widetilde{\mathsf{C}}$ and the second-order tensor $\widetilde{\boldsymbol{M}}$ given by,

$$\widetilde{\mathsf{C}}(\nabla \boldsymbol{w}) = \left(\sum_{j=1}^{3} \frac{1}{\gamma_j}\frac{\partial w_j}{\partial x_j}\right)\mathsf{I}$$

$$\text{and} \quad \widetilde{\boldsymbol{M}} = \sum_{j=1}^{3} \gamma_j \boldsymbol{e}_j \otimes \boldsymbol{e}_j,$$

where, I is the fourth-order identity tensor and $\boldsymbol{e}_j$'s are the unit vectors along the spatial directions. The optimally-tuned functions provided by Bermudez et al.[11] are chosen among the various choices for the complex stretching functions γ_j's, giving,

$$\gamma_j(x_j) = \begin{cases} 1 & |x_j| \le L_j, \\ 1 + \mathrm{i}\dfrac{c_{\mathrm{F}}}{\omega(L_j^\infty - |x_j|)} & L_j \le |x_j| \le L_j^\infty. \end{cases}$$

Here, L_j and L_j^∞ are respectively the lengths of the Cartesian box of the truncated fluid domain and the PML domain, along the direction x_j from the origin. The definition of γ_j as a piece-wise function ensures the absorption of waves only along the outward direction of propagation. Consequently, the tensors $\widetilde{\mathsf{C}}$ and $\widetilde{\boldsymbol{M}}$ are piece-wise and needs to be considered with care during the implementation.

The model described in Equations (2.1)-(2.5) explain the propagation of a generic acoustic field and needs adaptation for our initial problem of computing the acoustic scattering of an incident wave. The total normal displacement at the object boundary are zero ($g = 0$) since the structure is assumed rigid. The principle of superposition may then be utilized to split the total field into the incident field and scattered field components. The equations are then rewritten in terms of the scattered part of the field to obtain right-hand-sides, some of which are non-null.

Considering that the displacement vector fields, $\boldsymbol{u}_{\mathrm{F}}$, $\boldsymbol{u}_{\mathrm{P}}$ and $\boldsymbol{u}_{\mathrm{PML}}$, are defined in exclusive domains albeit with different smoothing requirements, it may be unified to be a member of a functional space $\mathbf{V}$ introduced as,

$$\mathbf{V} = \Big\{\boldsymbol{v} \in [\mathrm{L}^2(\Omega)]^3 : \boldsymbol{v}|_{\Omega_{\mathrm{F}}} \in \mathbf{H}(\mathrm{div}, \Omega_{\mathrm{F}}),\ \boldsymbol{v}|_{\Omega_{\mathrm{P}}} \in \mathbf{H}(\mathrm{div}, \Omega_{\mathrm{P}}),\ \widetilde{\boldsymbol{M}}\boldsymbol{v}|_{\Omega_{\mathrm{PML}}} \in [\mathrm{L}^2(\Omega_{\mathrm{PML}})]^3,$$
$$\sum_{j=1}^{3} \frac{1}{\gamma_j}\frac{\partial v_j}{\partial x_j}\Big|_{\boldsymbol{v}\in\Omega_{\mathrm{PML}}} \in \mathrm{L}^2(\Omega_{\mathrm{PML}}), \boldsymbol{v}\cdot\boldsymbol{n} = 0 \text{ on } \Gamma_\infty\Big\},$$

which also ensures the necessary continuity and differentiable properties at the interfaces. The variational form can then be deduced from Equations (2.1)-(2.2) by multiplying a test function $\boldsymbol{v} \in \mathbf{V}$ and utilizing the Green's

theorem : Find $\boldsymbol{u} \in \mathbf{V}$ such that,

$$\begin{aligned}
&\int_{\Omega_\mathrm{F}} \rho_\mathrm{F} c^2 (\operatorname{div} \boldsymbol{u})(\operatorname{div} \boldsymbol{v})\,\mathrm{d}V - \int_{\Omega_\mathrm{F}} \rho_\mathrm{F} \omega^2 \boldsymbol{u}\cdot\boldsymbol{v}\,\mathrm{d}V \\
&+ \int_{\Omega_\mathrm{P}} \mathrm{K}_\mathrm{P}(\omega)(\operatorname{div} \boldsymbol{u})(\operatorname{div} \boldsymbol{v})\,\mathrm{d}V - \int_{\Omega_\mathrm{P}} \rho_\mathrm{P}(\omega)\omega^2 \boldsymbol{u}\cdot\boldsymbol{v}\,\mathrm{d}V \\
&+ \int_{\Omega_\mathrm{PML}} \rho_\mathrm{F} c^2 \widetilde{\mathsf{C}}(\nabla \boldsymbol{u}) : \nabla \boldsymbol{v}\,\mathrm{d}V - \int_{\Omega_\mathrm{PML}} \rho_\mathrm{F}\omega^2 \widetilde{\boldsymbol{M}}\boldsymbol{u}\cdot\boldsymbol{v}\,\mathrm{d}V = \int_{\Omega_\mathrm{F}} \boldsymbol{f}_\mathrm{F}\cdot\boldsymbol{v}\,\mathrm{d}V + \int_{\Omega_\mathrm{P}} \boldsymbol{f}_\mathrm{P}\cdot\boldsymbol{v}\,\mathrm{d}V \\
&\qquad\qquad + \int_{\Omega_\mathrm{PML}} \boldsymbol{f}_\mathrm{PML}\cdot\boldsymbol{v}\,\mathrm{d}V
\end{aligned} \tag{2.6}$$

holds for all $\boldsymbol{v} \in \mathbf{V}$ and also, $\boldsymbol{v} = 0$ at Γ_S. The Equation (2.6) maybe more conveniently expressed in the general form of a linear variational problem with $\mathcal{A}$ and $\mathcal{L}$ being the sesquilinear and linear functionals as,

$$\mathcal{A}(\boldsymbol{u}, \boldsymbol{v}) = \mathcal{L}(\boldsymbol{v}). \tag{2.7}$$

A practical implementation of this model would require the approximation of an infinite dimensional functional space $\mathbf{V}$, with a discrete n-dimensional space $\mathbf{V}_h$ with a finite set of basis functions $\boldsymbol{\psi}_h$, $h = 1, 2, ..., n$. This reforms Equation (2.7) as,

$$\sum_{r=1}^{n} \mathcal{A}(\psi_r, \psi_s)\mu_r = \mathcal{L}(\psi_s) \qquad \text{for } s = 1, 2..., n;$$

with the μ_r's as coefficients of the basis functions. A solution may then be obtained by solving this system of equations. The following sections details the implementation of this model along with a specific example of acoustic transmission across a porous layer around a vibrating sphere.

2.3. Implementation

The implementation follows the requirements of the model and may be divided into three main stages viz., mesh generation, solving equations and visualizing solutions. The different stages of the implementation and the overall workflow is illustrated in Fig.2.2. The mesh generation stage requires the user inputs on geometrical configuration of the setup. This includes the exact dimensions of the structure, porous layer, fluid domain and PML. Considering that the variational form includes integrals which differ in sub-domains, it is necessary to mark the mesh cells according to region requiring conformality of the mesh with the geometry of sub-domains. Furthermore, user inputs may be needed to suggest local refining of the mesh in a particular region or surface to capture the geometry accurately. The generated mesh also needs to be adapted to the file format compatible with the solver. The solver imports the mesh data and categorizes cells according the sub-domain regions. It is responsible for implementing the finite-element method - defining the discrete functional space with chosen basis functions and assembling the system of equations before solving them. The solution obtained may also include processing for analysis before being saved in a memory-efficient storage format. Finally, the visualization tool reads the simulated solution from the disk to provide graphical representations aiding the user in deriving information and performing analysis. The following sections explain the usage of each of the stages and the related tools in detail. The software tools used in the implementation of the project require a minimal UNIX system with at least 1GB of memory and about 500MB of disk space (swap) for execution. It is recommended to have some higher configuration would ease the workflow and be capable of handling problems of larger order.

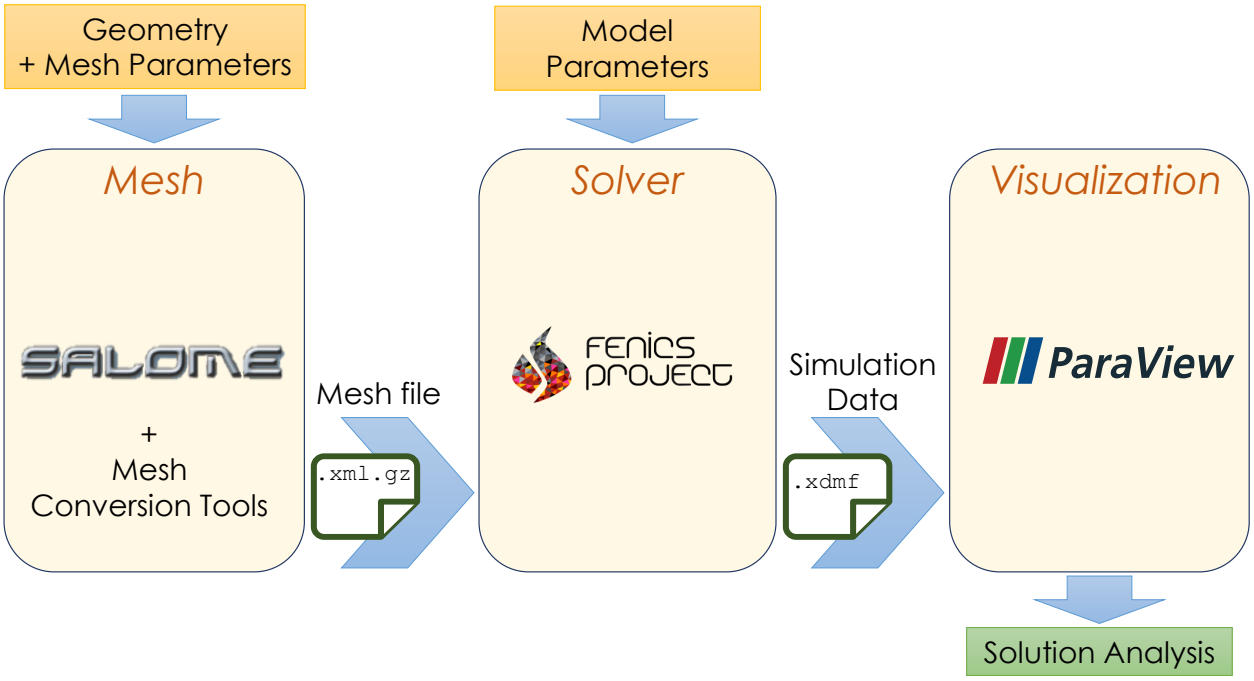


Figure 2.2: Workflow representing implementation stages and their interfaces.

2.3.1. Geometry and Meshing

The digital representation of the setup is first done by modeling the geometry and then discretizing it to form a mesh. While several tools and techniques are available for this, the open-source modules offered by SALOME are used in this article, which provides capabilities for interfacing with various numerical simulation tools. It has a flexible cross-platform architecture made of reusable components allowing for customized integration and handling of complex geometrical objects. It allows for creation of geometry and meshes using either (or both) the graphical user interface (GUI) and a text user interface (TUI). The following sections explains the usage of creating geometries and meshes using the TUI, a powerful Python-based scripting interface. The same may also be achieved either in part or entirety by using the GUI which allows for exporting the equivalent state in a TUI script.

It allows for creating basic objects and primitives in 1D, 2D, 3D; perform boolean operations like fuse, common, cut and section operations; execute extrusion, rotation and other linear operations; create higher order topological objects like solids and compounds grouped from primitives; and implement an advanced partition or gluing between geometrical structures, among others (see the SALOME documentation [12] for further details).

The GitHub repository contains three different scripts to build the meshes (in folder `source/00_meshes`), which corresponds to the three different benchmarks included in the repository:

```
$ salome -t microflown_pu_probe.py  # Mesh for coupled PU probe
$ salome -t sphere_PML.py  # Mesh for coupled fluid-PML
$ salome -t sphere_porous_PML.py  # Mesh for coupled fluid-porous-PML
```

2.3.2. Solver

The solver of choice is FEniCS, a widely used open-source finite-element library for solving partial differential equations (PDEs). It offers a rich interface with data-structures and optimized algorithms for finite-element code which makes it easy to write PDEs. The library is optimized and parallel by design and it is easy to deploy and scale the code into high-performance computing clusters. With its Python and C++ interfaces, FEniCS offers powerful capabilities to integrate into workflows.

The FEniCS library offers a number of component modules and the interfacing is done mainly through its DOLFIN and UFL modules. DOLFIN is the highly optimized computational back-end written in C++ responsible for finite-element machinery. It provides abstract data-structures similar to mathematical terms such as mesh, finite element, function spaces and functions. It also includes compute-intensive algorithms such as finite-element assembly and mesh refinement, and, interfaces to various linear algebra solvers and libraries like PETSc. UFL, on the other hand, provides an abstract mathematical language to express variational problems which are interpreted automatically and connected to DOLFIN classes.

The powerful feature of the solver is its ability to interpret the variational form in an easily readable UFL framework. The Python module also allows for finer control through a detailed interface to the underlying C++ code enabling sub-classing and base class overloading. Among others, it provides an `Expression` class which can be used for user-defined expressions specified by C++ code and compiled during execution by a just-in-time (JIT) compiler for efficiency.

A detailed documentation along with numerous examples are offered by Langtangen et al.[13] and at the official FEniCS documentation webpage[14]. It is to be noted that the library is limited by its inability to handle complex numbers and needs additional care to ensure that the real and imaginary parts of the equations and function spaces are represented separately.

The Python scripts related with the three benchmarks in the GitHub repository are structured in two folders:

- Folder `source/02_scattering_sphere/`: it contains the planewave scattering benchmarks involving spherical geometries. This directory contains scripts to develop and analyze FEM method to solve for scattering by a sphere of an incident plane/radial wave. Since the exact solutions can be obtained for this scenario it is ideal to perform analysis of the FEM method and also validate the solver. This folder contains the first two benchmarks regarding spherical geometries:
  - Subfolder `source/02_scattering_sphere/planewave_scattering/`: it contains the planewave scattering benchmark involving only rigid sphere surrounded by fluid. In this benchmark, an incident plane wave is impinged on a rigid sphere, and these scripts computes the scattered field utilizing a classical quadratic PML absorption boundary condition. The variational form is written in terms of the scattered field. Exact solution is known and the errors can be computed to validate the solver. The absorption coeffients for the quadratic PML functions are estimated using the monopole test case and used here. Exact solution is provided by C++ scripts provided in the exact folder which are compiled during runtime by the FEniCS provided JIT compiler. FE Errors can be computed and convergence behaviour can be investigated using these scripts. To run the numerical simulation, just type the Python script:

    ```
     # Solver for the coupled fluid-PML
    $ python3 scattering_sphere_classical_PML.py
    ```

  - Subolder `source/02_scattering_sphere/radialwave_scattering_porous_coupling/`: it contains the radial wave scattering by a rigid sphere with a spherical porous layer. A radial wave is incident on a rigid sphere which is surrounded by a spherical porous layer. The solver computes the resulting field using the Johnson-Champoux-Allard-Lafarge (JCAL) model for a few typical parameter values, and compares them against the exact solution. The exact solution requires an inverse method to compute radial wave amplitudes within the subdomains (see a

detailed description of this benchmark in [15, Chapter 2]). To run the numerical simulation, the following Python script must be used:

```
# Solver for thex coupled fluid-porous-PML
$ python3 scattering_sphere_porous.py
```

- Folder source/03_scattering_pu_probe/: it contains the planewave scattering benchmark involving a Microflown PU Probe. In this benchmark, an incident plane wave is scattered by the Microflown PU Probe which may contain a steel mesh (treated as acoustically equivalent to micro-perforated panels (MPP)) and a porous windscreens. Dah-You-Maa's model is used for the MPP and JCAL model for the porous layer. Optimal PML model is used to replicate the non-reflecting boundaries. The ensemble is coupled with interface conditions and solved using the displacement formulation of the Helmholtz-like equations using the FEM method. Notice that rhe parameters for these models needs to be obtained from experimental data. To run the solver, a Python script should be executed:

```
# Solver for the PU probe
$ python3 scattering.py
```

2.3.3. Post-processing and Visualization

Visualization of generated simulation data is critical in understanding the physical process. Implementing and representing this data in a simple and effective manner is extremely useful for deriving information, presenting results, and also in testing and debugging. The post-processing operations on the solution is dependent on the study undertaken by the user. In this specific use case, some routine cases of analysis include validation of the solver for a test case, measure of a field at a particular point in space and directivity patterns of fields around the object amongst many others. The implementation of these could either be included in the solver phase or during the visualization phase. Within the solver phase, these could just be operations on the solution data done using Python and plotted using some common user preferred graphing libraries like Matplotlib[16]. The approach quickly gets overwhelming while dealing with 3D datasets and it is useful to use a dedicated visualization tool like ParaView. It is a widely used open-source visualization tool for plotting and viewing solutions and graphs. It offers a powerful and an intuitive 3D visualization interface allowing for heavy in-situ customization and processing of simulation data. Furthermore, it also provides a Python scripting interface to automate visualization for batch processing.

ParaView uses a three-stage procedure for visualization of data: reading, filtering and rendering, all done using the user interface. The simulation data from the solver is read into memory through many supported file formats. The dataset being typically large, the XDMF (eXtensible Data Model and Format) file format is used for storage, which is able to manage extremely large datasets and is scalable for parallel systems. Filters provide the ability to extract or analyze this data into information. There are a wide range of filters available for analysis and visualization including plotting graphs, contours, surface plots, vector field plots etc. In addition, it is also possible to define user-defined filters to perform customized operations. The rendering stage deals with generating images or interactive plots from the filtered information. ParaView provides a user guide[17] and many tutorials[18] highlighting the usage and relevance of each of the stages along with the available functionalities to fully exploit its potential.

The three benchmarks provided in this GitHub repository save the output results using XDMF files, which are directly compatible and can be easily visualized using ParaView. The user interface is very intuitive and once the file is opened is allows the user to select the solution fields to import from the dataset into memory. Once the datasets are imported, it renders the volumetric data on the viewer. The toolbar offers some commonly used filters and are also accessible from the menu options. Initially, the dataset is bifurcated into truncated fluid

domain and the PML. This can be achieved using the `ExtractCellsByRegion` filter used with its 'box' configuration scaled appropriately to omit the cells in the PML. To obtain cross-sections of the volumetric data, as shown in Fig. 2.3, the filter `Clip` or `Slice` is used and the specifications of the cutting plane is provided. It is also possible to compute from the saved fields on the interface directly by using the filter `Calculator`.

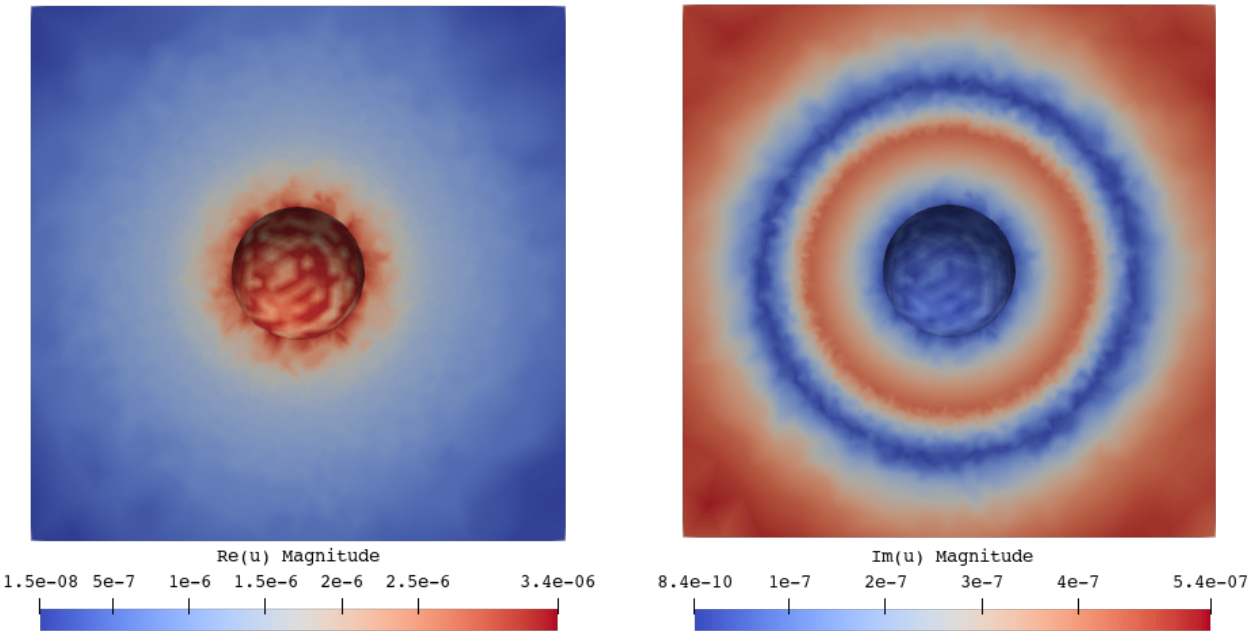


Figure 2.3: Cross-sectional contour views of the magnitude of the real part (left) and the magnitude of the imaginary part (right) of the computed displacement field in the fluid-porous-PML benchmark.

This filter allows the user to define an expression using the fields in memory and compute a derived field. It is useful in computing the total displacement field putting together the real and imaginary portions.

2.4. Conclusion

A detailed step-by-step procedure has been described to run three different benchmarks involving coupling problems in acoustic. The simulation methodology is entirely based on open-source software. The scripting interface interlace with the graphical interface for mesh generation provided by SALOME is useful in meshing complex geometry and automating the procedure either entirely or partly. The solver stage with the intricate finite element machinery is handled by the FEniCS library offering ease-of-use to the user while focusing their attention towards the development and prototyping of the model. The visualization tool, ParaVIEW is feature-rich providing access to many post-processing algorithms along with an intuitive interface. The entire toolchain can also be controlled on Python scripts allowing for easier development and future customization.

Part III.
Acceleration of Sinkhorn Algorithm using ϵ scaling with applications to the Reflector Problem

Jean-David Benamou, Chazareix Guillaume, Wilbert IJzerman, Giorgi Rukhaia

Abstract

FreeForm Optics is the branch of Optics concerned with the design of non-conventional asymmetric refractive and reflective optical elements or systems of such elements. This research is important to improve the energy efficiency of lighting devices and reduce light pollution (for example of street lighting). A classic application of FreeForm Optics (amongst many) is the irradiance tailoring problem: design an optical system transferring a given light source emittance (e.g a car headlight bulb) to a prescribed irregular target irradiance (e.g. the angular far-field distribution of projected light). At the industrial level, FreeForm Optics design has remained so far largely heuristic.

On the academic side, two classes (collimated or point source illuminance) of idealized tailoring irradiance problems can be exactly modeled and solved using Optimal Transport theory. Optimal Transport defines a unique map or a coupling between prescribed distributions representing given illuminance and irradiance. This map can then be used to construct the optical element shape. Recent advances in Optimal Transport numerical solvers allow tackling systems described by millions of degrees of freedom. This offers a sound mathematical and numerical background to FreeForm Optics.

There are several different approaches for finding numerical solutions of Optimal Trasport problems, varying in efficiency, accuracy, and complexity. This work concentrates on the Sinkhorn algorithm. The main advantages of the Sinkhorn algorithm are its simple structure of implementation, involving only simple basic linear algebra operations, and it's fastness both from the mathematical foundation and from a wide selection of fast linear algebra libraries. Also, this algorithm can be drastically speeded up using model hierarchy techniques such as discretization and regularization parameter scaling.

Keywords: Reflector Problem, FreeForm Optics, Optimal Transport, Entropic Regularization, Sinkhorn Algorithm.

Latest release: https://doi.org/10.5281/zenodo.5171811

GitHub repository: https://github.com/ROMSOC/benchmarks-PS-reflector

3.1. Introduction

A light source, also called "illuminance", is sufficiently small compared to the reflecting surface so that it can be regarded as a point in space. It can therefore be modelled as a probability distribution on the sphere, it will be denoted μ in this paper. The light hits a perfect mirror and we are also given a desired target light distribution, the "illumination" in the far field. From the far field the reflecting surface can be regarded as a point and the illumination again modelled as a probability distribution, denoted ν, on the sphere. Total light conservation is assumed. The reflector problem is to determine the shape of the mirror which produces the specular reflection from the source to the target distribution. This can be interpreted as the inverse problem of generating some illumination given an illuminance and a reflector (see figure 3.1).

3.1.1. Optimal Transport model

This problem has an elegant mathematical modelization and solution based on the optimal transportation (OT) theory due to [19] and [20]. We briefly recall the main result as presented in [20]. In its Kantorovich primal and dual form (see [21]) :

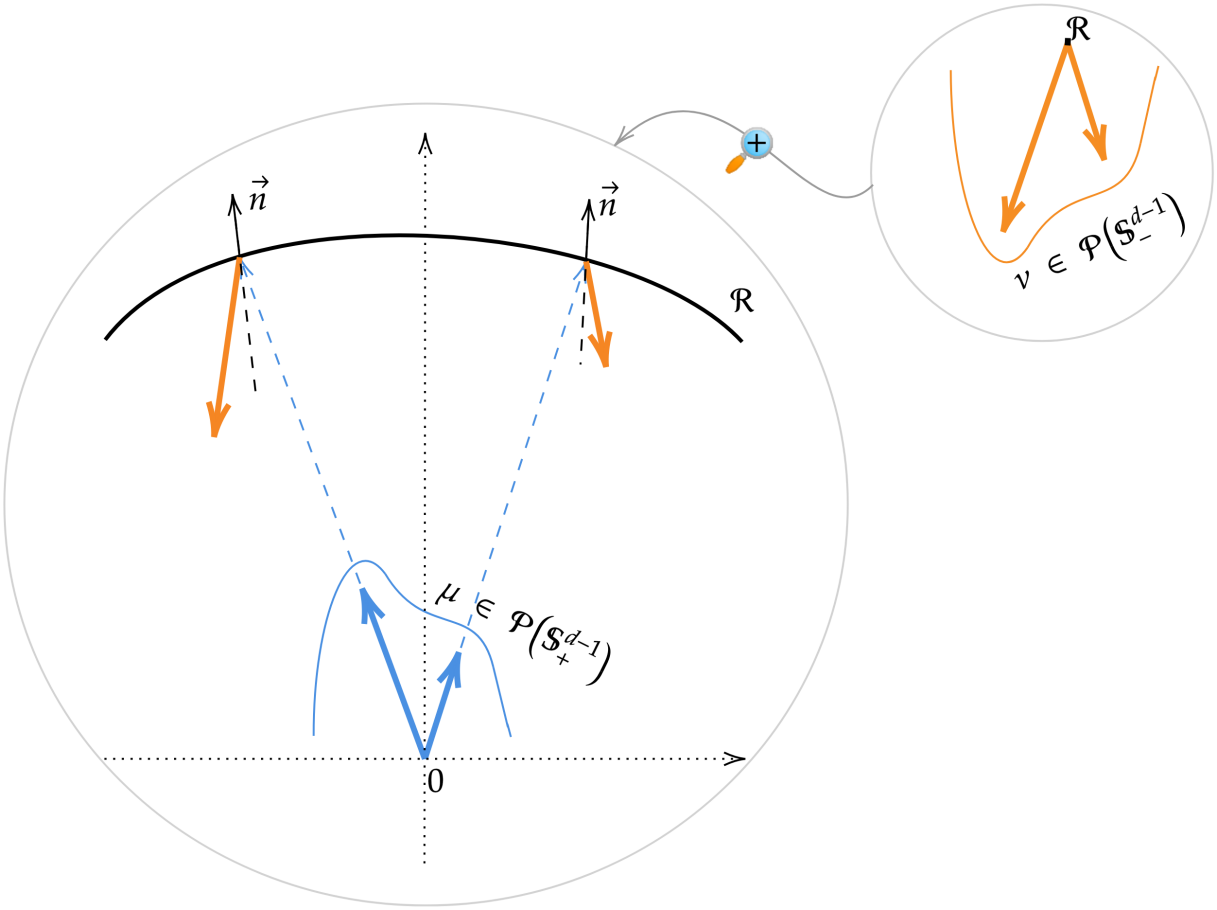


Figure 3.1: Reflector problem from Point source O to Far Field.

Theorem 1 (Kantorovich duality)**.** *Given two compact manifold X and Y endowed with a continuous, bounded from below cost function $c : X \times X \to \mathbf{R}$ and two borel probability measures $(\mu, \nu) \in \mathcal{P}(X) \times \mathcal{P}(Y)$. Then, Kantorovich problem in primal and dual forms (1) has solutions.*

$$OT(\mu,\nu) := \min_{\gamma \in \Pi(\mu,\nu)} \langle c, \gamma \rangle_{X\times Y} = \max_{f,g \in C} \langle f, \mu \rangle_X + \langle g, \nu \rangle_Y$$

with respectively primal :

$$\Pi(\mu,\nu) := \{\gamma \in \mathcal{P}(X \times Y), \langle 1_X, \gamma \rangle_Y = \nu \ \langle 1_Y, \gamma \rangle_X = \mu\},$$

and dual :

$$C = \{(f,g) \in \mathcal{C}(X) \times \mathcal{C}(Y),\ f \oplus g \leq c\},$$

constraints sets

The notation $\langle f, \alpha \rangle_\Omega$ stands for the duality product $\int_\Omega f \, d\alpha$ between bounded continuous functions $f \in \mathcal{C}(\Omega)$ and probability measures $\alpha \in \mathcal{P}(\Omega)$, $\{f \oplus g\}(x,y) = f(x) + g(y)$ is the direct sum and $\mu \otimes \nu \in \mathcal{P}(X \times Y)$ the tensor product. Finally 1_Ω is the characteristic function, i.e. a constant 1 on Ω.

Under suitable hypothesis on c (as they are technical and satisfied for the costs in this paper, we skip this part), the OT problem is well posed and the the optimal transference plan γ is concentrated on a graph of the OT map

$y = T(x)$ implicitely defined by the saturation of the dual constraint :

$$f(x) + g(T(x)) = c(x, T(x)),\ \mu\, a.e.$$

The pair (f, g) are called the Kantorovich potentials and is unique up to an additive constant .

By construction T is a measure preserving map characterizing the transport. The measure preserving property is usually denoted $\nu = T\#\mu$ (T pushes forward μ to ν). The pushforward of μ is the measure defined as

$$\nu(A) = T_{\#}\mu(A) = \mu(T^{-1}(A)) \text{ for all } \nu \text{ measurable subset } A$$

Remark 1 (L^p Wasserstein metric)**.** *For complete separable metric space X and L^p costs $c := 1/p\, d^p(x, y)$, this OT problems defines a separable metric on the set of probability measures with finite second moments: the "Wasserstein" distance, which is given by $W_p^p(\mu, \nu) := OT(\mu, \nu)$.*
This metric metrizes weak convergence of measures and is a fundamential tool in image processing (see [22]).

In [20], Wang shows that the point source reflector model can be translated to an OT problem. More precisely, he proved the following theorem :

Theorem 2. *Let $S_0 \in \mathbb{S}^{d-1}$ and $S_\infty \in \mathbb{S}^{d-1}$ be connected domains in northern and southern hemispheres respectively, μ and ν which represent the given illuminance and illumination probability distributions. Then theorem 1 applies to the cost function*

$$c(x, y) = -\log(1 - x \cdot y).$$

A transport map T satisfying (3.1.1) exists and the solution of the corresponding OT problem can be used to build the desired reflector.

The construction of the reflector can be summarized as follows : Taking the exponential of the dual constraints and the saturation property (3.1.1) we get

$$\frac{e^{-g(T(x))}}{1 - x \cdot T(x)} = e^{f(x)} \leq \frac{e^{-g(y)}}{1 - x \cdot y},\ \mu \otimes \nu\ a.e.$$

We now define in $\mathbb{R}^d$ a family of parabolic reflectors with axis $y \in S_\infty : x \in S_0 \to P_y(x) := \dfrac{e^{-g(y)}}{1 - x \cdot y}$. And directly infer that the reflector shape parameterized over the directions in S_0 and given as :

$$R = \{x e^{f(x)} | x \in S_0\}.$$

Under this choice the map $x \to T(x)$ can be interpreted as the specular reflection of an optical ray at $R(x)$ onto a parabola of axis $T(x)$ while the illumination and illuminance constraints are enforced by (3.1.1).

3.1.2. Entropic Regularization of Optimal Transport

Entropic regularization has been introduced for OT computations in [23] (see [22] for a comprehensive review). The entropic regularization of the Kantorovich problem (1) is based on the following KullBack-Leibler divergence or "relative entropy" (KL) penalization :

$$\begin{aligned} OT_\epsilon(\mu, \nu) := \quad & \min_{\gamma_\epsilon \in \Pi(\mu,\nu)} \langle c, \gamma_\epsilon \rangle_{X \times Y} + \epsilon\ \mathrm{KL}(\gamma_\epsilon \,|\, \mu \otimes \nu) = \\ & \max_{f_\epsilon, g_\epsilon} \langle f_\epsilon, \mu \rangle_X + \langle g_\epsilon, \nu \rangle_Y - \epsilon \left\langle \exp(\tfrac{1}{\epsilon}(f_\epsilon \oplus g_\epsilon - c)) - 1, \mu \otimes \nu \right\rangle_{X \times Y} \end{aligned}$$

where $\epsilon > 0$ is a small "temperature" parameter (see [24] for a Statistical Physics interpretation of this problem due to Schroedinger) and

$$\mathrm{KL}(\gamma \,|\, \mu \otimes \nu) := \int_{X\times Y} \log(\frac{d\gamma}{d\mu \otimes d\nu})\, d\gamma \text{ if } \gamma \text{ is absolutely continuous w.r.t to } \mu \otimes \nu \text{ and } +\infty \text{ else.}$$

The primal-dual optimality condition is given by

$$\gamma_\epsilon = \exp(\frac{1}{\epsilon}(f_\epsilon \oplus g_\epsilon - c))\, \mu \otimes \nu.$$

The optimal entropic plan is therefore the scaling by the Kantorovich potentials of a fixed Kernel $\exp(-\frac{1}{\epsilon}c)$.

Remark 2. *Of course, altering the desired target functional (1) results in altered solution and thereof γ_ϵ is not the exact transport plan that we are looking for. It is diffuse, i.e. not concentrated on a map, and ϵ can be interpreted as a bandwidth under which the transport is blurred.*

Although, this entropic plan γ_ϵ converges to γ, the minimizer of (1), when ϵ goes to 0.(see [22])

3.1.3. Sinkhorn Algorithm for Regularized Optimal Transport

Numerical solutions are produced using the discretization of this problem, i.e. replacing $(X,\, Y\,, c\,, \mu,\, \nu)$ by $(X_N,\, Y_N\,, c_N\,, \mu_N\, \nu_N)$ in the following way:

$$\mu_N = \sum_{i=1}^{N} p_i \delta_{x_i}, \quad \nu_N = \sum_{j=1}^{N} q_j \delta_{y_j}, \quad where \quad \sum_{i=1}^{N} p_i = \sum_{j=1}^{N} q_j = 1.$$

Of course the number of discrete points for μ and ν may differ, we keep N for both to simplify the presentation.

This discretisation provides a natural discretization of the OT problem (1). Settting $X_N = \{x_i\}_{i=1..N}$, $Y_N = \{y_j\}_{j=1..N}$, $c_N = \{c(x_i, y_j)\}_{i,j=1..N}$, $q = \{q_j\}_{j=1..N}$ and $p = \{p_i\}_{i=1..N}$. we can again use the $\langle ., . \rangle$. notation :

$$OT_N(p, q) := \min_{\gamma_N \in \Pi(p,q)} \langle c_N, \gamma_N \rangle_{X_N \otimes Y_N}$$

where

$$\Pi(p, q) := \left\{ \gamma_N \in \mathbb{R}_+^{N\times N} | \langle 1_{X_N}, \gamma_N \rangle_{Y_N} = p,\ \langle 1_{Y_N}, \gamma_N \rangle_{X_N} = q \right\}$$

Similarly, discretization of regularized problem (3.1.2) gives

$$OT_{\epsilon,N} := \max_{f_\epsilon, g_\epsilon} \langle f_\epsilon, \mu_N \rangle_{X_N} + \langle g_\epsilon, \nu_N \rangle_{Y_N} - \epsilon \left\langle \exp(\frac{1}{\epsilon}(f_\epsilon \oplus g_\epsilon - c_N)) - 1, \mu_N \otimes \nu_N \right\rangle_{X_N \times Y_N}.$$

where we use the same notation (f_ϵ, g_ϵ) for discrete vectors in $\mathbb{R}^N$.

We solve (3.1.3) with Sinkhorn algorithm. It corresponds to a block coordinate (f_ϵ and g_ϵ) ascent : Initialize with $g_\epsilon^0 = 0_Y$ and then iterate (in k) :

$$f_\epsilon^{k+1} = -\epsilon \log(\langle \exp(\tfrac{1}{\epsilon}(g_\epsilon^k - c_N)), \nu_N \rangle_{Y_N})$$

$$g_\epsilon^{k+1} = -\epsilon \log(\langle \exp(\tfrac{1}{\epsilon}(f_\epsilon^{k+1} - c_N)), \mu_N \rangle_{X_N})$$

As discussed in [22](Remark 4.13), for sufficiently regular data (for example when exact map T is guaranteed to be smooth) following estimate holds for sufficiently large number of iterations k in (3.1.3):

$$\sup_{X_N} |f_\epsilon(x) - f_\epsilon^k(x)| = O(1 - \epsilon)^k$$

Where f_ϵ is an exact regularized potential of (3.1.3).

3.1.4. Benchmark Cases

The following benchmark cases are being discussed in this chapter:

- Computing a reflector for the problem where the Source is a uniform distribution with support on a set, which is the inverse stereographic projection of unit square centered at the origin and the desired light distribution is a uniform distribution with support on a set, which is an inverse stereographic projection of circle centered at the origin and Diameter 1.
- Computing a reflector for the problem where the Source is a uniform distribution with support on a set, which is an inverse stereographic projection of unit square centered at the origin and the desired light distribution is a sum of two gauss distributions which are the centered respectively at inverse stereographic projection of points (0.25,-0.25),(0.25,0.25).

3.2. Hierarchical approach to Sinkhorn Algorithm

3.2.1. ϵ scaling

As mentioned in remark (2), decreasing ϵ would result in a more accurate solution for (1). On the other hand, estimate (3.1.3) suggests that smaller ϵ we take, higher number of iterations will be required for Sinkhorn algorithm to converge. Also, taking ϵ too small, would result into numerical overflows due to the exponential terms of order $e^{\frac{1}{\epsilon}}$ in (3.1.3)

As discussed in [25], problem of numerical stability can be tackled by working with the increments of the potentials rather then full potentials during the iterative steps.

That is, if we look at the updates f_ϵ^{k+1} and g_ϵ^{k+1} in (3.1.3) as $f_\epsilon^{k+1} = f_\epsilon^k + \hat{f}_\epsilon^{k+1}$ and $g_\epsilon^{k+1} = g_\epsilon^k + \hat{g}_\epsilon^{k+1}$, then by moving previous approximations to the right hand side, we will get the following new iterative scheme for the increments:

$$\begin{aligned}
\hat{f}_\epsilon^{k+1} &= -\epsilon \log(\langle \exp(\tfrac{1}{\epsilon}(g_\epsilon^k + f_\epsilon^k - c_N)), \nu_N \rangle_{Y_N}) \\
f_\epsilon^{k+1} &= f_\epsilon^k + \hat{f}_\epsilon^{k+1} \\
\hat{g}_\epsilon^{k+1} &= -\epsilon \log(\langle \exp(\tfrac{1}{\epsilon}(f_\epsilon^{k+1} + g_\epsilon^k - c_N)), \mu_N \rangle_{X_N}) \\
g_\epsilon^{k+1} &= g_\epsilon^k + \hat{g}_\epsilon^{k+1}
\end{aligned}$$

Those iterations will be more stable due to the saturation property of the optimizing potentials (3.1.1). This property tells us that quantity $f(x_i) + g_(y_j) - c(x_i, y_j)$ is zero for exact potentials and optimal pairs (x_i, y_j) while being strictly negative for non-optimal pairs. Thereof, when the iterates f_ϵ^k and g_ϵ^k are close to the true potentials, new updating steps would not cause a numerical overflow.

Although, this approach alone would not help at the first steps of the algorithm, since we have no guarantees that initial approximations would be close to the exact potentials, and for small ϵ we would get an overflow at the first step of the iterations. In order to avoid this, possible approach would be to start with higher values of ϵ and gradually decrease it to the desired final value ϵ_{final} (see [25] [26]).

More formally, one can define a sequence of regularization parameters $\epsilon_k \to \epsilon_{final}$ and use ϵ_k at k-th iteration in (3.2.1). A common choice is to start with $\epsilon_0 = 1$. We use a scaling parameter $\lambda \in (0, 1)$ and define $\epsilon_k := \max\{\epsilon_{final}, \lambda^k \epsilon_0\}$.

Remark 3. *It has been empirially established (see [25] and references therein), that above discussed approach of gradually increasing ϵ_k at each iteration, not only provides more numerically stable scheme, but also increases the convergence speed. In other words, a smaller number of iterations is required for achieving a given error threshold with decreasing ϵ_k at each iteration, then while using fixed ϵ_{final} for all iterations.*

3.2.2. Discretization scaling

In [25] (see also [27]), it is discussed that the entropic regularization with ϵ acts as a smoothing filter on the data, which smoothers out any details that are on the finer scale then ϵ. This means that using Sinkhorn iterations with discretizations such that $\min_{i,j} d(x_i, x_j) << \epsilon$ does not provide any valuable improvement over working with discretizations that are on the scale of ϵ.

Thereof, it would be more efficient to also use a sequence of discretizations $(X_{N_k}, Y_{N_k}, c_{N_k}, \mu_{N_k}, \nu_{N_k})$ where $N_k = O(\frac{1}{\epsilon_k})^d$ (where d is the dimension of the problem). In order to implement this approach, one would need to find a way to interpolate approximations $f_\epsilon^k, g_\epsilon^k$ on the discretization $X_{N_{k+1}}, Y_{N_{k+1}}$ respectively, while they are computed on the grids X_{N_k}, Y_{N_k}.

Luckily, Sinkhorn algorithm provides a canonical way of computing such interpolations, even for the full spaces X and Y. If we expand the definition of scalar product in (3.1.3) and replace $c_N = c_N(x_i, y_j)$ by $c(x, y_j)$ and $c(x_i, y)$ respectively, we obtain following continuous extensions for given approximations $f_{\epsilon_k}^k$ and $g_{\epsilon_k}^k$:

$$\tilde{f}_{\epsilon_k}^k(x) := -\epsilon_k \log\Big(\sum_{j=1..N_k} \exp\big(\frac{1}{\epsilon_k}(g_{\epsilon_k}^k(y_j) - c(x, y_j))\big)\nu_{N_k}(y_j)\Big), \ \forall x \in X.$$

$$\tilde{g}_{\epsilon_k}^k(y) := -\epsilon_k \log\Big(\sum_{i=1..N_k} \exp\big(\frac{1}{\epsilon_k}(f_{\epsilon_k}^k(x_i) - c(x_i, y))\big)\mu_{N_k}(x_i)\Big), \ \forall y \in Y.$$

Thereof, at k-th iteration, we can take $k-1$-th approximations to be restrictions of $\tilde{f}_\epsilon^{k-1}(x)$ and $\tilde{f}_\epsilon^{k-1}(x)$ on the spaces X_{N_k} and Y_{N_k} respectively.

Putting it all together, we obtain the following iterative procedure in k:

$$\begin{aligned}
f_{\epsilon_k}^{k-1} &= \tilde{f}_{\epsilon_{k-1}}^{k-1}|_{X_{N_k}} \qquad g_{\epsilon_k}^{k-1} = \tilde{g}_{\epsilon_{k-1}}^{k-1}|_{Y_{N_k}} \\
\hat{f}_{\epsilon_k}^k &= -\epsilon_k \log(\langle \exp(\tfrac{1}{\epsilon_k}(g_{\epsilon_{k-1}}^{k+1} + f_{\epsilon_{k-1}}^{k-1} - c_{N_k})), \nu_{N_k}\rangle_{Y_{N_k}}) \\
f_{\epsilon_k}^k &= f_{\epsilon_k}^{k-1} + \hat{f}_{\epsilon_k}^k \\
\hat{g}_{\epsilon_k}^k &= -\epsilon_k \log(\langle \exp(\tfrac{1}{\epsilon_k}(f_{\epsilon_k}^k + g_{\epsilon_k}^{k-1} - c_{N_k})), \mu_{N_k}\rangle_{X_{N_k}}) \\
g_{\epsilon_k}^k &= g_{\epsilon_k}^{k-1} + \hat{g}_{\epsilon_k}^k
\end{aligned}$$

In this setting, taking ϵ_{final} to 0 means also refining the discretization. To the best of our knowledge the joint convergence in N and ϵ has only be studied in [27]:

Theorem 3 (Berman joint convergence - corollary 1.3 [27])**.** *We assume μ and ν are in $\mathcal{C}^{2,\alpha}$ and positive, and that N and ϵ are dependent parameters : $N = (1/\epsilon)^d$ where d is the dimension of the problem. A technical condition on the sequence of discretization $(X_N, Y_N, c_N, \mu_N \nu_N)$ called "density propery" (see remark 5 below) is also necessary. Then there exists a positive constant A_0 such that for any $A > A_0$ the folowing holds : setting $m_\epsilon = [-A\log(\epsilon)/\epsilon]$ the continuous interpolation provided by $\tilde{f}_\epsilon^{m_\epsilon}$, built using the cannonical extension (3.2.2) from the discrete Sinkhorn iterate at $k = m_\epsilon$, satisfies the estimate*

$$\sup_X |\tilde{f}_\epsilon^{m_\epsilon} - f| \leq -C\epsilon\log(\epsilon)$$

for some constant C (depending on A) and f an optimal potential for (1).

Remark 4. *Assumptions of Theorem 3 holds on the sphere for the reflector cost (see section 6.3.3 [27]). However, while estimating the necessary number of iterations m_ϵ, this theorem does not take into account the improved effect on the convergence, coming from the ϵ-scaling.*

Remark 5 (Density property Lemma 3.1 [27]). *For any given open set U intersecting the support X of μ (same for Y and ν)*

$$\liminf_{\epsilon \to 0} \epsilon \log(\mu_N(U)) = 0$$

For the flat space $X \subset \mathbb{R}^d$, this condition is enough. For curved surfaces, a technical generalization is required. But in both cases, this density property ensures the discretization of X and μ (3.1.3) is such that, for U the sequence of approximations $\mu_N(U)$ never converges faster to 0 than ϵ (remember that $N = (1/\epsilon)^d$).

For the sphere this can be achieved by either the Quasi Monte-Carlo discretizations that are sampled uniformly with respect to the surface element of the sphere (see [28]), or by adjusting the weights of the discretization points according to the deviation from the surface element (e.g. for the orthogonal grids projected from a plane to the sphere).

3.3. Entropic Bias

3.3.1. Entropic Bias and Sinkhorn Divergences

The rate of convergence, both in estimate (3.1.3) and in theorem 3, have infinite slope at $\epsilon = 0$. Because of this, it is a known issue, that even with above-discussed modifications, using computationally feasible values of ϵ will leave certain "entropic bias" in the approximate potentials.

This problem is discussed in depth in [29] where it is proposed, that in order to correct the bias, to add "diagonal terms" to correct the entropic cost :

$$S_\epsilon(\mu, \nu) = OT_\epsilon(\mu, \nu) - \frac{1}{2}(OT_\epsilon(\mu, \mu) + OT_\epsilon(\nu, \nu)).$$

Quite remarkably, the authors show that this quantity, called Sinkhorn divergence, remains positive and is convex. It also obviously vanishes for $\mu = \nu$ wich is not the case for OT_ϵ. Thanks to the symmetry, there is only one dual potential for each of diagonal problems. We denote them $f^\mu_{OT_\epsilon}$ and $f^\nu_{OT_\epsilon}$. They can be computed using the independent Sinkhorn iterations :

$$f^{\mu,k+1}_{OT_\epsilon} = -\epsilon \log(\langle \exp(\tfrac{1}{\epsilon}(f^{\mu,k}_{OT_\epsilon} - c_N)), \mu \rangle_X)$$
$$f^{\nu,k+1}_{OT_\epsilon} = -\epsilon \log(\langle \exp(\tfrac{1}{\epsilon}(f^{\nu,k}_{OT_\epsilon} - c_N)), \nu \rangle_Y)$$

The μ gradient of S_ϵ, denoted f_{S_ϵ} may be formed by a simple substraction.

$$f_{S_\epsilon} = f_\epsilon - f^\mu_{OT_\epsilon}$$

Numerical simulations of gradient flows in [29] indicate that f_{S_ϵ} is a better approximation of exact potential f. For more comperhensive review of entropic bias and it's effect on the reflector problem see [30].

3.4. Implementation

Although theoretically it is more efficient to conduct the iterations on the appropriate discretization at every iteration as in (3.2.2), in practice, altering memory and memory containers at every iteration is not feasible due to hardware properties.

It is a common knowledge in software engineering, that arranging computations in a way that memory is accessed in a continuous way, so that processor doesn't have to wait for the delivery of necessary memory components, produces better practical computational time even when theoretical count of operations is far larger.

With this in mind, depending on the desired ϵ_{final} and $N_{\epsilon_{final}}$, we define two discretization levels : $N_{small} =$

$O(N_{final})^{\frac{1}{2d}}$ and $N_{large} = O(N_{final})$. Similarly, 2 different intermediate values of ϵ are chosen: $\epsilon_{small} = \epsilon_{final}^{\frac{1}{2}}$, $\epsilon_{large} = \epsilon_{final}$.

The sequence ϵ_k is initialized by $\epsilon_0 = 1$ and for $k > 0$ $\epsilon_k = \left(\epsilon_{k-1}^{-1} + \epsilon_{current}^{-\frac{1}{3}}\right)^{-1}$ where $\epsilon_{current}$ is either ϵ_{small} or ϵ_{large}.

With this choice, outline of hierarchical sinkhorn algorithm for reflector problem would be following:

Data: Source and target distributions μ, ν
Result: Approximations of Kantorovich potentials $f_{\epsilon_{final}}$, $g_{\epsilon_{final}}$

Initialization: $k = 0$, $N = N_{small}$, $\epsilon_{current} = \epsilon_{small}$, $f^0 \equiv 0$, $g^0 \equiv 0$;
while $\epsilon_k < \epsilon_{final}$ **do**
  $\hat{f}_\epsilon^{k+1} = -\epsilon_k \log(\langle \exp(\frac{1}{\epsilon_k}(g_\epsilon^k + f_\epsilon^k - c_N)), \nu_N \rangle_{Y_N})$;
  $f_\epsilon^{k+1} = f_\epsilon^k + \hat{f}_\epsilon^{k+1}$;
  $\hat{g}_\epsilon^{k+1} = -\epsilon_k \log(\langle \exp(\frac{1}{\epsilon_k}(f_\epsilon^{k+1} + g_\epsilon^k - c_N)), \mu_N \rangle_{X_N})$;
  $g_\epsilon^{k+1} = g_\epsilon^k + \hat{g}_\epsilon^{k+1}$;
  k=k+1
  if $\epsilon_k > \epsilon_{current}$ **then**
    if $\epsilon_{current} = \epsilon_{final}$ **then**
      Stop;
    else
      $N = N_{final}$, $\epsilon_{current} = \epsilon_{final}$;
      $f_\epsilon^k = \tilde{f}_{\epsilon_k}^k|_{X_N}$, $g_\epsilon^k = \tilde{g}_{\epsilon_k}^k|_{Y_N}$;
$f_{\epsilon_{final}} = f_{\epsilon_k}^k - f_{OT_\epsilon}^\mu$;
$g_{\epsilon_{final}} = g_{\epsilon_k}^k - f_{OT_\epsilon}^\nu$;

Here de-biasing terms $f_{OT_\epsilon}^\mu$ and $f_{OT_\epsilon}^\nu$ coming from (3.3.1) are computed using the diagonalized versions of above algorithm:

Data: Source or target distribution μ and corresponding space X
Result: Approximation of Kantorovich potential $f_{\epsilon_{final}}^\mu$

Initialization: $k = 0$, $N = N_{small}$, $\epsilon_{current} = \epsilon_{small}$, $f^0 \equiv 0$;
while $\epsilon_k < \epsilon_{final}$ **do**
  $\hat{f}_\epsilon^{k+1,\mu} = -\epsilon_k \log(\langle \exp(\frac{1}{\epsilon_k}(f_\epsilon^{k+1,\mu} + f_\epsilon^{k+1,\mu} - c_N)), \mu_N \rangle_{X_N})$;
  $f_\epsilon^{k+1,\mu} = f_\epsilon^{k,\mu} + \frac{1}{2}\hat{f}_\epsilon^{k+1,\mu}$;
  k=k+1;
  if $\epsilon_k > \epsilon_{current}$ **then**
    if $\epsilon_{current} = \epsilon_{final}$ **then**
      Stop;
    else
      $N = N_{final}$, $\epsilon_{current} = \epsilon_{final}$;
      $f_\epsilon^{k,\mu} = \tilde{f}_{\epsilon_k}^{k,\mu}|_{X_N}$;
$f_{\epsilon_{final}}^\mu = f_{\epsilon_k}^{k,\mu}$;

3.5. Computer Requirements

The software is written in C++, but most of the computational data structures are C-style fixed-size Arrays and most of the computational work is done either by basic operations available in C as well, or by Intel's Math Kernel Library. No manually written classes are used and C++ standard library functions are used for secondary tasks, such as data filling, data sorting or time counting.

Intel's Math Kernel Library (MKL) is a library of optimized math routines. It includes BLAS, LAPACK, ScaLAPACK, sparse solvers, fast Fourier transforms, and vector math. The routines in MKL are hand-optimized specifically for Intel processors. In this software, only BLAS and vector math functions are used.

The library is available free of charge under the terms of Intel Simplified Software License which allows redistribution. Commercial support is available when purchased as standalone software or as part of Intel Parallel Studio XE or Intel System Studio. It can be downloaded from Intel's official web-page https://software.intel.com/mkl where installation instructions are also provided.

Due to high importance of good memory management, and software's primary purpose for now being the development of the method, code doesn't follow standard suggestions for C++ code development.

C++ compiler with version 11 or higher is required, as code uses timing functions and arithmetic of "inf" and "nan" values introduced in this version.

The code is OS independent as long as appropriate compiler and ability to link with MKL library are available, except, for convenience of output handling, system command is called to create and move folders. Right now code uses Linux commands "mkdir " and "mv ". For other operating systems one could just change those commands in the main function.

Due to the use of Intel's MKL library, the software will be much more efficient when running on the Intel processor, compared to other processors of the same power. Other then the capability of installing Intel's MKL Library, there is no other definitive hardware requirement, but for computational stability, it is desirable for double-precision floating-point variable to hold numbers up to 16 digit precision, so it is recommended that hardware is capable of handling such precision.

3.5.1. Runing Manual

How to compile files that use MKL libraries, can be found on the Intel's official webpage: https://software.intel.com/

The software includes a Makefile file, which can be used on linux environment to compile the code, if the cmake package is present. Although it is important to verify the location of the MKL library on the system, and adjust the file accordingly.

If the cmake package is absent, one can copy the line inside the Makefile into a terminal with workind directory at the location of the code files (of course, after adjusting the MKL library location).

After the code is compiled, it can be run from a terminal by executing the compiled object.

In order to switch between the benchmark cases, one should edit the first line of main.cpp file, specifing the address of the desired benchmark case, as instructed in the code.

3.6. Numerical Demonstration

Here we demonstrate that, for the benchmark cases, using ϵ-scaling speeds up the computation by decreasing number of iterations required for achieving a given precision in Sinkhorn algorithm. Also, using discretization scaling does not worsen total number of iterations, while achieving speedup by using cheaper iterations on the first stage.

As stopping criteria for the native sinkhorn algorithm (3.2.1), we use the absolute value of the change $\tilde{f}_\epsilon^{k+1}$ in the first potential. In order to achieve a fair comparison with the ϵ-scaling algorithms, we also force ϵ-scaling

algorithms to continue with native iterations after reaching final value of regularization parameter, until they achieve same threshold.

Our input are the discretization of the analytical descriptions of the illumination/illuminance μ and ν described below. All benchmark cases presented in this paper will have the same source and target domains X and Y. The source domain $X \subset \mathbb{S}^2$ will be the inverse stereograpic projection in the northern hemisphere of the square domain centered at the origin $\{(x_1, x_2) \in \mathbb{R}^2 | -0.6 \leq x_1 \leq 0.6, -0.6 \leq x_2 \leq 0.6\}$. Similarly, $Y \subset \mathbb{S}^2$ will be the inverse stereographic projection in the southern hemisphere of same domain.

As discussed in remark 5 (see also [27]), we discretize those domains using Quasi Monte-Carlo discretizations from [28]. We take $N = 16488 \approx 128 * 128$ points in each discretization. For discretization scaling, we use $N_{small} = 381$. We compute each benchmark case with two different values for final ϵ, $\frac{1}{4*128}$ and $\frac{1}{16*128}$. For the second value, native sinkhorn algorithm is not applicable as we get an overflow on the very first iteration.

As according to convergence result from Theorem 3, for given $\epsilon = O(N)^{-\frac{1}{d}}$ we can expect only approximations of order $\epsilon log(\epsilon)$, we take $\tilde{f}_\epsilon^k < 1.e-5$ as a stopping criteria, since when changes become smaller, each new iteration adds less improvement, and approximation error becomes dominant.

Benchmark Case 1: Square To Circle. The source distribution μ will be the uniform distribution over the set with a square stereographic projection $StP(supp(\mu)) = \{(x_1, x_2) \in \mathbb{R}^2 | -0.5 \leq x_1 \leq 0.5, -0.5 \leq x_2 \leq 0.5\}$. The target distribution ν will be an uniform distribution over the set with a disck (circle) stereographic projection $StP(supp(\mu)) = \{(x_1, x_2) \in \mathbb{R}^2 | x_1^2 + x_2^2 \leq 0.5^2\}$. Even though the densities are constant, mapping from a non smooth support geometry of the square to the smooth geometry of a circle is not a trivial task.

ϵ_{final}	Type of Algorithm	Number of iterations
$\frac{1}{4*128}$	Native	146
	ϵ-scaling	116
	Discretization scaling	121
$\frac{1}{16*128}$	Native	NA(Numeric Overflow)
	ϵ-scaling	171
	Discretization scaling	182

Benchmark Case 2: Square To Two Gaussians. The source distribution μ is the same as in the previous. The target distribution ν is a gaussian distribution with density on the projected domain $\rho(x_1, x_2) = e^{-16*((x_1-0.25)^2+(x_2-0.25)^2)} + e^{-16*((x_1-0.25)^2+(x_2+0.25)^2)}$ over whole target domain Y

ϵ_{final}	Type of Algorithm	Number of iterations
$\frac{1}{4*128}$	Native	NA (Slow convergence)
	ϵ-scaling	202
	Discretization scaling	175
$\frac{1}{16*128}$	Native	NA(Numeric overflow)
	ϵ-scaling	502
	Discretization scaling	184

An interesting phenomena occurs for this case. Since the target has very high steepness, convergence speed for native algorithm, even for moderate value of ϵ, is extremely slow. Even after 1000 iterations, absolute value of the incrementing term was of order $1.e-3$. On the other hand, scaling algorithm managed to converge with comparable number of iterations as in previous case.

Part IV.
Reduced Order Multirate Simulation of Circuits

M.W.F.M. Bannenberg, A. Ciccazzo, M. Günther

Abstract

A benchmark case is presented for the application of reduced order multirate schemes in a MATLAB environment. Starting with the problem formulation and then a synopsis of the mathematical techniques applied in the benchmark a test case is discussed. Then the numerical approach to the simulation implementation is outlined, and finally a numerical experiment is shown verifying the validity of this benchmark case.

Keywords: Model Order Reduction, Multirate, Differential Algebraic Equations, Coupled Systems, Circuit Simulation.

Latest release: https://doi.org/10.5281/zenodo.5171813

GitHub repository: https://github.com/ROMSOC/benchmarks-ROMR-schemes

4.1. Introduction

In integrated circuit design, there are a significant number of design possibilities under which the internal components need to be guaranteed to work. This leads to a whole range of explorations to ensure sound functionality of the design. These explorations are performed by numerical simulations of the circuits mathematical model. Due to the ever increasing number of components, and thus the degrees of freedom in the model, the required simulation times may become prohibitively large.

Besides the sheer number of components inside the integrated circuit, a large contribution to the complexity of the mathematical model originates form the method of deriving these equations. As these models grow, generating a state-space model with a minimal set of unknowns cannot be generated in an automatic way. Therefore, the mathematical models have to be derived through use of algorithmic analysis. This automation comes at a cost. The resulting system of differential-algebraic equations (DAE) is numerically harder to solve, and may contain redundant network variables.

To decrease these ever increasing simulation costs a multitude of different approaches have been proposed in the past decades. For instance, the redundancy originating from the network analysis can be exploited by using model order reduction. This technique aims to solve a model of reduced size that still approximates the solution of the original model. Furthermore, the large original system can be partitioned into subsystems which each have their own characteristic rate of evolution through time. This property is capitalised upon by using multirate (MR) time integration. This paper is specifically aimed at the combination of the two previously mentioned techniques, and provides an overview from definition to implementation.

4.2. Problem Formulation

Consider the following coupled system of two semi explicit DAE systems, where the subscripts $\{F, S\}$ indicate a fast or slow time-scale, respectively, and independent transient sources have been omitted for notational convenience:

$$\frac{d}{dt}y_{\mathrm{F}} = f_{\mathrm{F}}(t, y_{\mathrm{F}}, z_{\mathrm{F}}, y_{\mathrm{S}}, z_{\mathrm{S}}), \quad y_{\mathrm{F}}(t_0) = y_{F_0}, \tag{4.1a}$$

$$0 = g_{\mathrm{F}}(t, y_{\mathrm{F}}, z_{\mathrm{F}}, y_{\mathrm{S}}, z_{\mathrm{S}}), \quad z_{\mathrm{F}}(t_0) = z_{F_0}, \tag{4.1b}$$

$$\frac{d}{dt}y_{\mathrm{S}} = f_{\mathrm{S}}(t, y_{\mathrm{F}}, z_{\mathrm{F}}, y_{\mathrm{S}}, z_{\mathrm{S}}), \quad y_{\mathrm{S}}(t_0) = y_{S_0}, \tag{4.1c}$$

$$0 = g_{\mathrm{S}}(t, y_{\mathrm{F}}, z_{\mathrm{F}}, y_{\mathrm{S}}, z_{\mathrm{S}}), \quad z_{\mathrm{S}}(t_0) = z_{S_0}, \tag{4.1d}$$

with the functions $f_A : \mathbb{R} \times \mathbb{R}^a \times \mathbb{R}^b \times \mathbb{R}^c \times \mathbb{R}^d \to \mathbb{R}^a$, with $A \in \{\mathrm{F}, \mathrm{S}\}$, where $\{a, b, c, d\} \in \mathbb{N}$ are the respective dimensions, and equivalent definitions for g_A. Consistent initial conditions are assumed, which means that Equations (4.1b) and (4.1d) are satisfied at initial time t_0. The quantities $y_{\{F,S\}} : I \to \mathbb{R}^{\{a,b\}}$ and $z_{\{F,S\}} : I \to \mathbb{R}^{\{c,d\}}$ denote the differential and algebraic variables defined on the time interval $[t_0, t_1] = I$. Both subsystems and the joint system are guaranteed to be index-1 by the assumption that the Jacobians

$$\frac{\partial g_\mathrm{F}}{\partial z_\mathrm{F}}, \frac{\partial g_\mathrm{S}}{\partial z_\mathrm{S}} \text{ and } \begin{pmatrix} \frac{\partial g_\mathrm{F}}{\partial z_\mathrm{F}} & \frac{\partial g_\mathrm{F}}{\partial z_\mathrm{S}} \\ \frac{\partial g_\mathrm{S}}{\partial z_\mathrm{F}} & \frac{\partial g_\mathrm{S}}{\partial z_\mathrm{S}} \end{pmatrix} \text{ are invertible}$$

in the neighbourhood of the solution of the system. From this assumption the algebraic variables $z_{\{F,S\}}$ can be solved locally by using the implicit function theorem

$$\begin{aligned} z_F &= G_{t,F}(y_\mathrm{F}, z_S, y_\mathrm{S}), \\ z_S &= G_{t,S}(y_\mathrm{F}, z_F, y_\mathrm{S}), \end{aligned}$$

where the second z subscript is the opposite of the first z subscript. The partition of the system into subsystems can originate from different physical systems, such as temperature diffusion and electric currents. However, differences in time scale can also be identified by different orders of time derivatives. Here the partition is considered to be fixed during the time integration.

4.3. The Reduced Order Multirate Method

In this section a brief overview of the mathematical aspects of the reduced order multirate method is presented. For a more in depth description see [31, 32, 33].

4.3.1. Maximum Entropy Snapshot Sampling

Let m and n be positive integers and $m \gg n > 1$. Define a finite sequence $X = (x_1, x_2, ..., x_n)$ of numerically obtained states $x_j \in \mathbb{R}^m$ at time instances $t_j \in \mathbb{R}$, with $j \in \{1, 2, ..., n\}$, of a dynamical system governed by either ODEs or DAEs. Provided a probability distribution p of the states of the system, the second-order Rényi entropy of the sample X is

$$H_p^{(2)}(X) = -\log \sum_{j=1}^{n} p(x_j)^2 = -\log \mathbb{E}(p(X)),$$

with $\mathbb{E}(p(X))$ the expected value of the probability distribution p with respect to p itself. When n is large enough, according to the law of large numbers, the average of $p_1, p_2, ..., p_n$ almost surely converges to their expected value,

$$\frac{1}{n} \sum_{j=1}^{n} p(x_j) \to \mathbb{E}(p(X)) \quad \text{as } n \to \infty,$$

thus each $p(x_j)$ can be approximated by the sample's average sojourn time or relative frequency of occurrence. To obtain this frequency of occurrence, consider a norm $\| \cdot \|$ on $\mathbb{R}^m$. Then the notion of occurrence can be translated into a proximity condition. In particular, for each $x_j \in \mathbb{R}^m$ define the open ball that is centred at x_j and whose radius is $\epsilon > 0$,

$$B_\epsilon(x) = \{y \in \mathbb{R}^m \mid \|x - y\| < \epsilon\},$$

and introduce the characteristic function with values

$$\chi_i(x) = \begin{cases} 1, & \text{if } x \in B_\epsilon(x_i), \\ 0, & \text{if } x \notin B_\epsilon(x_i). \end{cases}$$

Under the aforementioned considerations, the entropy of X can be estimated by

$$\hat{H}_p^{(2)}(X) = -\log\left(\frac{1}{n^2}\sum_{i=1}^{n}\sum_{j=1}^{n}\chi_i(x_j)\right). \tag{4.3}$$

Provided that the limit of the evolution of $\hat{H}_p^{(2)}$ exists, for n large enough, and measures the sensitivity of the evolution of the system itself [34, §6.6], a reduced sequence $X_\mathrm{r} = (\bar{x}_{j_1}, \bar{x}_{j_2}, \ldots, \bar{x}_{j_r})$, with $r \leq n$, is sampled from X, by requiring that the entropy of X_r is a strictly increasing function of the index $k \in \{1, 2, \ldots, r\}$ [35]. The state vector $\bar{x}_{j_k}$ added to sampled snapshot space is the average value of all states in the selected ϵ-ball. A reduced basis is then generated from X_r with any orthonormalization process.

The Estimation of ϵ: The open ball parameter ϵ, which is directly responsible for the degree of reduction within the MESS framework, can be chosen arbitrarily, much like the number of selected basis vectors provided by a POD approach. For a ballpark estimate of this parameter the following optimisation approach is provided. The quantity within the logarithm in the entropy estimate (4.3) is often referred to as the sample's correlation sum and can be written as

$$C_\epsilon = \frac{1}{n^2}\|R_\epsilon\|_\mathrm{F}^2,$$

with $R_\epsilon \in \{0,1\}^{n\times n}$ being the recurrence matrix whose entries are unity, when $\|x_i - x_j\| < \epsilon$, and $\|\cdot\|_\mathrm{F}^2$ being the Frobenius norm. In terms of probability theory, C_ϵ is a cumulative distribution function of ϵ, and hence, its derivative $\mathrm{d}C_\epsilon/\mathrm{d}\epsilon$ is the associated probability density function of ϵ. A commonly justified hypothesis is that the correlation sum scales as ϵ^D [36, Chapter 1], with $D \geq 0$ being the so-called correlation dimension of the manifold that is formed in $\mathbb{R}^m$ by the terms of X. Under this power law assumption, the maximum likelihood estimate [37, Chapter 8] of the correlation dimension is estimated as follows. We find a sample $\{\epsilon_i\}$, with $\epsilon_i \in [0,1]$ for all $i \in \{1, 2, \ldots, q\}$, of a random variable E that is sampled according to C_ϵ. Then, the probability of finding a sample in $(\epsilon_i, \epsilon_i + \mathrm{d}\epsilon_i)$ in a trial is

$$\prod_{i=1}^{q} D\epsilon^{D-1}\mathrm{d}\epsilon_i.$$

To calculate the ϵ value for which this expression is maximized, we take the logarithm

$$q\cdot\ln D + (D-1)\sum_{i=1}^{q}\ln\epsilon_i,$$

and note that the maximum of this expression is attained when

$$\frac{q}{D} + \sum_{i=1}^{q}\ln\epsilon_i = 0.$$

This results in the most likely value $D_* = -1/\langle\ln E\rangle$. The value for ϵ_* is then estimated by choosing the ϵ from the sample that produces a quotient that is closest to D^*. Thus ϵ can be estimated by

$$\epsilon_* = \mathrm{argmin}(|D_* - \ln C_\epsilon/\ln\epsilon|).$$

4.3.2. The Gauß-Newton with approximated tensors method

Unfortunately, a direct application of MESS is not feasible in practice, [38, Section 7.6], therefore a simplified Gauß-Newton with Approximated Tensors (GNAT), equipped with a function-sampling-hyper-reduction scheme is used. Firstly, a direct Galerkin projection may yield an unsolvable reduced system for DAEs. Secondly, the computational effort required to solve this reduced system and the full system is about the same in

the nonlinear cases. This is due to the fact that the evaluation costs of the reduced system are not reduced at all because the projection basis will be a dense matrix in general.

Considering a general DAE in the form

$$\dot{\phi}(t,u) + \psi(t,u) = 0,$$

where ϕ and ψ are functions of time t and some state vector u. In the discrete case, we assume that the numerical scheme exactly solves the following nonlinear system for each time step t_i,

$$R(u) = 0, \tag{4.4}$$

where $u \in \mathbb{R}^N$, u^0 the initial condition and the residual $R : \mathbb{R}^N \to \mathbb{R}^N$. Note that for ease of notation, the relevant time subscripts have been omitted, as this equation is solved for each individual time step. For the reduction of the dimension of Equation (4.4), a projection is used to search the approximated solution in the incremental affine trial subspace $u^0 + \mathcal{V} \subset \mathbb{R}^N$. Thus $\tilde{u}$ is given by

$$\tilde{u} = u^0 + V_u u_r, \tag{4.5}$$

where $V_u \in \mathbb{R}^{N \times n_u}$ is the n_u-dimensional projection basis for $\mathcal{V}$, and u_r denotes the reduced incremental vector of the state vector. Now deviating from the direct Galerkin projection process, Equation (4.5) is substituted into Equation (4.4). This results in an overdetermined system of N equations and n_u unknowns. Because V_u is a matrix with full column rank, it is possible to solve this system by a minimisation in least-squares sense through

$$\min_{\tilde{u} \in u^0 + \mathcal{V}} ||R(\tilde{u})||_2.$$

This nonlinear least-squares problem is solved by the Gauß-Newton method, leading to the iterative process for $k = 1, ..., K$, solving

$$s^k = \text{argmin}_{a \in \mathbb{R}^{n_u}} ||J^k V_u a + R^k||_2,$$

and updating the search value w_r^k with

$$w_r^{k+1} = w_r^k + s^k,$$

where K is defined through a convergence criterion, initial guess w_r^0, $R^k \equiv R(u^0 + V_u w_r^k)$ and $J^k \equiv \frac{\partial R}{\partial u}(u^0, V_u u_r^k)$. Here J^k is the full order Jacobian of the residual at each iteration step k. Since the computation of this Jacobian scales with the original full dimension of Equation (4.4) this is a computational bottleneck. This bottleneck can be circumvented by the application of hyper reduction methods, for which this paper utilises a gappy data reconstruction method.

Gappy Maximum Entropy Snapshot Sampling: The evaluation of the nonlinear function $R(u_0 + V_u w_r^k)$ has a computational complexity that is still dependent on the size of the full system. To reduce the complexity of this evaluation the gappy MESS procedure, based on gappy POD, is applied. Like the gappy POD approach gappy MESS uses a reduced basis to reconstruct gappy data. However, unlike the gappy POD approach the basis used is now not obtained through POD but by MESS. Gappy MESS starts by defining a mask vector n for a solution state u as

$$n_j = 0 \text{ if } u_j \text{ is missing,}$$
$$n_j = 1 \text{ if } u_j \text{ is known,}$$

where j denotes the j-th element of each vector. The mask vector n is applied point-wise to a vector by $(n,u)_j = n_j u_j$. This sets all the unobserved values to 0. Then, the gappy inner product can be defined as $(x,y)_n = ((n,x),(n,y))$, which is the inner product of the each vector masked respectively. The induced norm is then $(||x||_n)^2 = (x,x)_n$. Considering the reduction basis obtained by MESS $V_{\text{gap}} = \{v^i\}_{i=1}^r$, now we

can construct an intermediate "repaired" full size vector $\tilde{g}$ from a reduced vector g with only r elements by

$$\tilde{g} \approx \sum_{i=1}^{r} b_i v^i,$$

where the coefficients b_i need to minimise an error E between the original and repaired vector, which is defined as

$$E = ||g - \tilde{g}||_n^2,$$

using the gappy norm so that only the original existing data elements in g are compared. This minimisation is done by solving the linear system

$$Mb = f,$$

where

$$M_{ij} = (v^i, v^j)_n, \text{ and } f_i = (g, v^i)_n.$$

From this solution $\tilde{g}$ is constructed. Then the complete vector is reconstructed by mapping the reduced vectors elements to their original indices and filling the rest with the reconstructed values.

4.3.3. The Reduced System

To incorporate the previous two sections into the partitioned DAE system (4.1a)-(4.1d), we first rewrite (4.1c)-(4.1d) in a more general DAE form, to have the slow subsystem encapsulated into one equation.

$$\begin{aligned}
\frac{d}{dt} y_{\mathrm{F}} &= f_{\mathrm{F}}(t, y_{\mathrm{F}}, z_{\mathrm{F}}, u_{\mathrm{S}}), \quad y_{\mathrm{F}}(t_0) = y_{\mathrm{F}_0}, \\
0 &= g_{\mathrm{F}}(t, y_{\mathrm{F}}, z_{\mathrm{F}}, u_{\mathrm{S}}), \quad z_{\mathrm{F}}(t_0) = z_{\mathrm{F}_0}, \\
\frac{d}{dt}\phi(u_{\mathrm{S}}) &= F_{\mathrm{S}}(t, y_{\mathrm{F}}, z_{\mathrm{F}}, u_{\mathrm{S}}), \quad u_{\mathrm{S}}(t_0) = (y_{\mathrm{S}_0}, z_{\mathrm{S}_0})^\top,
\end{aligned}$$

where $F_{\mathrm{S}} : \mathbb{R} \times \mathbb{R}^a \times \mathbb{R}^b \times \mathbb{R}^{m_{\mathrm{S}}} \to \mathbb{R}^{m_{\mathrm{S}}}$ and $u_{\mathrm{S}} = (y_{\mathrm{S}}, z_{\mathrm{S}})^\top$. Into these equations we incorporate the back projected reduced state $\tilde{u}_{\mathrm{S_r}} = u_{\mathrm{S}}^0 + V_u u_{\mathrm{S_r}}$

$$\begin{aligned}
\frac{d}{dt} y_{\mathrm{F_r}} &= f_{\mathrm{F}}(t, y_{\mathrm{F_r}}, z_{\mathrm{F_r}}, \tilde{u}_{\mathrm{S_r}}), \\
0 &= g_{\mathrm{F}}(t, y_{\mathrm{F_r}}, z_{\mathrm{F_r}}, \tilde{u}_{\mathrm{S_r}}), \\
\frac{d}{dt}\phi(\tilde{u}_{\mathrm{S_r}}) &= F_{\mathrm{S}}(t, y_{\mathrm{F_r}}, z_{\mathrm{F_r}}, \tilde{u}_{\mathrm{S_r}}).
\end{aligned}$$

and then, with the Gappy MESS complexity reduction incorporated we obtain

$$\begin{aligned}
\frac{d}{dt} y_{\mathrm{F_r}} &= f_{\mathrm{F}}(t, y_{\mathrm{F_r}}, z_{\mathrm{F_r}}, \tilde{u}_{\mathrm{S_r}}), \\
0 &= g_{\mathrm{F}}(t, y_{\mathrm{F_r}}, z_{\mathrm{F_r}}, \tilde{u}_{\mathrm{S_r}}), \\
\frac{d}{dt}\phi(\tilde{u}_{\mathrm{S_r}}) &= F_{\mathrm{S_r}}(t, y_{\mathrm{S_r}}, z_{\mathrm{F_r}}, \tilde{u}_{\mathrm{S_r}}).
\end{aligned}$$

Where $F_{\mathrm{S_r}}$ denotes the function F_{S} solved by the reduced least squares approach. Note that the subscript r denotes a reduction, and not the reduction factor.

4.4. Numerical integration

Since the set of equations used to describe the electrical circuits is constructed according to the topological structure of the network. This often results in a coupled system of implicit differential and nonlinear equations,

or more general a system of differential-algebraic equations (DAEs)

$$f(\dot{x}, x, t) = 0 \text{ with } \det \frac{\partial f}{\partial \dot{x}} \equiv 0.$$

This system may represent ill-posed problems and is in general more difficult to solve numerically than the more standard systems of ordinary differential equations (ODEs).

4.4.1. Backward Differentiation Formula

Starting from the consistent initial values, the time domain is discretised into time points $t_0, t_1, ..., t_N$, and the solution for each of these time points is approximated by an implicit linear numerical integration formula. A direct approach, as proposed in [39], is by applying backward differentiation formula (BDF) method. This multistep method is applied to a DAE system by using the ϵ-embedding method. Consider a semi-explicit system with dynamical variables y and algebraic variables z,

$$\begin{aligned} \dot{y} &= f(y, z), \\ \epsilon \dot{z} &= g(y, z). \end{aligned}$$

then the multistep method gives

$$\begin{aligned} \sum_{i=0}^{k} \alpha_i y_{n+i} &= h \sum_{i=0}^{k} \beta_i f(y_{n+i}, z_{n+i}), \\ \epsilon \sum_{i=0}^{k} \alpha_i z_{n+i} &= h \sum_{i=0}^{k} \beta_i g(y_{n+i}, z_{n+i}). \end{aligned}$$

Then by putting $\epsilon = 0$ we obtain

$$\begin{aligned} \sum_{i=0}^{k} \alpha_i y_{n+i} &= h \sum_{i=0}^{k} \beta_i f(y_{n+i}, z_{n+i}), \\ 0 &= \sum_{i=0}^{k} \beta_i g(y_{n+i}, z_{n+i}). \end{aligned}$$

which enables us to apply this method to a semi-explicit differential algebraic system. However, we want to solve system, which can be an implicit differential algebraic system. Therefore, the multistep system for an implicit DAE system, $M\dot{x} = f(x)$, is given by

$$M \sum_{i=0}^{k} \alpha_i x_{n+i} = h \sum_{i=0}^{k} \beta_i f(x_{n+i}) \tag{4.9}$$

In general form, applying Equation (4.9) to an implicit nonlinear system of DAEs at time step t_n yields

$$f(\frac{1}{h} \sum_{i=0}^{k} \frac{\alpha_i}{\beta_i} x_{n-i}, x_n, t_n) = 0.$$

This gives that the numerical solution of the system is thus reduced to the solution of the system of nonlinear Equations 4.4.1. This system is solved iteratively for x_n by applying Newton's method.

4.4.2. Multirate time-integration

The previously seen integration method is considered a singlerate time-integration method, as it integrates each part of the equation with the same step-size. Opposed to this classical approach there are the multirate time-integration methods, which use a different step-size, or even integration method, for parts of the equations with different dynamical behaviour.

To apply multirate time-integration methods to DAEs it is required that all the subsystems in itself are stable, which means that the DAE-index should be less than or equal to that of the full DAE. As we are partitioning network equations of circuit models, it is known that they are composed of subcircuits or natural phenomena in a hierarchical way. If we partition based on this hierarchy, we can check easily if the subsystems fulfil the index requirements and obtain viable partitions.

As there are several different approaches available to implement multirate time-integration methods, we specify in this subsection which approach is used. As has been shown in [32] and [31] a feasible method is given by the *Coupled-Slowest-First* integration approach coupled with the implicit Euler method specified by a first order BDF method. First, the full system is integrated one macro step, then by using interpolated values for the slow system between t_n and t_{n+1}, the fast subsystem is integrated. This approach is chosen due to its stability using constant interpolation on $x_{\mathrm{S},n+1}$, and ease of implementation [40].

4.5. Verification and benchmark

The verification of the reduced order multirate method, utilising the discussed methods, and the netlist parser to create the network equations is done through numerical experiments. For experiment an academic circuit consisting of resistors, capacitors and diodes is considered. To benchmark the implementation of the reduced order multirate method it is compared against the original BDF method and a multirate BDF method.

4.5.1. Academic experiment

The academic circuit shown in Figure 4.1 is a combination of a short diode chain and then a long ladder of diodes and resistors. Similar to a standard diode chain model [41], this model contains sufficient redundancy to make it eligible for model order reduction. Furthermore, increasing the resistance for each R_i with $R < R_i < R_{i+1}$ makes that the ladder part of the circuits behaves on a slower timescale. This makes the circuit excellent for a time integration with the reduced order multirate approach.

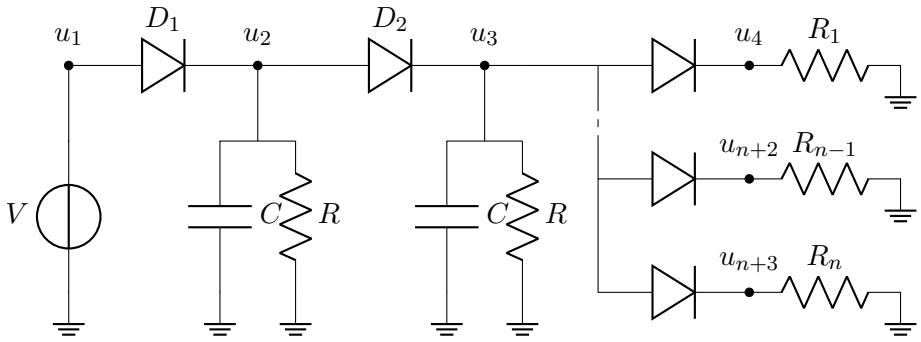


Figure 4.1: The academic diode chain test model with redundancy.

4.6. Implementation and Results

The reduced order multirate simulation of circuits program is implemented using MATLAB R2019b. The main file to run the benchmark is `ROMSOC.m`. This file can be run 'as is' to replicate the results of this paper. The file starts by including the sub-folders in which numerical methods and netlist parsing is encapsulated.

For the reduced order multirate scheme benchmark the file `multirate_example_long_nonlin.cir` is used as netlist file. This file is parsed and then the resulting equations are used for the circuit simulation. Should you desire to simulate other circuits, you can create your own netlist according to the syntax used in the example netlists. Do notice that in the current version the indices for the fast and slow subsystems must be partitioned manually. To create a longer diode chain for the academic diode chain test model, please use the `writeNetlist.m` file and change the length of the second for-loop.

In the main file, first a reference solution is obtained by an integration with the MATLAB integrator `ode15`. Then this reference solution is used to create the reduced order basis used in the model order reduction techniques. These bases and parameters are stored in the `mor_object` data structure and passed along each integrator.

To benchmark the convergence of the reduced order multirate method the simulation is run for an increasing number of time steps. The other simulation parameters are given in the box below.

Simulation parameters of the academic model

Starting time	t_0	0 s
Ending time	t_N	0.004 s
Number of steps	N	$[100\ 200\ 400\ 800]$
Multirate factor	m	20
Newton tolerance	tol	10^{-8}
Original dimension	n	3000
Reduced dimension	r	2
Hyper-reduction factor	g	23
Voltage source	V	$5\sin(40 \cdot 2\pi t)$ V
Resistance	R	$1000\ \Omega$
Resistance	R_i	$i \cdot 1000\ \Omega$
Capacitance	C	$10\ \mu$F
Diode saturation current	I_S	10^{-12} A

Regarding the convergence of the reduced order multirate integration scheme, the left figure in Figure 4.2 illustrates the order 1 convergence rate. We see that the reduced order multirate accuracy is nearly identical to that of the full order solutions. Furthermore, in the right figure of Figure 4.2 it shows that this accuracy is achieved with a significant reduction in computational time. The computational effort is almost a order of magnitude lower for the reduced schemes, while the precision is maintained. The positive effects of model order reduction, multirate time integration and the combination of both is evident.

4.7. Conclusion

A clear definition of a benchmark case for the reduced order multirate method is presented. Along with the source code distributed with this document a circuit simulation incorporating this new method can be run, and the result are easily verified. The combination of multirate time integration and model order reduction into one integration scheme shows a clear advantage for circuits with large redundancy. Higher order schemes are under development and will be incorporated in the final deliverable.

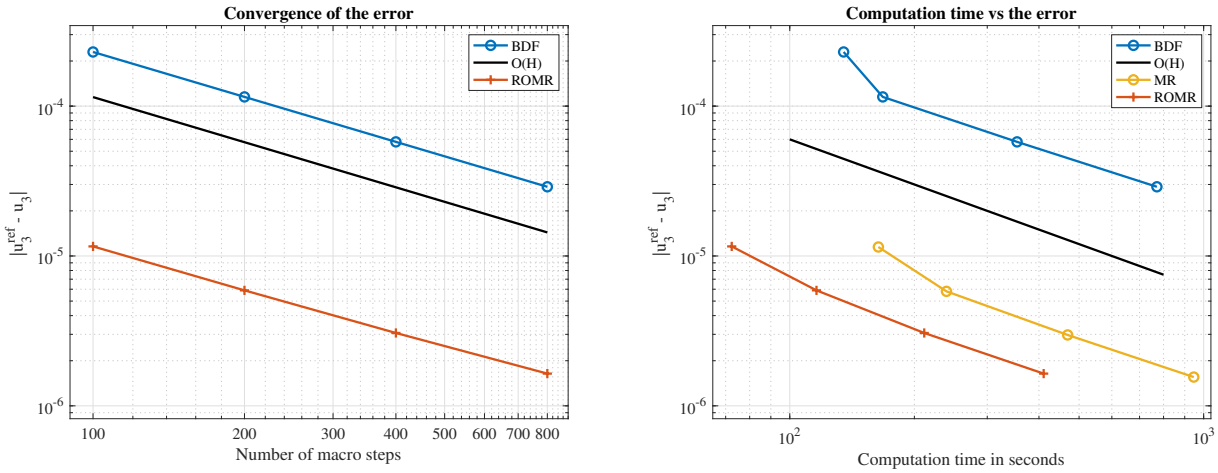


Figure 4.2: Convergence of the numerical schemes, where the error is plotted against the number of macro steps (left). Computational effort of the numerical schemes, where the error is plotted against the computation time in seconds (right). The error is defined as the absolute value between the computed voltage and reference voltage for the output node. The MR error is omitted in the left figure as the difference introduced by the reduction is negligible.

Part V.
Model order reduction for parametric high dimensional models in the analysis of financial risk

Andreas Binder, Onkar Jadhav, Volker Mehrmann

Abstract

It is essential to be aware of the financial risk associated with an invested product. The risk analysis of financial instruments often requires the valuation of such instruments under a wide range of future market scenarios. The market scenarios (e.g., interest rates) are then input parameters in a valuation function that delivers the fair value of such financial instruments. These models are calibrated based on market scenarios that generate a high-dimensional parameter space. In short, to perform the risk analysis, the financial model needs to be solved for such a high dimensional parameter space, and this requires efficient algorithms. The first benchmark case presents the yield curve simulation with parameter calibration. On the other hand, the second case gives the model order reduction approach based on the proper orthogonal decomposition approach with greedy sampling approaches for parameter sampling.

Keywords: Model order reduction, proper orthogonal decomposition, adaptive greedy sampling, financial risk analysis, yield curve simulation

Latest release: `https://doi.org/10.5281/zenodo.5171809`

GitHub repository: `https://github.com/ROMSOC/benchmarks-MOR-finance`

5.1. Introduction

It is essential to be aware of the financial risk associated with an invested product. The risk analysis of financial instruments often requires the valuation of such instruments under a wide range of future market scenarios. The market scenarios (e.g., interest rates) are then input parameters in a valuation function that delivers the fair value of such financial instruments. These models are calibrated based on market scenarios that generate a high-dimensional parameter space. In short, to perform the risk analysis, the financial model needs to be solved for such a high dimensional parameter space, and this requires efficient algorithms. These two benchmark cases present the model order reduction approach based on the proper orthogonal decomposition approach with greedy sampling approaches for parameter sampling.

The first case generates the 10000 simulated yield curves, which are then used to calibrate the financial model parameters. The second case presents both the classical and adaptive greedy sampling approaches. The source code for the yield curve simulation is given the file `YieldCurveSimulation.m`. The parameter calibration based on these yield curves is given in file `1FHWcalibration.nb`. The work provides a method to perform such a computationally costly task as fast as possible but with a reliable outcome based on a model order reduction approach.

5.2. Parametric Model Order Reduction Approach

We employ the projection based MOR technique to solve the full model (FM)

$$A(\rho(t))V^{n+1} = B(\rho(t))V^n, \qquad V(0) = V_0,$$

where the matrices $A(\rho) \in \mathbb{R}^{M\times M}$, and $B(\rho) \in \mathbb{R}^{M\times M}$ are parameter dependent matrices. $V \in \mathbb{R}^M$ is a high dimensional state vector. t is the time variable $t = [0, T]$. ρ is a group of model parameters. We need to solve the system (5.2) for at least 10,000 parameter groups ρ generating a parameter space $\mathcal{P}$ of $10000 \times m$, where m is total number of tenor points of the yield curve. The idea is to project a high dimensional space onto a low

dimensional subspace, Q as

$$\bar{V}^n = QV_d^n,$$

where $Q \in \mathbb{R}^{M\times d}$ is a reduced order basis with $d \ll M$, V_d is a vector of reduced coordinates, and $\bar{V} \in \mathbb{R}^M$ is the solution obtained using the reduced order model. We get the reduced model as

$$\begin{aligned} Q^T A(\rho) Q V_d^{n+1} &= Q^T B(\rho) Q V_d^n, \\ A_d(\rho) V_d^{n+1} &= B_d(\rho) V_d^n, \end{aligned}$$

where the matrices $A_d(\rho) \in \mathbb{R}^{d\times d}$ and $B_d(\rho) \in \mathbb{R}^{d\times d}$ are the parameter dependent reduced matrices. We now solve this reduced model for the entire parameter space $\mathcal{P}$.

We obtain the parametric reduced order model (5.2) based on a proper orthogonal decomposition method (POD) [42, 43]. POD generates an optimal order orthonormal basis Q which serves as a low dimensional subspace in the least square sense for a given set of computational data. The selection of a data set plays an important role, and most prominently obtained by the *method of snapshots* [44]. In this method, the optimal basis is computed based on a set of state solutions. These state solutions are known as snapshots and are calculated by solving the HDM for some parameter values. The quality of the parametric reduced model mainly depends on the selection of training parameters $\rho_1, \rho_2, \ldots, \rho_k$ for which the snapshots are computed. Thus, it necessitates defining an efficient sampling technique for the high dimensional parameter space. We present a classical as well as adaptive greedy sampling approach to select these training parameters. The greedy sampling method introduced in [45] is proven to be an efficient method for sampling a high dimensional parameter space in the framework of MOR for financial risk analysis.

5.2.1. Greedy Sampling Techniques

Greedy algorithms were first introduced as optimization techniques [46]. Their popularity increased in many fields because, even though they do not necessarily find a globally optimal solution, they succeed in obtaining local optima in a relatively short time. In the model order reduction, the greedy algorithm iteratively selects the best possible parameter groups and constructs the reduced basis. The idea is to select the parameter groups at which the error between the reduced model and the full model is maximal. At each greedy iteration, the method seeks a parameter group where the reduced model solution is the worst for the existent reduced basis. Thus, adding the full model solution for the worst parameter group to the snapshot matrix will ultimately improve the quality of the reduced basis for the next iteration. Let ϵ_{RM} be the relative error between the full model and the reduced model given as $\|\epsilon_{\mathrm{RM}}(.,\rho)\| = \|V(.,\rho) - \bar{V}(.,\rho)\|/\|V(.,\rho)\|$. Since the computation of the relative error requires the full model solution, the process becomes extremely computationally costly. Therefore, the greedy sampling algorithm is generally trained on some randomly chosen pre-defined parameter subset $\hat{\mathcal{P}} \subset \mathcal{P}$, instead of the entire parameter space $\mathcal{P}$. Also, the relative error is replaced by the error bounds or the relative residual for the approximate solution $\bar{V}$.

For a high dimensional parameter space, it is not feasible to compute the error estimate for each parameter vector. Thus, we run the greedy sampling algorithm for a pre-defined set of parameter vectors $\hat{\mathcal{P}} \subseteq \mathcal{P}$. The selection of this subset could be random. However, the random selection of a parameter set may not contain the parameter vector corresponding to the most significant error. Therefore, instead of selecting $\hat{\mathcal{P}}$ randomly, we propose to select it adaptively. We construct a surrogate model $\bar{\varepsilon}$ to approximate the error estimator ε over the entire parameter space. Further, we use the surrogate model to locate the parameter vectors $\hat{\mathcal{P}}$. The detailed developed adaptive greedy approach is presented in [45].

5.3. Benchmark Case 1

The first proposed benchmark case is to validate the numerical methods implemented for the simulation of yield curves and parameter calibration. The implemented numerical methods are following the guidelines provided by the PRIIP regulations [47, 48].

We perform a principal component analysis on the collected historical data to ensure that the simulation results in a consistent curve. Further, using the principal components corresponding to their maximum energies, we calculate the consistent interest rates and composed them into a matrix called as the matrix of returns. Finally, we obtain the simulated yield curve by applying the bootstrapping procedure on the matrix of returns. To fulfill the regulations demand, we perform the bootstrapping process for at least 10,000 times. The detailed procedure can be found in the PRIIPs regulations. In this work, we implement the parameter calibration as described in [49]. We use the inbuilt UnRisk functions for the parameter calibration. UnRisk PRICING ENGINE integrates the pricing and calibration engines into Mathematica.

5.3.1. Description of Input Data

To perform yield curve simulation, we collect historical interest rate data. We construct a data matrix $A \in \mathbb{R}^{n\times m}$ of the collected historical interest rates data, where each row of the matrix forms a yield curve, and the column represents the m tenor points, which are the different contract lengths of an underlying instrument. For example, we have collected the daily interest rate data at $m \approx 20$ tenor points in time over the past five years, and since a year has approximately 260 working days, we obtain $n \approx 1306$ observation periods. This data matrix is read as A in the MATLAB file `YieldCurveSimulation.m`.

5.3.2. Step-by-Step Procedure

5.3.2.1. Yield Curve Simulation

The yield curve simulation procedure is well described in [45]. The regulations demand to take the natural logarithm of the ratio between the interest rate at each observation period and the interest rate at the preceding period. To ensure that we can form the natural logarithm, we need that all elements of the data matrix A are positive which is achieved by adding a correction term a as shown in the `.m` file `corr_A1 = A+a`. Then we calculate the log returns over each period and store them into a new matrix $\hat{A} = \hat{A}_{ij} \in \mathbb{R}^{n\times m}$ as

$$\hat{A}_{ij} = \frac{\ln(\bar{A}_{ij})}{\ln(\bar{A}_{i-1,j})}.$$

We calculate the arithmetic mean μ_j of each column of the matrix $\hat{A}$,

$$\mu_j = \frac{1}{n}\sum_{i=1}^{n} \hat{A}_{ij},$$

subtract μ_j from each element of the corresponding jth column of $\hat{A}$ and store the obtained results in a matrix $\bar{\bar{A}}$ with entries $\bar{\bar{A}}_{ij} = \hat{A}_{ij} - \mu_j$. This corrected returns are stored in the variable `BarBarA` in `.m` file. We then compute the singular value decomposition of the matrix $\bar{\bar{A}}$

```
[Vr, Fig1, Fig2] = PCA_YC(BarBarA, p);

AVr = BarBarA*Vr;

%% Matrix of Returns
VrT = transpose(Vr);
Matreturns = AVr * VrT;
```

to generate the matrix of returns stored in the variable `Matreturns`. The selection of singular vectors is based on their energy levels. To do so, we plot the singular values of the data matrix A. MATLAB `Figure(1)` and `Figure(2)` show these plots of monotonously decreasing singular values. We then select the first p right

singular vectors corresponding to the p largest singular values. The regulations suggest selecting the first three singular vectors.

We then perform *bootstrapping*, where large numbers of small samples of the same size are drawn repeatedly from the original data set. According to the PRIIP regulations, for the yield curve simulation we have to perform a bootstrapping procedure for at least 10 000 times. The standardized KID also has to include the recommended *holding period*, i.e., the period between the acquisition of an asset and its sale. The time step in the simulation of yield curves is typically one observation period. If H is the recommended holding period in days, e.g., $H \approx 2600$ days, then there are H observation periods in the recommended holding period. The source code for the bootstrapping simulation is addressed in the subsection `%% Simulations` in the `YieldCurveSimulation.m` file. For each such observation period, we select a random row from the matrix of returns M_R, i.e., altogether H random rows, and construct a matrix $[\chi_{ij}] \in \mathbb{R}^{H\times m}$ from these selected rows. In our benchmark example $H = 10$ years ≈ 2600 days. Then we sum over the selected rows of the columns corresponding to the tenor point j, i.e.,

$$\bar{\chi}_j = \sum_{i=1}^{h} \chi_{ij},\ j = 1, \cdots, m.$$

In this way, we obtain a row vector $\bar{\chi} = [\bar{\chi}_1\ \bar{\chi}_2\ \cdots\ \bar{\chi}_m] \in \mathbb{R}^{1\times m}$. The final simulated yield rate y_j at tenor point j is then the rate $\bar{d}_{nj}$ of the last observation period at the corresponding tenor point j, multiplied by the exponential of $\bar{\chi}_j$, adjusted for any shift γ used to ensure positive values for all tenor points, and adjusted for the forward rate so that the expected mean matches current expectations.

The forward rate between time points t_k and t_ℓ starting from a time point t_0 is given as

$$r_{k,\ell} = \frac{R(t_0,t_\ell)(t_\ell - t_0) - R(t_0,t_k)(t_k - t_0)}{t_\ell - t_k},$$

where t_k and t_ℓ are measured in years and $R(t_0,t_k)$ and $R(t_0,t_\ell)$ are the interest rates available from the data matrix for the time periods (t_0,t_k) and (t_0,t_ℓ), respectively. The forward rate calculation is presented in the section `%% Forward Rates` in the `.m` file. Thus, the final simulated yield curve between time points t_k and t_ℓ is given by

$$y(t_\ell) = \bar{d}_{k,\ell}\exp(\bar{\chi}_\ell) - \gamma + r_{k,l},\ \ell = 1, \cdots, m,$$

and the simulated yield curve from the calculated simulated returns is given by

$$y = [y_1\ y_2\ \cdots\ y_m].$$

We then perform the bootstrapping procedure for at least $s = 10\,000$ times and construct a simulated yield curve matrix

$$Y = \begin{bmatrix} y_{11} & \cdots & y_{1m} \\ \vdots & \vdots & \vdots \\ y_{s1} & \cdots & y_{sm} \end{bmatrix} \in \mathbb{R}^{s\times m}.$$

The simulated yield curves are stored in the variable `Sim_return` and in percentage form `Sim_returnPerc`, which are then saved into excel file `SimulatedYieldCurves.xlsx` for the parameter calibration. These simulated yield curves can be plotted using the `plot` function of the MATLAB.

```
%% Plot output for simulated yield curves.
tic
figure(3)
X = [1 5 10 15 20 25 30 40 50]; % xticks
```

```
T1 = TenorPoints;                % tenor points of yield curves
plot(T1,Sim_returnPerc')
set(gca,'FontSize',14)
set(gcf, 'Position',  [100, 100, 800, 500])
ax.FontSize = 14;
xticks(X);
xlabel('Terms in years','fontsize',18,'interpreter','latex');
ylabel('Simulated yield curves','fontsize',18,'interpreter','latex');
grid on
toc
```

5.3.2.2. Calibration of the Parameter $a(t)$

For a zero coupon bond $B(t,T)$ maturing at time T, based on the Hull-White model, one obtains a closed-form solution, see [49], as

$$B(t,T) = \exp\{-r(t)\Gamma(t,T) - \Lambda(t,T)\},$$

where $\kappa(t) = \int_0^t b(s)ds = bt$, since b is assumed constant,

$$\Gamma(t,T) = \int_t^T e^{-\kappa(t)}dt,$$
$$\Lambda(t,T) = \int_t^T \left[e^{\kappa(v)}a(v)\left(\int_v^T e^{-\kappa(z)}dz\right) - \frac{1}{2}e^{2\kappa(v)}\sigma^2\left(\int_v^T e^{-\kappa(z)}dz\right)^2\right]dv.$$

Here we have again considered that σ is constant.

To perform the calibration, we use as input data i) the initial value of $a(0)$ at $t = 0$, ii) the zero-coupon bond prices, iii) the constant value of the volatility σ of the short-rate $r(t)$, and iv) the constant value b each for all maturities T_m, $0 \leq T_m \leq T$, where T_m is the maturity at the mth tenor point. Then we compute $\kappa(t)$ from $\frac{\partial}{\partial T}\kappa(T) = \frac{\partial}{\partial T}\int_0^T b(s)ds = b$ and use

$$\frac{\partial}{\partial T}\Gamma(0,T) = e^{-\kappa(T)}$$

to compute $\Gamma(t)$.

Then, for $0 \leq T_m \leq T$, we get

$$\begin{aligned}
\frac{\partial}{\partial T}\Lambda(0,T) &= \int_0^T \Big[e^{\kappa(v)} a(v) e^{-\kappa(T)} \\
&\quad - e^{2\kappa(v)}\sigma^2 e^{-\kappa(T)} \Big(\int_v^T e^{-\kappa(z)} dz \Big) \Big] dv, \\
e^{\kappa(T)} \frac{\partial}{\partial T}\Lambda(0,T) &= \int_0^T \Big[e^{\kappa(v)} a(v) - e^{2\kappa(v)}\sigma^2 \Big(\int_v^T e^{-\kappa(z)} dz \Big) \Big] dv, \\
\frac{\partial}{\partial T}\Big[e^{\kappa(T)} \frac{\partial}{\partial T}\Lambda(0,T) \Big] &= e^{\kappa(T)} a(T) - \int_0^T e^{2\kappa(v)}\sigma^2 e^{-\kappa(T)} dv, \\
e^{\kappa(T)}\Big[e^{\kappa(T)} \frac{\partial}{\partial T}\Lambda(0,T) \Big] &= e^{2\kappa(T)} a(T) - \int_0^T e^{2\kappa(v)}\sigma^2 dv, \\
\frac{\partial}{\partial T}\Big[e^{\kappa(T)}\Big[e^{\kappa(T)} \frac{\partial}{\partial T}\Lambda(0,T) \Big]\Big] &= \frac{\partial a(T)}{\partial T} e^{2\kappa(T)} + 2a(T) e^{2\kappa(T)} \frac{\partial}{\partial T}\kappa(T) - e^{2\kappa(T)}\sigma^2, \\
\frac{\partial}{\partial T}\Big[e^{\kappa(T)}\Big[e^{\kappa(T)} \frac{\partial}{\partial T}\Lambda(0,T) \Big]\Big] &= \frac{\partial a(T)}{\partial T} e^{2\kappa(T)} + 2a(T) e^{2\kappa(T)} b(T) - e^{2\kappa(T)}\sigma^2.
\end{aligned}$$

The simulated yield $y(T)$ at the tenor point T is given by [50]

$$y(T) = -\ln B(0,T),$$

and from (5.3.2.2) and (5.3.2.2) we obtain $\Lambda(0,T) = [y(T) - r(0)\Gamma(t)]$. In this way, for $a(t)$ we get the ordinary differential equation (ODE)

$$\frac{\partial}{\partial t} a(t) e^{2\kappa(t)} + 2a(t) \cdot b \cdot e^{2\kappa(t)} - e^{2\kappa(t)}\sigma^2 = \frac{\partial}{\partial t}\Big[e^{\kappa(t)}\Big[e^{\kappa(t)} \frac{\partial}{\partial t}(y(t) - r(0)\Gamma(0,t)) \Big]\Big],$$

which we solve numerically with the given initial conditions. If we approximate $a(t)$ by a piecewise constant function with values $a(i)$ which change at the tenor point i, then we obtain a linear system

$$L\alpha = F,$$

for the vector $\alpha = [a(i)]$, where L is lower triangular matrix with non-zero diagonal elements. In [51] it is noted that the integral equation Λ is of the first kind with L2 kernel and a small perturbation (noise) in the market data that are used to obtain the yield curves leads to large changes in the model parameter $a(t)$. This means that the problem to compute $a(t)$ from the data is an ill-posed problem and for this reason we determine the vector α via Tikhonov regularization as

$$\alpha_\mu^\delta = \text{argmin} \|L\alpha - F^\delta\|^2 + \mu\|\alpha\|^2,$$

where α_μ^δ is an approximation to α, μ is the regularization parameter, $\delta = \|F - F^\delta\|$ is the noise level, and $\mu\|\alpha\|^2$ is a regularization term. We then solve the optimization problem (5.3.2.2) to obtain an approximation of the parameter $a(t)$ via the commercial software *UnRisk PRICING ENGINE* [52] in Mathematica. The source code for the calibration of the model parameter $a(t)$ is in the file `1FHWcalibration.nb`. This program takes the simulated yield curves stored in the file `SimulatedYieldCurves.xlsx` as an input. The parameters $b = 0.015$ and $\sigma = 0.006$ are kept constant. Note that this file demands a special license to the commercial software UnRisk Pricing Engine, which is initiated by `Needs["UnRisk UnRiskFrontEnd"]`. The program

returns the 10 000 deterministic drift rates $a(t)$ for 10 000 simulated yield curves. For the floater example, we need parameter values only until the 10Y tenor point (maturity of the floater). Henceforth, we consider the simulated yield curves with only the first 12 tenor points `t = {0, 65, 260, 520, 780, 1040, 1300, 1560, 1820, 2080, 2340, 2600}`. The parameter $a(t)$ is stored in the variable `DDF1FL` after calibrating the Hull-White model using the command `MakeGeneralHullWhiteModel`. Then we export these piecewise constant parameters into a file using the command `Export["DD_1FHWModel_10000.xlsx", DDF1FL]`. By providing the simulated yield curve, the UnRisk pricing function returns the calibrated parameter $a(t)$ for that yield curve. Based on $s = 10\,000$ different simulated yield curves, we obtain s different piecewise constant parameters $a_\ell(t)$, which change their values $\alpha_{\ell,i}$ only at the m tenor points. We incorporate these in a matrix

$$\mathcal{A} = \begin{bmatrix} \alpha_{11} & \cdots & \alpha_{1m} \\ \vdots & \vdots & \vdots \\ \alpha_{s1} & \cdots & \alpha_{sm} \end{bmatrix}.$$

5.4. Benchmark Case 2

The main task of this research is to implement a parametric model order reduction approach for financial risk analysis. Second benchmark case is to verify the implemented MOR algorithm. We use a finite difference method for simulating the convection-diffusion-reaction PDE. The projection-based MOR approach has been implemented, and the reduced-order basis is obtained using the proper orthogonal decomposition approach with the classical and adaptive greedy sampling methods.

The benchmark case addresses both the algorithms and presents MATLAB code for the same.

5.4.1. Description of Input Data

The classical greedy sampling and the adaptive greedy sampling have been used to locate the training parameters required to generate the reduced basis. The classical greedy sampling algorithm takes the parameter space $\mathcal{P}$, maximum number of iterations I_{max}, maximum number of parameter groups c, and tolerance ε_{tol} as inputs. The variable `a` defines the matrix composed of parameter vectors $a(t)$ in the section `%% Model Parameters` in `.m` file. `a` takes the calibrated parameters stored in the excel file `DD_1FHWModel_10000.xlsx` obtained in the benchmark case 1. The user can define the number of maximum parameter groups required to initiate the algorithm, the maximum number of iterations, and the tolerance using the variables `c`, `Imax`, `max_tol`, respectively. For example, in our sample code, the values are `c = 20`, `Imax = 10`, `max_tol` $= 10^{-5}$.

The adaptive greedy sampling algorithm also takes the parameter space $\mathcal{P}$, maximum number of iterations I_{max}, maximum number of parameter groups c, and tolerance e_{tol}^{max} as inputs. Along with other inputs, the user needs to specify the number of adaptive candidates to complete the surrogate model loop. The user can define these inputs using the variables `c`, `Imax`, `max_tol`, and `ck` respectively. For example, in our sample code, the values are `c = 20`, `Imax = 10`, `ck = 10`, and `max_tol` $= 10^{-8}$.

5.4.2. Step-by-Step Procedure

5.4.2.1. Classical Greedy Sampling

The algorithm is initiated by selecting a parameter group ρ_1 from the parameter set $\mathcal{P}$ and computing a reduced basis Q_1. The first group of parameter is shown using variables `a(1:1,:), b, sigma`. It is necessary to note that the choice of a parameter group to obtain the initial reduced basis Q_1 does not affect the final result. One can initiate the greedy sampling algorithm with any parameter group. Nonetheless, for the simplicity of computations, we select the first parameter group ρ_1 from the parameter space $\mathcal{P}$ to obtain Q_1.

The greedy iteration are indexed with `Niter` and runs for `Imax` iterations. Furthermore, a pre-defined pa-

rameter set $\hat{\mathcal{P}}$ of cardinality c has been chosen randomly from the set $\mathcal{P}$ and shown with variable `NpSel` in MATLAB file. At each point of $\hat{\mathcal{P}}$, the algorithm determines a reduced model with reduced basis Q_1 and then computes error estimator values, $\varepsilon(\rho_j)_{j=1}^{c}$. The parameter group in $\hat{\mathcal{P}}$ at which the error estimator is maximal is then selected as the optimal parameter group ρ_I.

`max` function is used to locate this parameter group. The error estimator values obtained by solving the reduced models are stored in `Error`. The following command locates the parameter group that maximizes the error estimator

```
[max_Error(1,(Niter-1)),max_idErr(1,(Niter-1))] = max(Error(:))
```

It also gives the location of the parameter group within the pre-defined parameter set `NpSel`. Then the full model is simulated for this parameter group and the snapshot matrix $\hat{V}$ is updated. Finally, a new reduced basis is obtained by computing a truncated singular value decomposition of the updated snapshot matrix. The truncated SVD is based on the randomized algorithm, which is given by the `function RandSVD`

```
[U1,S1,V1] = RandSVD(SnapShots,RankS(1,Niter-1));
```

These steps are then repeated for I_{max} iterations or until the maximal value of the error estimator is lower than the specified tolerance ε_{tol}.

```
if max_Error(1,(Niter-1)) < max_tol
    Q = Q_Niter;
    break
end
```

The truncated SVD computes only the first k columns of the matrix Φ. The optimal projection subspace Q then consists of d left singular vectors ϕ_i known as *POD modes*. The dimension d of the subspace Q is chosen such that we get a good approximation of the snapshot matrix. According to [53], large singular values correspond to the main characteristics of the system, while small singular values give only small perturbations of the overall dynamics. The relative importance of the ith POD mode of the matrix $\hat{V}$ is determined by the *relative energy* Ξ_i of that mode

$$\Xi_i = \frac{\Sigma_i}{\sum_{i=1}^{k} \Sigma_i}$$

If the sum of the energies of the generated modes is 1, then these modes can be used to reconstruct a snapshot matrix completely [54]. In general, the number of modes required to generate the complete data set is significantly less than the total number of POD modes [55]. Thus, a matrix $\hat{V}$ can be accurately approximated by using POD modes whose corresponding energies sum to almost all of the total energy. Thus, we choose only d out of k POD modes to construct $Q = [\phi_1 \cdots \phi_d]$ which is a parameter independent projection space. `%% Selection of the reduced dimension` describes the procedure to select the first d left singular vectors based on their relative energies that generate the reduced basis `Q_d`.

```
Eg = diag(S1)./sum(diag(S1));
%
for ii = 1:length(Eg)
    SumEg = sum(Eg(1:ii,1))*100;
    if SumEg > 99.99
        d = ii;
        break
    end
end
%
Q_d = Q_Niter(:,1:d);
```

The function `CGPlots` is used to plot the results obtained using the classical greedy sampling approach. The progression of the maximal and average residuals with each iteration of the greedy algorithm is presented in `Figure(1)`. The projection error associated with the reduced basis is calculated using (5.4.2.1) is plotted in `Figure(3)`.

$$\epsilon_{\text{POD}}^{\text{CG}} = \frac{1}{iT}\sum_{i=1}^{iT} \|V_i(\rho_I) - \sum_{iT=1}^{\ell} (V_i(\rho_I)\phi_k)\phi_k\|_2^2 = \sum_{\ell=d+1}^{iT} \Sigma_\ell^2.$$

5.4.2.2. Adaptive Greedy Sampling

To overcome drawbacks of the classical greedy sampling approach, we implement the adaptive sampling approach which selects the parameter groups adaptively at each iteration of the greedy procedure, using an optimized search based on surrogate modeling. The surrogate model constructs an approximate model for the desired simulation output, i.e., in our case, the error estimator. Then the surrogate model is trained on some error estimator values. This training data is obtained by solving the reduced model for some random parameter groups. Subsequently, we can deploy this trained surrogate model to perform simulations instead of the original model. Since the single evaluation of the surrogate model is faster than the original model, performing thousand of evaluations for the given high dimensional parameter space is no longer a problem. In short, the surrogate modeling methods make those expensive computations economical.

There are different choices to build a surrogate model: regression models [56], decision trees [57], machine learning and artificial neural networks [56, 58], and kriging models [59]. In this work, we present two options based on the principal component regression model (PCR) and the K-nearest neighbor (KNN) algorithm. The PCR algorithm is presented in the function `PCRSM`, while the KNN algorithm is in the function `KNN`.

The adaptive greedy sampling algorithm utilizes the designed surrogate model to locate optimal parameter groups adaptively at each greedy iteration $i = 1, ..., I_{max}$. The greedy iteration are indexed with `i` and runs for `Imax` iterations. The first few steps of the algorithm resemble the classical greedy sampling approach. First the parameter group ρ_1 is selected from the parameter space $\mathcal{P}$ and the reduced basis Q_1 is constructed. Furthermore, the algorithm randomly selects c_0 parameter groups (defined using `c0` in `.m` file) and constructs a temporary parameter set $\hat{\mathcal{P}}_0 = \{\rho_1, ..., \rho_{c_0}\}$ and shown with variable `NpSel` in MATLAB file. For each parameter group in the parameter set $\hat{\mathcal{P}}_0$, the algorithm determines a reduced order model and computes an array of corresponding residual errors $\hat{\varepsilon}_0 = \{\varepsilon(\rho_1), ..., \varepsilon(\rho_{c_0})\}$. Then a surrogate model for the error estimator $\bar{\varepsilon}$ is constructed based on the estimator values $\{\varepsilon(\rho_j)\}_{j=1}^{c_0}$ using either `PCRSM` or `KNN`. The user can choose the surrogate model as desired. The obtained surrogate model is then simulated for the entire parameter space $\mathcal{P}$. Furthermore, we locate c_k parameter groups corresponding to the first c_k maximal values of the surrogate model. This selection is made using the maxk function as follows

```
[MAX_SM(jjj,:),MAX_SMid(jjj,:)] = maxk(SM,ck)
```

which determines the first `ck` values of the surrogate model `SM` and the corresponding parameter groups `a(MAX_SMid',:)`. We then construct a new parameter set $\hat{\mathcal{P}}_k = \{\rho_1, ..., \rho_k\}$ composed of these c_k parameter groups as

```
NpSel2 = cat(1,NpSel,a(MAX_SMid',:)).
```

The algorithm determines a reduced model for each parameter group within the parameter set $\hat{\mathcal{P}}_k$ and obtains an array of error estimator values $\hat{\varepsilon}_k = \{\varepsilon(\rho_1), ..., \varepsilon(\rho_{c_k})\}$. Furthermore, we concatenate the set $\hat{\mathcal{P}}_k$ and the set $\hat{\mathcal{P}}_0$ to form a new parameter set $\hat{\mathcal{P}} = \hat{\mathcal{P}}_k \cup \hat{\mathcal{P}}_0$. Let $e_{\text{sg}} = \{\hat{\varepsilon}_0 \cup \cdots \cup \hat{\varepsilon}_k\}$ be the set composed of all the error estimator values available at the kth iteration. The algorithm then uses this error estimator set e_{sg} to build a new surrogate model. The quality of the surrogate model increases with each iteration as we get more error estimator values. This process is repeated until the cardinality of the set $\hat{\mathcal{P}}$ reaches c, giving

$$\hat{\mathcal{P}} = \hat{\mathcal{P}}_0 \cup \hat{\mathcal{P}}_1 \cup \hat{\mathcal{P}}_2 \cup \cdots \cup \hat{\mathcal{P}}_K.$$

Finally, the optimal parameter group ρ_I which maximizes the error estimator is extracted from the parameter set $\hat{\mathcal{P}}$. The following command locates the parameter group that maximizes the error estimator

```
[max_Error(1,(Niter-1)),max_idErr(1,(Niter-1))] = max(Error2(:))
```

It also gives the location of the parameter group with the parameter set `NpSel2`.

5.4.2.2.1. Convergence of the Adaptive Greedy Sampling Algorithm In the classical greedy sampling approach, we used the residual error to observe the convergence of the algorithm, which estimates the relative error between the full model and the reduced model. However, in the adaptive greedy POD algorithm, we use an approximate model $\hat{\epsilon}_{\mathrm{RM}}$ (`RE_appx`) for the relative error ϵ_{RM} (`RE`) as a function of the residual error ε to monitor the convergence. This is more accurate than using only the residual error as a convergence criterion.

At each greedy iteration, the algorithm solves one full model for the optimal parameter group ρ_I to update the snapshot matrix and construct a new reduced basis. We can utilize this information for the construction of an approximate relative error model. We solve two reduced models for the optimal parameter group, one before and another after updating the reduced basis (Q_{bef}, Q_{aft}), and obtain respective error estimator values $\varepsilon^{bef}(\rho_I)$ and $\varepsilon^{aft}(\rho_I)$ (`resd_b, resd_a`). Then we calculate the relative errors $\epsilon^{bef}_{\mathrm{RM}}$, $\epsilon^{aft}_{\mathrm{RM}}$ (`RE_b, RE_a`) between the full model and the two reduced models constructed before and after updating the reduced basis. Here superscript bef and aft denote the before and after updating the reduced basis. In this way, we get a set of error values E_p at each greedy iteration that we can use to construct an approximate model for the relative error based on the error estimator.

$$E_p = \{(\epsilon^{bef}_{\mathrm{RM},1}, \varepsilon^{bef}_1) \cup (\epsilon^{aft}_{\mathrm{RM},1}, \varepsilon^{aft}_1), \ldots, (\epsilon^{bef}_{\mathrm{RM},i}, \varepsilon^{bef}_i) \cup (\epsilon^{aft}_{\mathrm{RM},i}, \varepsilon^{aft}_i)\}$$

The error set is generated as follows in the code

```
RE_U = cat(2,RE_b,RE_a);
resd_U = cat(2,resd_b,resd_a);
Ep = cat(RE_u, resd_U);
```

We then construct an approximate model for the relative error using the `polyfit` command based on the error estimator as

$$\log(\hat{\epsilon}_{\mathrm{RM},i}) = \gamma_i \log(\varepsilon) + \log\tau.$$

Setting $\mathcal{Y} = \log(\hat{\epsilon}_{\mathrm{RM}}), \mathcal{X} = \log(\varepsilon)$ and $\hat{\tau} = \log(\tau)$ we get

$$\mathcal{Y} = \gamma_e \mathcal{X} + \hat{\tau},$$

where γ_e is the slope of the linear model and $\hat{\tau}$ is the intercept with the logarithmic axis $\log(y)$.

Similar to the classical greedy sampling program, `%% Selection of the reduced dimension` describes the procedure to select the first d left singular vectors based on their relative energies that generate the reduced basis `Q_d`.

```
Eg = diag(S1)./sum(diag(S1));
%
for ii = 1:length(Eg)
    SumEg = sum(Eg(1:ii,1))*100;
    if SumEg > 99.99
        d = ii;
        break
    end
```

```
end
%
Q_d = Q_Niter(:,1:d);
```

The function `AGPlots` is used to plot the results obtained using the classical greedy sampling approach. The progression of the maximal and average residuals with each iteration of the greedy algorithm is presented in `Figure(1)`. The projection error associated with the reduced basis is calculated using (5.4.2.1) is plotted in `Figure(3)`.

$$\epsilon_{\text{POD}}^{\text{AG}} = \frac{1}{iT}\sum_{i=1}^{iT} \|V_i(\rho_I) - \sum_{iT=1}^{\ell} (V_i(\rho_I)\phi_k)\phi_k\|_2^2 = \sum_{\ell=d+1}^{iT} \Sigma_\ell^2.$$

After each greedy iteration, we get more data points in the error set E_p, which increases the accuracy of the error model, provided that the linear model assumption is validated. `Figure(4)` illustrate with the obtained results that this linear model assumption is sufficient to achieve an acceptable approximate relative error model. We then use this error model in the adaptive greedy sampling to monitor the convergence of the algorithm.

5.5. Summary

The benchmark cases present the procedure for yield curve simulation along with the developed model order reduction framework for a numerical example of a floater instrument with caps and floors. We solve this instrument using the robust one-factor Hull-White model.

All computations are carried out on a PC with 4 cores and 8 logical processors at 2.90 GHz (Intel i7 7th generation). We used MATLAB R2018a for the yield curve simulations, FDM, and the model reduction. The numerical method for the yield curve simulations is tested with real market based historical data. Market data are available from market data providers like Thomson Reuters, Bloomberg, and several others. We obtained this data from MathConsult within their UnRisk Omega datasets [52]. The daily interest rate data are collected at 21 tenor points in time over the past 5 years, where each year has 260 working days, so there are 1300 observation periods. We have used the inbuilt UnRisk tool for the parameter calibration, which is well integrated with Mathematica (version used: Mathematica 11.3). Further, we used calibrated parameters for the construction of Hull-White models.

Part VI.
Model order reduction for parametric high dimensional interest rate models in the analysis of financial risk

Andreas B´armann, Alexander Martin, Jonasz Staszek

Abstract

The aim of this benchmark is to provide the basis for comparison of various solution algorithms and approaches against a realistic dataset, reflecting the trains to be performed in February 2020, the available locomotives and drivers, as well as information about its compatibility.

Keywords: discrete optimization, joint vehicle and staff assignment, integer model, joint vehicle and crew scheduling.

Latest release: https://doi.org/10.5281/zenodo.5171817

GitHub repository: https://github.com/ROMSOC/benchmarks-mip-rail-scheduling

6.1. Introduction

DB Cargo Polska works with the Chair of Analytics and Mixed Integer Optimization at Friedrich-Alexander-Universität Erlangen-Nürnberg to build a prototype of a new planning tool, which would come up with locomotive schedules and driver rosters simultaneously. As of today, the industry standard is to plan locomotives and drivers separately. The reason for this is the fact that both the planning areas are challenging on their own, and – without computational support – it would be impossible to consider both the areas at the same time. Such fragmented approaches were criticised in the literature already in 1980s' – see for example [60].

The purpose of the presented benchmarks is to present a set of instances, which we derived in collaboration with the industrial partner, as well as the procedures required to come up with complete Binary Programming models for these instances. In this work, we will only introduce the naive formulation of the mentioned binary models. Formulation improvements, as well as the algorithms to efficiently solve these models will be introduced in Deliverables D4.2 and D4.3.

6.1.1. A joint model for locomotive scheduling and driver rostering in rail freight transport

We model the joint locomotive scheduling and driver rostering problem as a combination of a set packing problem with compatibility, conflict and multiple-choice constraints and a multicommodity flow problem. Our objective is to maximize the number of trains performed, i.e. to maximize the number of trains for which both a locomotive and a driver were found. The inputs to the model are: a set T of trains to be performed, a set of locomotives $\mathscr{L}$ and a set of drivers D. To denote the sets of locomotives compatible with a driver $d \in D$ or a train $t \in T^d$, we use $\mathscr{L}^d$ and $\mathscr{L}^t$ respectively. Let D^l and D^t represent a set of drivers compatible with a locomotive $l \in \mathscr{L}$ or a train $t \in T$ respectively. Finally, let T^l and T^d be a set of trains compatible with a locomotive $l \in \mathscr{L}$ or a driver $d \in D$ respectively.

6.1.1.0.1. Sets required for constraints construction We also introduce a number of sets required to build the constraints of the model. Table 2 presents a summary of the sets required for constraint construction. Their exact definitions can be found in the Appendix A.

Name	Description
$T^{B+}_{t,d}$	Trains which cannot be assigned to a driver d if t is his last job in a working day
$T^{35h}_{t,d}$	Trains which cannot be assigned to a driver d if t is his last job before a weekly 35h break
$T^{B-}_{t,d}$	Trains which cannot be assigned to a driver d if t is his first job in a working day
$T^{week}_{w,d}$	Trains which belong to a calculation week w
$T^{sunday}_{w,d}$	Trains which belong to a Sunday falling on a week w
$T^{shift\_beginning}_{t,d}$	Past trains which could have been assigned to driver d if he is assigned to a job t.
$T^{shift\_end}_{t,d}$	Future trains which can be assigned to driver d if he is assigned to a job t.
$T^{time}_{t,d}$	Trains which are feasible for driver d and in time conflict with the train t
$T^{after\_break}_{t,d}$	Trains which could be the first jobs of the next shift after the 11h break following train t
$T^{before\_break}_{t,d}$	Trains which could have been the last job of the previous shift before the 11h break preceding train t
$T^{next\_l}_{t,l}$	Future trains which could be assigned to locomotive l if is assigned to a job t

Table 2: Descriptions of the sets required for constraint construction

6.1.1.1. Multi-commodity flow part of the model

We consider the set $\mathscr{L} = \{l_1, l_2, l_3, \ldots\}$ of locomotives to be modeled as commodities which need to be "pushed" through a directed graph $G = (V, A)$, which could be defined as $V \coloneqq T$ and

$$A \coloneqq \{(t_1, t_2) : t_1 \in T^l \quad \wedge \quad t_2 \in T^{next\_l}_{t,l} \quad \forall\ l \in \mathscr{L}\}.$$

Let us also define Σ and $\Theta \in V$ as the source and sink nodes of the graph G, respectively. Additionally let us assume that we have one item of each commodity. Further, let each arc $a \in A$ have unit capacity i.e. it can host at most one commodity. We also assume the limited compatibility of each arc with regard to commodities – that means we may not be allowed to push every commodity through every arc.

6.1.1.2. Decision variables

In the model, we need to make sure that each train is staffed by exactly one suitable driver and one suitable locomotive. These decisions are modelled with binary variables x^t_d (for drivers) and f^{t_1,t_2}_l (for locomotives).

To comply with the working time requirements, we need to distinguish between the first job in a shift, denoted by a binary variable y^t_d, the last job in a shift before a short (11 hour) break, denoted by a binary variable v^t_d and the last job in a shift before a long (35 hour) break, denoted by a binary variable z^t_d. We also need to know whether a driver has worked on a given Sunday. This is denoted by a binary variable h^w_d.

Finally, for modelling purposes we also need to know which trains t are the first and the last job for drivers in the planning period. We do that with the help of binary variables α^t_d and ω^t_d, which denote that the train t is respectively the first or the last one in the planning period for a driver $d \in D$. All the variables are summarized in Table 3.

Name	Description	Type
$f_l^{u,v}$	Trains u, v are served by a locomotive l	binary
x_d^t	Train t is served by a driver d	binary
y_d^t	Train t is the first job of a driver d in their shift	binary
v_d^t	Train t is the last job of a driver d before a 12h break	binary
z_d^t	Train t is the last job of a driver d before a 35h break	binary
α_d^t	Train t is the first train of driver d in the planning period	binary
ω_d^t	Train t is the last train of driver d in the planning period	binary
h_d^w	Driver d has worked on the Sunday of the week w	binary

Table 3: Summary of decision variables used in the model

6.1.1.3. Model formulation

$$\max \quad \sum_{t\in T}\sum_{d\in D^t} x_d^t \tag{6.1}$$

s.t.

$$x_d^t \leq \sum_{\substack{l\in\mathscr{L}^d\cap\mathscr{L}^t \\ u:(t,u)\in E}} f_l^{t,u} \qquad (\forall t\in T)(\forall d\in D^t) \tag{6.2}$$

$$\sum_{v:(t,v)\in E} f_l^{t,v} \leq \sum_{d\in D^l\cap D^t} x_d^t \qquad (\forall t\in T\setminus\{A\})\ (\forall l\in\mathscr{L}^t) \tag{6.3}$$

$$\sum_{d\in D^t} x_d^t \leq 1 \qquad (\forall t\in T) \tag{6.4}$$

$$\sum_{t\in T^d} \alpha_d^t \leq 1 \qquad (\forall d\in D) \tag{6.5}$$

$$\sum_{t\in T^d} \omega_d^t \leq 1 \qquad (\forall d\in D) \tag{6.6}$$

$$x_d^t + x_d^{t_1} \leq 1 \qquad (\forall d\in D)\ (\forall t\in T^d)\ (\forall t_1\in T_{t,d}^{time}) \tag{6.7}$$

$$y_d^t + x_d^{t_1} \leq 1 \qquad (\forall d\in D)\ (\forall t\in T^d)\ (\forall t_1\in T_{t,d}^{B-}) \tag{6.8}$$

$$v_d^t + x_d^{t_1} \leq 1 \qquad (\forall d\in D)\ (\forall t\in T^d)\ (\forall t_1\in T_{t,d}^{B+}) \tag{6.9}$$

$$z_d^t + x_d^{t_1} \leq 1 \qquad (\forall d\in D)\ (\forall t\in T^d)\ (\forall t_1\in T_{t,d}^{35h}) \tag{6.10}$$

$$v_d^t \leq \omega_d^t + \sum_{t_1\in T_{t,d}^{after\_break}} y_d^{t_1} \qquad (\forall d\in D)\ (\forall t\in T^d) \tag{6.11}$$

$$y_d^t \leq \alpha_d^t + \sum_{t_1\in T_{t,d}^{before\_break}} v_d^{t_1} \qquad (\forall d\in D)\ (\forall t\in T^d) \tag{6.12}$$

$$x_d^t \leq \sum_{t_1\in T_{t,d}^{shift\_beginning}} y_d^{t_1} \qquad (\forall d\in D)\ (\forall t\in T^d) \tag{6.13}$$

$$x_d^t \leq \sum_{t_1 \in T_{t,d}^{shift\_end}} v_d^{t_1} \quad (\forall d \in D)\ (\forall t \in T^d) \tag{6.14}$$

$$v_d^t \leq \sum_{t_1 \in T_{t,d}^{shift\_beginning}} y_d^{t_1} \quad (\forall d \in D)\ (\forall t \in T^d) \tag{6.15}$$

$$x_d^t \leq \sum_{t_1 \in T_{t,d}^{week}} z_d^{t_1} \quad (\forall d \in D)\ (\forall t \in T^d) \tag{6.16}$$

$$\alpha_d^t \leq \sum_{t_1 : t_1 \geq t} \omega_d^{t_1} \quad (\forall d \in D)\ (\forall t \in T^d) \tag{6.17}$$

$$x_d^t \leq h_d^w \quad (\forall d \in D)\ (\forall t \in T_{w,d}^{sunday}) \tag{6.18}$$

$$\sum_{w \in T^d} h_d^w \leq 3 \quad (\forall d \in D) \tag{6.19}$$

$$y_d^t \leq x_d^t \quad (\forall d \in D) \tag{6.20}$$

$$v_d^t \leq x_d^t \quad (\forall d \in D) \tag{6.21}$$

$$\alpha_d^t \leq x_d^t \quad (\forall d \in D) \tag{6.22}$$

$$\omega_d^t \leq x_d^t \quad (\forall d \in D) \tag{6.23}$$

$$\sum_{v \in \delta^{in}(t) \cap T^l} f_l^{v,t} - \sum_{w \in \delta^{out}(t) \cap T^l} f_l^{t,w} = 0 \quad (\forall t \in T)\ (\forall l \in \mathscr{L}^t) \tag{6.24}$$

$$\sum_{l \in \mathscr{L}^t \cap \mathscr{L}^{t_1}} f_l^{t,t_1} \leq 1 \quad (\forall (t,t_1) \in A) \tag{6.25}$$

$$\sum_{l \in \mathscr{L}^t} \sum_{t_0 \in \delta^{in}(t) \cap T^l} f_l^{t_0,t} \leq 1 \quad (\forall t \in T) \tag{6.26}$$

$$\sum_{t : (\Sigma,t) \in E \wedge t \in T^l} f_l^{\Sigma,t} \leq 1 \quad (\forall l \in \mathscr{L}) \tag{6.27}$$

$$\sum_{t_1 \in T^l} f_l^{\Sigma,t_1} - \sum_{t_2 \in T^l} f_l^{t_2,\Theta} = 0 \quad (\forall l \in \mathscr{L}) \tag{6.28}$$

$$x_d^t \in \mathcal{F} \quad (\forall t \in T)\ (\forall d \in D^t) \tag{6.29}$$

$$y_d^t \in \mathcal{F} \quad (\forall t \in T)\ (\forall d \in D^t) \tag{6.30}$$

$$v_d^t \in \mathcal{F} \quad (\forall t \in T)\ (\forall d \in D^t) \tag{6.31}$$

$$z_d^t \in \mathcal{F} \quad (\forall t \in T)\ (\forall d \in D^t) \tag{6.32}$$

$$\alpha_d^t \in \mathcal{F} \quad (\forall t \in T)\ (\forall d \in D^t) \tag{6.33}$$

$$\omega_d^t \in \mathcal{F} \quad (\forall t \in T)\ (\forall d \in D^t) \tag{6.34}$$

$$f_l^{t_1,t_2} \in \mathcal{F} \quad (\forall (t_1,t_2) \in E)\ (\forall l \in \mathscr{L}^{t_1} \cap \mathscr{L}^{t_2}) \tag{6.35}$$

With objective function (6.1), we maximize the number of trains running. Constraints (6.2) and (6.3) make sure that either both a locomotive and a driver or none of them are assigned to the train; they also take care that driver and locomotive are mutually compatible. With (6.4), we ensure that at most one driver is assigned to a train. Constraints (6.5) and (6.6) ensure that there is no more than one schedule per driver in the plan.

Using constraint (6.7), we ensure that no two trains which run simultaneously are assigned to the same driver. With (6.8) and (6.9) we model that the minimal length of a short break amounting to 11 hours is not violated. Additionally, with (6.10) we ensure the integrity of the long, 35-hour break. Using (6.11), we make sure that each last job t in a shift is succeeded by a first job of the next shift, or that the job t is the last one assigned to driver d in the plan. Similarly, with (6.12) we model that each first job t in a shift was predecessed by a last job of the previous shift, or that the job t is the first one assigned to driver d in the plan. Constraints (6.13), (6.14) and (6.15) ensure the integrity of drivers' shifts and model the maximal length of a shift amounting to 12 hours. Using (6.16) we make sure that at least one long break per week is assigned to each driver in every week falling in the planning period. With the help of (6.17) we make sure that the last job of a driver in the plan is the same or a later one as the first job in the plan. Using constraints (6.18) and (6.19) we make sure that a driver works on at most three Sundays in a given planning period. Constraints (6.20), (6.21), (6.22) and (6.23) tie each "indicator" variable to the actual decision variable.

For the locomotive part of the model, (6.24) ensures that a locomotive that serves a train t arrives at its origin station and in the due time, and similarly it later departs from train t's arrival station. Using (6.25) we ensure that at most one locomotive serves each train. With the use of (6.26), we make sure that an appropriate successor and predecessor are chosen for a locomotive $l \in \mathscr{L}$ given it serves a train $t \in T^l$. We use (6.27) to warrant that at most one first train is chosen for each locomotive. Constraint (6.27) ensures that each locomotive has a unique first and a unique last train in a plan. With the help of (6.28) we ensure the integrity of the locomotive schedule. Finally, constraints (6.29) to (6.35) ensure that the decision variables are binary.

6.2. Input data description

Our industry partner provided us with a high-quality real-world data set for the problem. They represent the trains the industrial partner planned to serve in February 2020, as well as the information about drivers and locomotives which was up to date on February 14, 2020. It comprises six files – we will now describe each in detail.

6.2.0.0.1. Order book It is a list of all the trains that need to be performed, including their origin and destination stations, as well as departure and arrival times and assignment to calculation weeks. In the supplied instance, there are four calculation weeks, starting on Saturday and lasting till next Friday. It is contained in file `trains.csv`.

6.2.0.0.2. List of drivers This file comprises the list of all the drivers, including their licenses to locomotive types, knowledge of routes and assignments to regions. It is included in file `drivers.csv`.

6.2.0.0.3. List of locomotives In this file, information about all the available locomotives is included. In particular, it comprises their class, source of energy (electric / diesel) and power. For each train powered by a locomotive which is not the property of the industrial partner, an artificial entry is made, stipulating only the required locomotive class. All that information can be found in the file `unique_locos.csv`.

6.2.0.0.4. Distances between stations This file includes estimated distances and travel times between all the stations present in the order book. This information is required to be able to allow drivers to move between various stations while not driving a train during their shift. These times were estimated using the API of Google Maps. They were up to date as of February 14, 2020. These distances and travel times are included in the file `distance_matrix.csv`.

6.2.0.0.5. Assignment of stations to regions Here, each station present in the order book is assigned to one of the driver regions. It is included in file `station_region_mapping.csv`.

6.2.0.0.6. Assignment of drivers to regions This is an auxiliary source of information about the assignment of drivers to planning regions. It It is included in file `driver_region_mapping.csv`.

Based on the data we received, we have developed ten instances. Table 4 below presents a summary of parameters of each instance we develop.

instance	# days	# trains	# drivers	# locomotives
1M	29	2551	217	112
3W_1	21	1854	217	112
3W_2	21	1838	217	112
2W_1	14	1239	217	112
2W_2	14	1242	217	112
2W_3	14	1228	217	112
1W_1	7	629	217	112
1W_2	7	610	217	112
1W_3	7	615	217	112
1W_4	7	613	217	112

Table 4: Overview of instance parameters and model generation times

6.3. Step-by-step procedure

In this section, the requirements for the scripts to run will be given. We will also provide a step-by-step manual for the usage of the benchmarks introduced.

6.3.1. Requirements

Our code is written in Python 3.7. Hence, the host machine needs to have a distribution of Python 3.7 or newer installed. Apart from packages available in the Python Standard Library, we also used `numpy` (for some numeric manipulations) and `networkx` (for the representation of graph objects and graph-related algorithms).

Our model was built with `gurobipy`, which is Python's API to the routines of the Gurobi solver. We have used Gurobi 9.1 in our work. Although Gurobi is a proprietary software, a free non-commercial license is available to all the members of academic community.

6.3.2. Usage of the benchmarks

6.3.2.0.1. Step 0: Installation Provided a Python interpreter and the required packages are available on your machine, you can simply clone from the ROMSOC Github repository:

```
git clone https://github.com/ROMSOC/benchmarks-mip-rail-scheduling
```

6.3.2.0.2. Step 1: Choice of instance Navigate to `instances` directory. From there, navigate to a directory whose name matches the instance you would like to consider.

6.3.2.0.3. Step 2: Run model construction There are two keywords which need to be supplied when running the scripts:

- {`weekly`,`monthly`} – determines the scope of the model. If you chose `1M` as your instance, use `monthly`, otherwise use `weekly`.
- {`write`,`nowrite`} – determines whether or not the resulting model will be written to an output file `model.lp`.

For example, if you wish to consider the instance `1W_1` and to generate an output file, you need to type:

```
python main.py weekly write
```

If you wish to consider the instance `1M` and **not** to generate an output file, you need to type:

```
python main.py monthly nowrite
```

6.4. Description of output data

If the user wishes so, the benchmarks introduced may generate an `model.lp` file which contain a text description of the integer model. It presents the objective function, variables and constraints of each model. The `.lp` files can be opened with any text editor (such as Notepad). We generally advise against storing models for larger instances, since their size may easily grow to gigabytes, which will render them useless for manual browsing.

Part VII.
Benchmarks inverse heat transfer problem: Boundary heat flux estimation in continuous casting mold

Patricia Barral, Federico Bianco, Riccardo Conte, Umberto E. Morelli, Peregrina Quintela, Gianluigi Rozza, Giovanni Stabile

Abstract

The present document includes a detailed description of the computer implementation of a inverse heat transfer benchmark case involving not only the required publically available data but also the used software packages, libraries and any other relevant information, which guarantee a fully reproducibility of the reported numerical results.

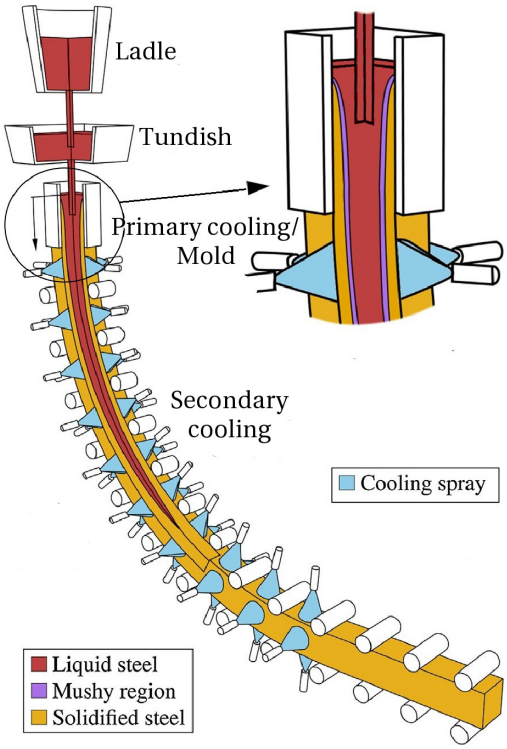
Keywords: Inverse problem, heat transfer, continuous casting mold, boundary flux estimation

Latest release: https://doi.org/10.5281/zenodo.5242918

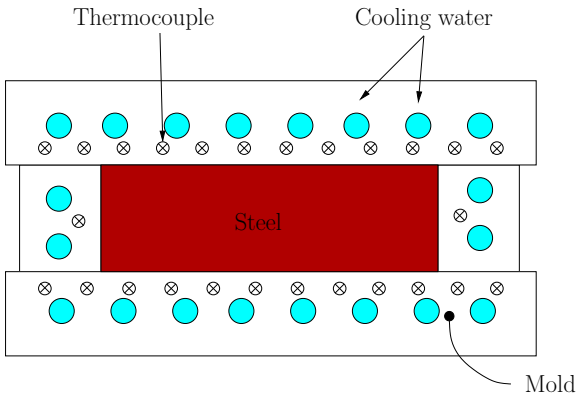
GitHub repository: https://github.com/ROMSOC/benchmarks-inverse-heat-transfer

7.1. Introduction

Continuous casting of steel is presently the most used process to produce steel worldwide. Figure 7.1(a) provides a schematic of the process. These casters are complex machinery, the most critical part for the process being the mold. Here the steel undergoes to its primary solidification. The mold extracts heat from the liquid steel thanks to a liquid cooling system. This system is composed of drilled channels in which water flows at high flow rate and pressure (see Figure 7.1(b)).



(a) Caster [61].



(b) Horizontal section of a mold.

Figure 7.1: Schematic of a continuous caster (a) and of a cross section of a mold (b).

For safety and quality reasons, it is important to control the heat extraction from the steel during the casting. For example, if the heat extraction is too little the steel solid skin is thin at its exit can brake. For this and other reasons, it is essential to know the real time behavior of the mold to properly control the casting process.

One way to compute this heat flux could be to simulate all the phenomena happening inside the mold: from the tapping of the molten steel to the secondary cooling region (e.g. multiphase flow, heat transport, solidification, thermodynamic reactions etc.). However, the resulting model would be quite complex and computationally expensive to deal with, especially for real time applications. Then, this option was discarded.

Also the fully experimental approach is no feasible since it is not possible to make direct measurements in the solidification region. The only measurements available are made by thermocouples that are buried inside the mold plates. They provide temperature measurements few centimeters into the mold. Then, our approach to study the real time behavior of continuous casting molds and the mold-slab heat flux is to solve an inverse problem having as data the thermocouples measurements.

7.1.1. Physical Problem

The physical phenomena happening in the interior of the mold are extremely complex and tightly coupled. Then, monitoring the casting by simulating all of them from the SEN to the secondary cooling region would be extremely complex and computationally expensive to deal with, especially for real-time applications. However, to monitor mold behaviour it is sufficient to know the mold-slab heat flux. Then, our approach is to solve an inverse problem having as control data the temperature measurements made by thermocouples that are buried inside the mold plates and the cooling water temperature measurements.

Using this approach, our domain is composed of the mold plates and the mold-slab heat flux is a Neumann BC on a portion of its boundary. Then, we only have to model the heat transfer in the mold plates.

In modeling the thermal behavior of the mold, we consider the following well established assumptions[62] :

- The copper mold is assumed a homogeneous and isotropic solid material.
- The cooling water temperature is known.
- The thermal expansion of the mold and its mechanical distortion are negligible.
- The material properties are assumed constant.
- The boundaries in contact with air are assumed adiabatic.
- No boiling in the water is assumed.
- The heat transmitted by radiation is neglected.

The adiabaticity of the boundaries in contact with air is justified when considering the magnitude of the heat extracted by the cooling water when compared to the one extracted by the still air around the mold. A similar justification, can be used for neglecting the mold radiation. Considering the thermal conductivity constant comes also from a practical consideration. CC molds are generally made by copper and they work in a temperature range in between $600\ K$ and $800\ K$. In this range the thermal conductivity varies of about 2%. Thus, this is the maximum error coming from this assumption.

Finally, since we want to have solution in real-time (e.g., at each second) and the casting speed is of few meters per minute, we consider steady-state models. Moreover, we only consider 3D mold models because we are interested in the heat flux in all the mold-slab interface.

As a final remark, the running parameters of the cooling system and its geometry ensure a fully developed turbulent flow. In fact, these molds are equipped with a closed loop cooling system where the water is pumped at a high pressure. The average velocity in each cooling channel is approximately 10 m/s, the diameter being approx. 10 mm. Thus, the Reynolds number in the cooling system is around 10^5, which ensures a turbulent flow.

Thanks to the high Reynolds number of the flow, we can further assume that the cooler and hotter water molecules are well mixed. Consequently, the temperature in each section of the cooling channel is approxi-

mately constant. Moreover the water is pumped in a closed circuit, so we can assume that the water flow rate is constant. In turn, since the channels have constant section, the velocity of the fluid is also uniform and constant (plug flow).

Then, we focus our attention on the following model:

1. The computational domain is only composed of the (solid) copper mold. We consider a steady-state three-dimensional heat conduction model with a convective BC in the portion of the boundary in contact with the cooling water. The water temperature is known at the inlet and outlet of the cooling system. The water temperature is assumed to be linear.

As a final remark on the model used, we consider a Neumann BC at the mold-steel interface instead of a convective BC because the perfect match between the strand and the mold is not ensured (air gaps are possible) and the strand surface temperature is also not known. So, in a convective BC situation, we would have to estimate the space varying heat transfer coefficient (that depends on the mold lubricant, the air gap, etc.) together with the strand surface temperature (also not constant), making the problem hardly solvable. Since the objective is to monitor the casting, estimating the heat flux provides to the CC operator all the information required for a proper control of the process and a fast problem detection.

In the following, we provide the mathematical formulation of the mold model and its numerical solution. Then, we formulate the respective inverse problem discussing the methodology for its solution. Finally, the core of this document is a step-by-step guide to the numerical tutorial.

7.2. Mathematical Formulation

Here, we only provide a short overview referring to [63] for a detailed description.

7.2.1. Computational Domain, Notation and Direct Problem

Let the domain be $\Omega = (0, L) \times (0, W) \times (0, H)$ as in Figure 7.2 with positive real constants L, W and H. Let Γ be boundary of Ω. Then, the different boundaries of the domain to be considered are

$$\begin{aligned} \Gamma_{sf} &:= \{\mathbf{x} \in \Gamma |\ \mathbf{x} = (x, W, z)\}, & \Gamma_{s_{in}} &:= \{\mathbf{x} \in \Gamma |\ \mathbf{x} = (x, 0, z)\}, \\ \Gamma_{I} &:= \{\mathbf{x} \in \Gamma |\ \mathbf{x} = (x, y, H)\}, & \Gamma_{III} &:= \{\mathbf{x} \in \Gamma |\ \mathbf{x} = (x, y, 0)\}, \\ \Gamma_{II} &:= \{\mathbf{x} \in \Gamma |\ \mathbf{x} = (L, y, z)\}, & \Gamma_{IV} &:= \{\mathbf{x} \in \Gamma |\ \mathbf{x} = (0, y, z)\}. \end{aligned}$$

We consider the following direct problem

Problem 1. *Find T such that*

$$-k\Delta T = 0, \ \textit{in } \Omega,$$

with BCs

$$\begin{cases} -k\nabla T \cdot \mathbf{n} = g_{an} & \textit{on } \Gamma_{s_{in}}, \\ -k\nabla T \cdot \mathbf{n} = q_L & \textit{on } \Gamma_L, L \in \{I, II, II, IV\}, \\ -k\nabla T \cdot \mathbf{n} = h(T - T_f) & \textit{on } \Gamma_{sf}. \end{cases}$$

Let a, b, c be real constants. To have an analytical solution in Ω, we consider the following data as BCs for Problem 1,

$$\begin{aligned} q_I(\mathbf{x}) &= 2kaH, & q_{III}(\mathbf{x}) &= 0, \\ q_{II}(\mathbf{x}) &= -k(2aL + by), & q_{IV}(\mathbf{x}) &= kby, \\ T_f(\mathbf{x}) &= \frac{k(bx + c)}{h} + ax^2 + cy - az^2 + bxW + c, \end{aligned}$$

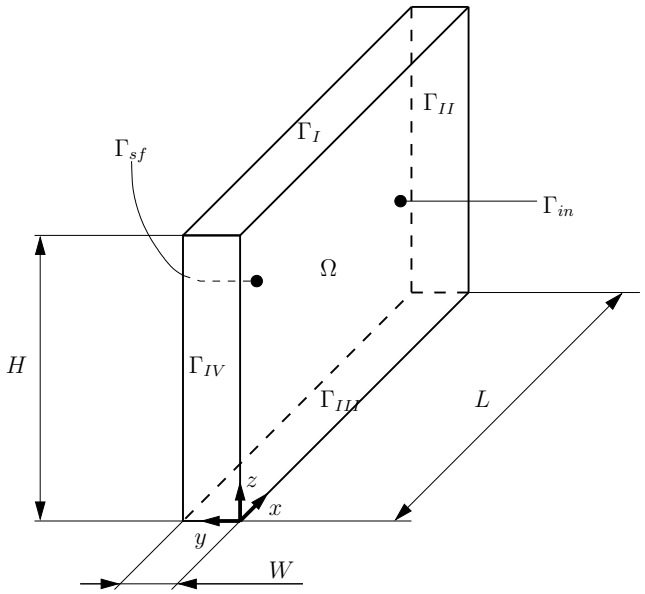


Figure 7.2: Schematic of the solid rectangular parallelepiped domain.

with

$$g_{an}(\mathbf{x}) = k(bx + c),$$

k being the thermal conductivity, that is assumed constant. Then,

$$T_{an}(\mathbf{x}) = ax^2 + bxy + cy - az^2 + c,$$

is the solution to Problem 1.

7.2.2. Inverse Problem

The inverse problem we want to solve is to estimate the heat flux g capable of reproducing the measured temperatures at the thermocouples' points. This can be stated as an optimal control problem with pointwise observations.

We introduce the following notation. Let $\Psi := \{\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_M\}$ be a collection of points in Ω. We define the application $\mathbf{x}_i \in \Psi \to \hat{T}(\mathbf{x}_i) \in \mathbb{R}^+$, $\hat{T}(\mathbf{x}_i)$ being the experimentally measured temperature at $\mathbf{x}_i \in \Psi$. Moreover, let G_{ad} be a bounded set in $L^2(\Gamma_{s_{in}})$.

Using a least square, deterministic approach, we state the inverse problem as

Problem 2. ***(Inverse)*** *Given $\{\hat{T}(\mathbf{x}_i)\}_{i=1}^M$, find the heat flux $g \in G_{ad}$ that minimizes the functional $J_1 : L^2(\Gamma_{s_{in}}) \to \mathbb{R}^+$,*

$$J_1[g] := \frac{1}{2}\sum_{i=1}^{M}[T[g](\mathbf{x}_i) - \hat{T}(\mathbf{x}_i)]^2,$$

where $T[g](\mathbf{x}_i)$ is the solution of Problem 1 at points $\mathbf{x}_i$, for all $i = 1, 2, \ldots, M$.

For the solution of this problem, we use two different methods. These methods are described in the following two subsections.

7.2.2.1. Alifanov's Regularization

The Alifanov's regularization method is a CGM applied on the adjoint equation.[64]

We consider the following iterative procedure for the estimation of the function g that minimizes the functional (2). Given an initial estimation $g^0 \in L^2(\Gamma_{s_{in}})$, for $n > 0$ a new iterant is computed as:

$$g^{n+1} = g^n - \beta^n P^n, \quad n = 0, 1, 2, \dots$$

where n is the iteration counter, β^n is the stepsize, also called correction factor, in the conjugate direction P^n given by

$$P^0 = J_1'[g^0], \quad P^{n+1} = J_1'[g^{n+1}] + \gamma^{n+1} P^n \text{ for } n \geq 1,$$

γ^{n+1} being the conjugate coefficient, and $J_1'[g]$ the Gâteaux derivative of J_1 given by

$$J_1'[g] = -\lambda[g] \text{ in } L^2(\Gamma_{s_{in}}),$$

where λ is the solution of

Problem 3. *(**Adjoint**) Find $\lambda[g]$ such that*

$$k\Delta\lambda[g] + \sum_{i=1}^{M} (T[g](\mathbf{x}) - \hat{T}(\mathbf{x}))\delta(\mathbf{x} - \mathbf{x}_i) = 0, \quad \textit{in } \Omega,$$

with BCs

$$\begin{cases} k\nabla\lambda[g] \cdot \mathbf{n} = 0 & \textit{on } \Gamma_{s_{in}} \cup \Gamma_{s_{ex}}, \\ k\nabla\lambda[g] \cdot \mathbf{n} + h\lambda[g] = 0 & \textit{on } \Gamma_{sf}, \end{cases}$$

$\delta(\mathbf{x} - \mathbf{x}_i)$ being the Dirac function centered at $\mathbf{x}_i$.
The stepsize β^n in (7.2.2.1) is obtained by minimizing the functional $J_1[g^n - \beta P^n]$ with respect to β. Therefore, β^n is the solution of the critical point equation of the functional J_1, restricted to a line passing through g^n in the direction defined by P^n, i.e. β^n is the critical point of $J_1[g^n - \beta P^n]$ which then satisfies

$$J_1[g^n - \beta^n P^n] = \min_{\beta} \left\{ \frac{1}{2} \sum_{i=1}^{M} \left[T[g^n - \beta P^n](\mathbf{x}_i) - \hat{T}(\mathbf{x}_i) \right]^2 \right\}.$$

Stating the sensitivity problem

Problem 4. *(**Sensitivity**) Find δT such that*

$$-k\Delta\delta T[\delta g] = 0, \quad \textit{in } \Omega,$$

with BCs

$$\begin{cases} -k\nabla\delta T[\delta g] \cdot \mathbf{n} = \delta g & \textit{on } \Gamma_{s_{in}}, \\ -k\nabla\delta T[\delta g] \cdot \mathbf{n} = 0 & \textit{on } \Gamma_{s_{ex}}, \\ -k\nabla\delta T[\delta g] \cdot \mathbf{n} = h(\delta T[\delta g]) & \textit{on } \Gamma_{sf}, \end{cases}$$

we can now write

$$J_1[g^n - \beta P^n] = \frac{1}{2} \sum_{i=1}^{M} \left[T[g^n - \beta P^n](\mathbf{x}_i) - \hat{T}(\mathbf{x}_i) \right]^2 = \frac{1}{2} \sum_{i=1}^{M} \left[(T[g^n] - \beta\delta T[P^n])(\mathbf{x}_i) - \hat{T}(\mathbf{x}_i) \right]^2.$$

Differentiating with respect to β, we obtain the critical point equation

$$\frac{dJ_1[g^n - \beta^n P^n]}{d\beta} = \sum_{i=1}^{M}[(T[g^n] - \beta^n \delta T[P^n])(\mathbf{x}_i) - \hat{T}(\mathbf{x}_i)](-\delta T[P^n](\mathbf{x}_i)) = 0.$$

Finally, we have

$$\beta^n = \frac{\sum_{i=1}^{M}\left[T[g^n](\mathbf{x}_i) - \hat{T}(\mathbf{x}_i)\right]\delta T[P^n](\mathbf{x}_i)}{\sum_{i=1}^{M}(\delta T[P^n](\mathbf{x}_i))^2}.$$

With respect to the conjugate coefficient, γ, its value is zero for the first iteration and for other iterations it can be calculated using Fletcher-Reeves expression as follows[65]

$$\gamma^n = \frac{\|J_1'[g^n]\|^2_{L^2(\Gamma_{s_{in}})}}{\|J_1'[g^{n-1}]\|^2_{L^2(\Gamma_{s_{in}})}}.$$

Notice that, to use this iterative procedure, we have to compute at each iteration the Gâteaux derivative $J_1'[g](\mathbf{x})$ which is given by (7.2.2.1). Thus, we must solve the adjoint problem to compute it.

Alifanov's regularization algorithm is summarized in Algorithm 1.

Algorithm 1: Alifanov's regularization.

Set g^0 and $n = 0$ **while** $n < n_{max}$ **do**
    Compute $T[g^n]$ by solving Problem 1
Compute $J_1[g^n]$ by (2) **if** $J_1[g^n] < J_{1_{tol}}$ **then**
    Stop
Compute $\lambda[g^n]$ by solving Problem 3
Compute $J_1'[g^n]$ by (7.2.2.1) **if** $n \geq 1$ **then**
    Compute the conjugate coefficient, γ^n, by (7.2.2.1)
Compute the search direction, P^n, by (7.2.2.1) **else**
    $P^0 = J_1'[g^0]$
Compute $\delta T[P^n]$ by solving Problem 4 with $\delta g = P^n$
Compute the stepsize in the search direction, β^n, by (7.2.2.1)
Update heat flux g^n by (7.2.2.1)
$n = n + 1$
return g^n

7.2.2.2. Parameterization of the Boundary Conditions

We now discuss the second method used for the solution of the inverse problem. We consider the parameterization of the boundary heat flux g. In the literature, the parameterization of g has already been proposed.[66] However, we propose a novel approach both for the parameterization and for the solution of the resulting inverse problem.

For the parameterization, we start considering that we want to parameterize an unknown function in $L^2(\Gamma_{s_{in}})$. We then notice that in thin slab casting molds, the thermocouples are all located few millimeters inward from $\Gamma_{s_{in}}$. All together they form a uniform 2D grid. Then, to parameterize g, we use Radial Basis Functions (RBFs) centered at the projections of the thermocouples' points on $\Gamma_{s_{in}}$.[67] Due to this choice we have as many basis functions as thermocouples. In particular, we use Gaussian RBFs that are continuous functions with a global support. However, the following discussion can be applied to other basis functions.

The parameterization of the boundary heat flux reads (see Prando's appendix[68])

$$g(\mathbf{x}) \approx \sum_{j=1}^{M} w_j \phi_j(\mathbf{x}),$$

where the $\phi_j(\mathbf{x})$ are M known base functions, and the w_j are the respective unknown weights. Notice that by doing the parameterization, we change the problem from estimating a function in an infinite dimensional space to estimating a vector $\mathbf{w} = (w_1, w_2, \dots, w_M)^T$ in $\mathbb{R}^M$.

Let $\boldsymbol{\xi}_i, 1 \leq i \leq M$, be the projection of the measurement point $\mathbf{x}_i \in \Psi$ on $\Gamma_{s_{in}}$ such that

$$\boldsymbol{\xi}_i = \underset{\boldsymbol{\xi} \in \Gamma_{s_{in}}}{\operatorname{argmin}} \left\| \mathbf{x}_i - \boldsymbol{\xi} \right\|_2, \quad \mathbf{x}_i \in \Psi.$$

By centering the RBFs in these points, their expressions are

$$\phi_j(\mathbf{x}) = e^{-\left(\eta \|\mathbf{x} - \boldsymbol{\xi}_j\|_2\right)^2}, \quad \text{for } j = 1, 2, \dots, M,$$

where η is the shape parameter of the Gaussian basis, increasing its values the radial decay of the basis slows down.

Suppose to have the solutions of Problem 1, $T[\phi_j]$, for $j = 1, 2, \dots, M$. Denote by T_{ad} the solution of

Problem 5. *Find T_{ad} such that*

$$-k\Delta T_{ad} = 0, \quad \textit{in } \Omega,$$

with BCs

$$\begin{cases} -k\nabla T_{ad} \cdot \mathbf{n} = 0 & \textit{on } \Gamma_{s_{in}} \cup \Gamma_{s_{ex}}, \\ -k\nabla T_{ad} \cdot \mathbf{n} = h(T_{ad} + T_f) & \textit{on } \Gamma_{sf}. \end{cases}$$

As described in [63], the solution of the inverse problem is then obtained by solving the linear system

$$\Theta^T \Theta \mathbf{w} = \Theta^T (\hat{\mathbf{T}} + \mathbf{T}_{ad}).$$

This is generally called the normal equation.

We conclude our discussion of this method by noticing its most interesting feature for our investigation. In fact, it is already suitable for real-time computation since we can divide it into an offline (expensive) phase and an online (cheap) phase. In the offline phase, we compute $T[\phi_j]$ for $j = 1, 2, \dots, M$ and T_{ad} by solving Problem 1 with each base as boundary heat flux and Problem 5. Then in the online phase, we input the measurements $\hat{\mathbf{T}}$ and solve the linear system (7.2.2.2). For the choice made when selecting the basis functions, the linear system has the dimensions of the number of thermocouples. Then, its solution can be easily done in real-time even with limited computational power. This makes this method very promising for our real-time application.

7.3. Software Implementation

The proposed benchmark case has been implemented as a tutorial in ITHACA-FV[69, 70] which is a C++ library based on OpenFOAM[71] developed at the SISSA Mathlab. ITHACA-FV is freely available for the download under the GNU LGPL, version 3, license at the dedicated GitHub page `https://github.com/mathLab/ITHACA-FV`. At this link, an up to date installation and usage guide is provided together with the prerequisites for the installation.

This benchmark case is implemented into ITHACA-FV as the tutorial `IHTP01inverseLaplacian`. It can be found at the path `ITHACA-FV/tutorials/inverseHeatTransfer`.

7.4. Input Data

We now discuss the physical parameters used in this benchmark case, the domain, its discretization and the input available for the user.

Table 5 shows the default physical and geometrical parameters used in this tutorial. The thermal conductivity, k, and the heat transfer coefficient, h, can be selected by the user in the `ITHACAdict` that can be found in the `caseDir/system` folder. The a, b, c and d parameters are defined at the beginning of the `IHTP01inverseLaplacian.C` file. Finally, the domain and its discretization are controlled by the `blockMeshDict` in the `caseDir/system` folder. We refer to the OpenFOAM User Guide [72] for details on the usage of this dictionary.

Parameter	Value
Thermal conductivity, k	$3.0\ W/(mK)$
Heat transfer coefficient, h	$5.0\ W/(m^2K)$
a	$5\ K/m^2$
b	$10\ K/m^2$
c	$15\ K/m^2$
L	$1\ m$
W	$1\ m$
H	$1\ m$

Table 5: Parameters used for the benchmark case.

As usual in OpenFOAM, the `fvSchemes` and `fvSolutions` dictionaries allows to control the equations discretization schemes and the methods used for the solution of the related linear systems. Both these dictionaries are in the `caseDir/system` directory. Also in this case, we refer to the OpenFOAM User Guide [72] for details.

Another input to the inverse problem must be the position of the thermocouples. The thermocouples' locations are defined in the `thermocouplesDict` in the `caseDir/constant` directory. The default values are as in Figure 7.3.

To conclude in the `ITHACAdict` dictionary, in the `caseDir/system` directory, the user can select the parameters for the Alifanov's regularization algorithm and the parameterization of the BC method. In particular, he/she can select the maximum number of iterations, `cgIterMax`, the absolute, `Jtolerance`, and relative, `JrelativeTolerance`, tolerance for the Alifanov's regularization algorithm. While, for the parameterization of the BC method, the user can select the shape parameter for the radial basis functions used for the parameterization, `rbfShapePar`. Moreover, in this same dictionary, the user can which test to perform setting to one the following flags:

- `CGtest` - Alifanov's regularization test;
- `parameterizedBCtest` - Parameterization of the BC test;
- `parameterizedBCtest_RBFwidth` - Test of the effects of different RBF shape parameter values for the parameterization of the BC method;
- `thermocouplesLocationTest_CG` - Test of the effects of moving the thermocouples plane distance from the mold hot face for the Alifanov's regularization;
- `thermocouplesLocationTest_paramBC` - Test of the effects of moving the thermocouples plane distance from the mold hot face for the parameterization of the BC method;
- `thermocouplesNumberTest_CG` - Test of the effects of changing the number of thermocouples in the Alifanov's regularization;
- `thermocouplesNumberTest_paramBC` - Test of the effects of changing the number of thermocouples in the parameterization of the BC method.

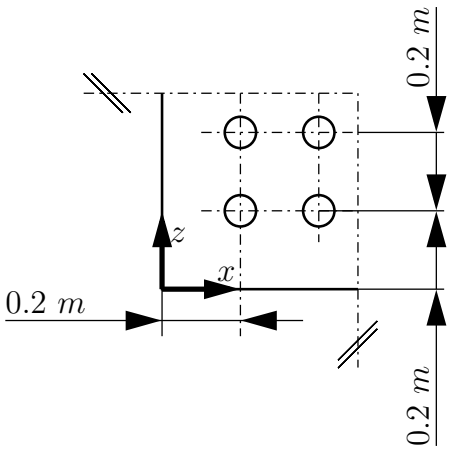


Figure 7.3: Positions of the virtual thermocouples for the benchmark case.

7.5. Step-by-step procedure

To run this ITHACA-FV benchmark, the user must go into the `caseDir` directory. The first thing to do is to run the OpenFOAM command `blockMesh` to create the mesh. Then, the used can select the tests to be run in the `ITHACAdict` dictionary and run the tutorial by typing the command `IHTP01inverseLaplacian`. The outputs are all saved into the directory `caseDir/ITHACAoutputs` in different subdirectories according to the different tests.

7.6. Output Data

The output of the simulations are all saved into the `caseDir/ITHACAoutputs` folder. For the post processing of the results, several python codes are available at the directory `pythonPlots`. To obtain the post processing plot, the user must run the command `python required-plots.py` where `required-plots` should be selected by the ones available in the folder, e.g. `python CGconvergence.py` to see the convergence of the Alifanov's regularization and the behaviour of the error with the iterations.

Part VIII.
Validation of Fluid-Structure Interaction Simulations in Membrane-Based Blood Pumps

Marco Martinolli, Christian Vergara, Luc Polverelli

Abstract

The benchmark consists in the validation of a numerical model for the fluid-structure interaction arising in membrane-based blood pumps against experimental data obtained by *in vitro* testings at CorWave Inc. The goal is to numerically reproduce the pump system under the same working conditions of the documented experimental sessions, in order to measure the pressure rise over the pump and the hydraulic power for different inflow velocities and finally compare the results with the experimental PQ and HQ curves. The software for the solution of the benchmark will be implemented in the C++ parallel library of finite elements LIFEV and tackled using the Extended Finite Element Method. The report includes the plan for the Docker installation of the LIFEV environment and the third-part packages, information on the future online availability of the software, the licences of use of the main libraries and the computer requirements to run the benchmark. The proposed benchmark can be used for training in the fields of mathematical modeling of a coupled system, model testing and error estimation.

Keywords: Benchmark, Fluid-Structure Interaction, Blood Pumps, Model Validation, HQ curves

Latest release: https://doi.org/10.5281/zenodo.5171806

GitHub repository: https://github.com/ROMSOC/benchmark-validation-wmbp

8.1. Introduction

WMBP! (**WMBP!**) [73], developed at CorWave SA, may represent the new frontier of **LVAD!** (**LVAD!**). The pumping technnology of WMBP is based on the wave undulations of an immersed elastic membrane that propels the blood against an adverse pressure gradient, from the left ventricle into the ascending aorta. In Figure 8.1, left, we show the cross sectional view of the pump device.

The intertwined dynamics in WMBP are studied in the framework of Fluid-Structure Interaction (FSI) modeling and solved in the mathematical domain shown in Figure 8.1, right. Blood is modeled as a viscous incompressible Newtonian fluid with density ρ_f and viscosity μ_f, by means of Navier-Stokes Equations. The wave membrane Ω_1^s is considered a linear material at small deformations, according with previous publication [74], with density ρ_s^1 and Lamé parameters λ_s^1 and μ_s^1. The same holds for the magnet ring Ω_2^s, with parameters ρ_s^2, λ_s^2 and μ_s^2. For sake of simplicity, we combine structure properties in unique space-dependante variables, e.g. $\tilde{\rho_s}(\mathbf{x}) = \rho_s^1$ if $\mathbf{x} \in \Omega_1^s$, and $\tilde{\rho_s}(\mathbf{x}) = \rho_s^2$ if $\mathbf{x} \in \Omega_2^s$. Hence, the formulation of the problem is the following: for each time $t > 0$, find fluid velocity and pressure $(\mathbf{u}(t), p(t))$ in the fluid domain $\Omega^f(t)$ and structures displacement $\widehat{\mathbf{d}}(t)$ in the reference structure domain $\widehat{\Omega}^s = \Omega^s(0) = \Omega_1^s(0) \cup \Omega_2^s(0)$, such that:

$$
\begin{aligned}
&\rho_f \left(\partial_t \mathbf{u} + \mathbf{u} \cdot \nabla \mathbf{u}\right) - \nabla \cdot \mathbf{T}^f(\mathbf{u}, p) = \mathbf{0} && \text{in } \Omega^f(\mathbf{d}), \\
&\nabla \cdot \mathbf{u} = 0 && \text{in } \Omega^f(\mathbf{d}), \\
&\tilde{\rho_s} \partial_{tt} \widehat{\mathbf{d}} - \nabla \cdot \widehat{\mathbf{T}}^s(\widehat{\mathbf{d}}) = \mathbf{0} && \text{in } \widehat{\Omega}^s, \\
&\mathbf{u} = \partial_t \mathbf{d} && \text{on } \Sigma(\mathbf{d}), \qquad (8.1) \\
&\mathbf{T}^f(\mathbf{u}, p)\, \mathbf{n} = \mathbf{T}^s(\mathbf{d})\, \mathbf{n} && \text{on } \Sigma(\mathbf{d}). \qquad (8.2)
\end{aligned}
$$

where $\mathbf{T}^f(\mathbf{u}, p) = -p\mathbf{I} + 2\mu_f \mathbf{D}(\mathbf{u})$, with $\mathbf{D}(\mathbf{u}) = \frac{1}{2}(\nabla \mathbf{u} + \nabla \mathbf{u}^T)$, is the fluid Cauchy stress tensor and $\widehat{\mathbf{T}}^s(\widehat{\mathbf{d}}) =$

$\tilde{\lambda}_s\,(\widehat{\nabla}\cdot\widehat{\mathbf{d}})\,\mathbf{I}+2\tilde{\mu}_s\,\widehat{\mathbf{D}}(\widehat{\mathbf{d}})$ is first Piola-Kirchhoff structure tensor. Notice that the fluid domain Ω_f and the interface Σ depend over time on structure displacement $\mathbf{d}$ (geometric coupling). Physical coupling conditions (8.1) and (8.2) guarantee the continuity of velocity and of stresses at the fluid-structure interface Σ, respectively.

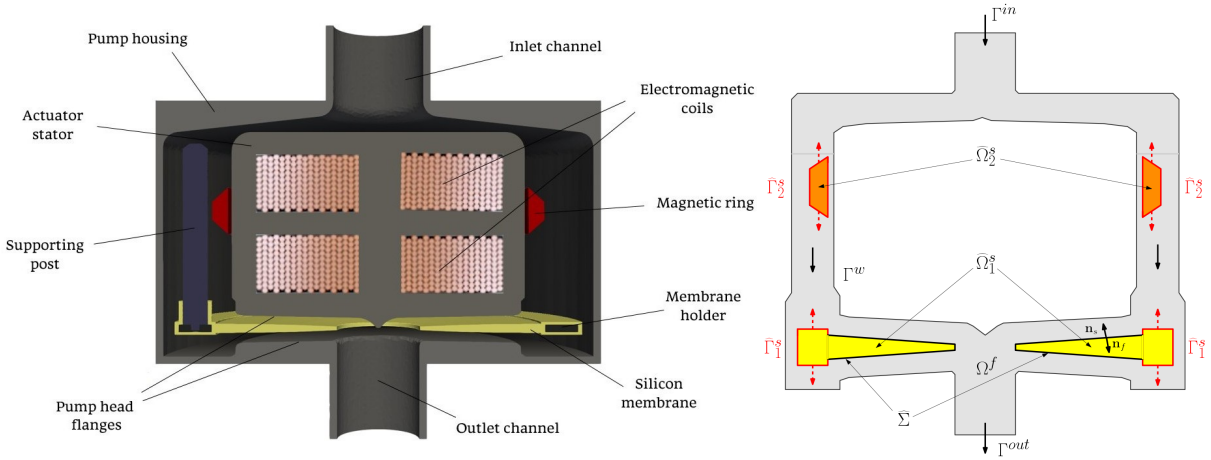


Figure 8.1: Left: Cross section of wave membrane blood pumps. The blood enters from the inlet channel, it flows down along the sides of the central actuator body, it interacts with the wave silicon membrane and it is finally ejected into the outlet channel. Membrane vibrations are triggered by the oscillations of the magnet ring. Right: Representation of the mathematical domain.

The boundary conditions define the operating point of WMBP. For the fluid sub-problem, the hydraulic resistance in the pump, that is the head pressue H, is prescribed by means of a pair of Neumann conditions at the inlet Γ^{in} and at the outlet Γ^{out}; while non-slip wall conditions are imposed at boundary Γ^w:

$$\begin{aligned} \mathbf{T}^f(\mathbf{u},p)\,\mathbf{n}^f &= \mathbf{0} && \text{on } \Gamma^{in}, \\ \mathbf{T}^f(\mathbf{u},p)\,\mathbf{n}^f &= H\,\mathbf{n}^f && \text{on } \Gamma^{out}, \\ \mathbf{u} &= \mathbf{0} && \text{on } \Gamma^{w}, \end{aligned}$$

The progressive undulations in the membrane are caused by the oscillations of the magnet ring with boundary Γ_2^s; then, they are transmitted to the external edge of the membrane Γ_1^s. Such oscillations are assumed to be purely sinusoidal with frequency f and amplitude Φ. Thus, the oscillations are prescribed by means of a Dirichlet condition on boundaries Γ_i^s $(i=1,2)$:

$$\mathbf{d}\,(t) = \Phi\,\sin\left(2\pi\,f\,t\right)\,\mathbf{e}_z \qquad \text{on } \Gamma^s$$

The benchmark consists in the validation of a numerical method to solve the FSI problem in WMBP against real hydraulic data provided by CorWave SA. Specifically, the goal is to predict the outflow volume rate Q^{sim} given a certain operating point of the pump (H, f, Φ) and compare the numerical result with the measured flow data Q^{data}.

In this report, we will describe the steps to take to solve the FSI problem with our numerical strategy, based on the Extended Finite Element Method (XFEM) [75, 76]. XFEM is an unfitted technique which has two main advantages compared with other approaches for FSI problems: i) since the fluid mesh is kept fixed on the background, it avoids the remeshing procedure normally occurring in case of element distortion; ii) the accuracy of the solution is maintained at the interface, thanks to the local enrichment of the functional space of the extended finite elements. However, since the structure mesh moves in the foreground cutting the underlying fluid mesh, XFEM requires to compute the mesh intersections at each time istant to identify the fluid elements that are cut in multiple subportions (called *split elements*), leading to a higher computational cost. For more

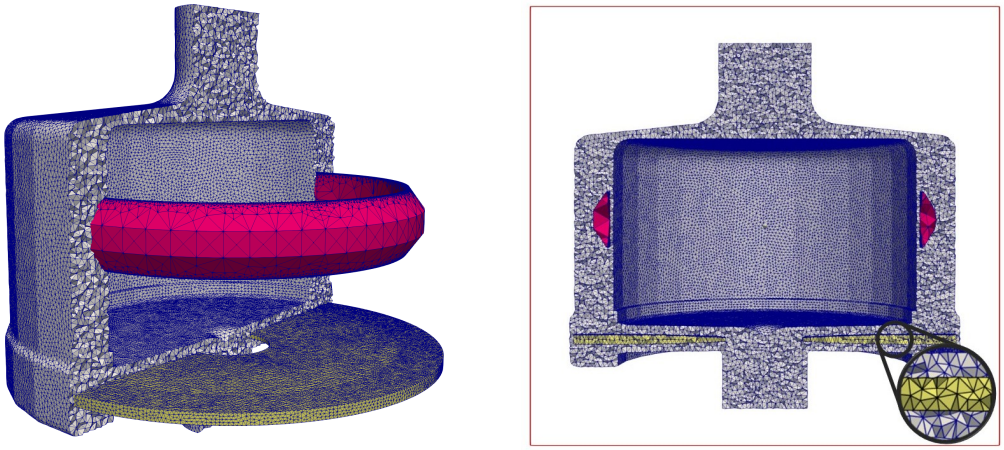


Figure 8.2: Prospective visualization and section of the fluid mesh (gray), the membrane mesh (yellow) and magnet mesh (red).

details on the numerical formulation of the XFEM with a DG mortaring at the interface, the reader can find an exaustive explanation in the reference papers [77, 78].

8.2. Input data

Input data to run the benchmark consist of four different types of files:

1. **meshes**: a folder (available from the Docker container) that includes the fluid mesh, the membrane mesh and the magnet mesh;
2. **dataFile**: it details the list of meshes, physical parameters, stability parameters, operating parameters, time settings and other numerical parameters;
3. **solverFile**: it contains the parameters for the linear solver;
4. **validation data**: a csv datafile that collects the experimental data to be used to validate the numerical results

8.2.1. Meshes

For this benchmark, we consider the flat membrane pump geometry, studied in [77]. The CAD geometry files have been provided by the partner company CorWave SA. The meshes of tetrahedra, reported in Figure 8.2, have been created using GMESH [79]. The unfitted meshes selected for this benchmark showed positive convergence results. Lengths are expressed in cm and may not correspond to the real pump design.

8.2.2. DataFile

The dataFile presents important information for the simulations, such as physical properties of the fluid and of the structures, discretization parameters, Operating Point (OP) parameters, Continuous Interior Penalty (CIP) stability parameters, and many others. The default values of all these data are reported in Table 6.

The experimental data for the validation are provided only for oscillation frequency of 120 Hz and amplitude Φ of 0.053 cm. Hence they should not be changed. Also notice that pressure conditions can largely affect the pump dynamics and consequently the stability requirements. For this reason, we suggest the user to set the head pressure parameter H and the CIP penalty parameters according to Table 7. All parameters are expressed in the unit system cgs.

Additional information in the DataFile include the strings of the mesh files (fluid: 1.2M elements, membrane: 300k elements, magnet: 50k elements) and of the solverFile (solverFile.xml is the default) and other precondi-

Section	Variable	Value
Blood properties	Density ρ_f	$1\ \frac{g}{cm^3}$
	Viscosity μ_f	$0.035\ \frac{dyne}{cm^2}$
Membrane properties	Density ρ_s^1	$1.125\ \frac{g}{cm^3}$
	Young Modulus E_s^1	$1.686 \cdot 10^7\ \frac{dyne}{cm^2}$
	Poisson ration ν_s^1	0.49
Magnet properties	Density ρ_s^2	$7.85\ \frac{g}{cm^3}$
	Young Modulus E_s^2	$2.05 \cdot 10^{12}\ \frac{dyne}{cm^2}$
	Poisson ration ν_s^2	0.28
OP parameters	Head Pressure H (*)	50 mmHg
	Frequency f	120 Hz
	Amplitude Φ	0.053 mm
Stability CIP parameters	Convection control γ_{v1} (*)	0.05
	Divergence control γ_{v2} (*)	0.5
	Inf-sup control γ_p (*)	0.05
Time settings	Initial time t_0	0 s
	Final time T	0.025 s
	Time step Δt	0.0002 s

Table 6: Default physical and discretization parameters defined in `dataFile`. Variables with (*) can be modified.

	$H = 50$ mmHg	$H = 55$ mmHg	$H = 60$ mmHg
γ_{v1}	0.05	0.05	0.5
γ_{v2}	0.5	0.5	5
γ_p	0.05	0.1	0.1

Table 7: Continuous Interior Penalty parameters for different head pressure conditions.

tioner settings.

8.2.3. SolverFile

The solverFile, reported in Appendix 8.5.A, collects some numerical settings for the solver of the linear system. In general, such parameters should be kept fixed for parallel runs with more than 3 processors (10 cores are suggested to reduce computational cost). In case of serial runs or less than 3 processors, the user has to reduce the dimension of Krylov space (kspace) to 200 and of the number of maximum iterations (max_iter) to 500. Hence, in this case use file *SolverFile_serial.xml*.

8.2.4. Validation data

The experimental data used for the benchmark are *HQ curves* (H head pressure, Q flow rate), which are standard representations of pump hydraulic performance. They are obtained during the *in vitro* testings in a pump characterization bench, measuring the pressure difference H between the outlet and the inlet (called head pressure), and the corresponding flow rate Q generated at the outlet. The head pressure represents the hydraulic resistance of the system (adverse pressure gradient), that has to be overcome to generate pump outflow. Figure 8.3 shows the HQ curve of the pump system when the frequency of oscillation of the membrane is fixed to 120 Hz and the amplitude of oscillation to 0.053 cm. Notice that the flow rate decreases as the head pressure increases, owing to the higher hydraulic resistance in the pump. Raw data are provided in the Excel file *HQ_curve.csv*, consisting of two separate data columns with the measurements of head pressure in mmHg (1

mmHg = 1333.22 g /cm s^2) and of the outflow volume rate in liters per minute, or lpm (1 lpm = 0.06 cm^3/s).

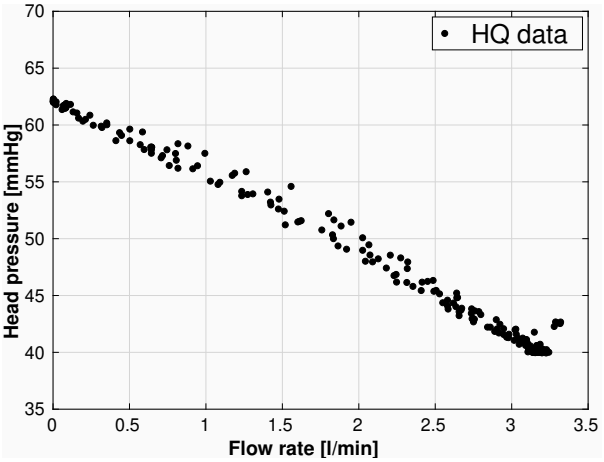


Figure 8.3: HQ experimental curve at oscillation frequency $f = 120$ Hz and amplitude $\Phi = 0.053$ cm.

8.3. Output data

The output of interest for this benchmark is the flow volume rate at the pump outlet Γ^{out}. Indeed, the goal is to compare, for a certain input head pressure H, the corresponding estimated outflow rate Q^{sim} with the experimental data Q^{data} extracted from the HQ curves (see Figure 8.3). At each time iteration, the software prints out the outflow computed at that time istant. Since this quantity oscillates in time with frequency f, we need to compute the average in time of the outflow results at regime, i.e. during the last period of oscillation τ. Thus:

$$Q^{sim} = \int_{T-\tau}^{T} \int_{\Gamma^{out}} \mathbf{u}(\mathbf{x}, t) \cdot \mathbf{n} \, d\mathbf{x} \, dt$$

Finally, a Python scrupt can be used to plot the HQ curve and the obtained numerical results Q^{sim} to check their proximity and compute absolute and relative errors. The user has just to type in the simulated head pressure H. In Table 8 we show the expected results for this benchmark for certain head pressure conditions, as reported in [77].

	$\Delta P = 50$ mmHg	$\Delta P = 55$ mmHg	$\Delta P = 60$ mmHg
Q^{data}	$1.834 \, \frac{\text{l}}{\text{min}}$	$1.091 \, \frac{\text{l}}{\text{min}}$	$0.352 \, \frac{\text{l}}{\text{min}}$
Q^{sim}	$1.792 \, \frac{\text{l}}{\text{min}}$	$1.039 \, \frac{\text{l}}{\text{min}}$	$0.400 \, \frac{\text{l}}{\text{min}}$
$\lvert Q^{data} - Q^{sim} \rvert$	$0.042 \, \frac{\text{l}}{\text{min}}$	$0.052 \, \frac{\text{l}}{\text{min}}$	$0.048 \, \frac{\text{l}}{\text{min}}$

Table 8: Experimental and simulation data for the model validation against experimental mesures.

8.4. Procedure

The aim of the proposed benchmark is to validate the FSI model via comparison of the numerical results against experimental HQ curves in wave membrane blood pumps. In this section, we explain the steps to take to install the software, set the data, run the FSI simulations and post-process the computed numerical results.

8.4.1. Step 0: Installation

The material to run the benchmark, including source files, validation data and post-processing tool, is available in the GitHub repository at https://github.com/ROMSOC/benchmark-validation-wmbp and can be downloaded or clone in your local machine. In addition, a docker image with the LifeV environment and all the additional libraries required to solve this benchmark is publically available in DockerHub (free registration to https://hub.docker.com/ is required). Then to download it, it it is sufficient to type:

```
docker pull martin592/lifev-validation:env
```

The DockerFile used to contruct such image is available in the Github repository.

Finally a Docker container can be generated using file `runLifeV.sh`:

```
./runLifeV.sh
```

Now you should have created a Docker LifeV container, including a copy of all the folders in the benchmark repository and all the libraries required to run the benchmark. All the changes in the container will be copied in the local repository.

8.4.2. Step 1: Setting input data

Move to the benchmark/ subfolder. The input data needed to run the simulations include i) the meshes/ folder, ii) the dataFile, and iii) the solverFile. Check that all these documents are present in the working directory.

Most of the input settings need to be maintaiend fixed for this benchmark. However, the user should set the head pressure H (in mmHg) in the pump by changing variable `[OP/pressure]` in the `dataFile` (default value is 50 mmHg). We suggest also to change the penalty parameters at lines `[penalty]` of the `dataFile`, according to what indicated in Table 7 to ensure stability.

Finally, in case of serial runs or for parallel runs with less than 4 processors, change the variable `[prec/paramListFile]` to `solverFile_serial.txt`. Also, move `solverFile_serial.txt` from paramsFile/ subfolder into the working directory.

8.4.3. Step 2: Run simulations

To run the FSI simulations in wave membrane blood pumps, it is sufficient to execute the program with the following command from command line:

```
./WMBP.exe -f dataFile > benchmark_output.txt
```

in case of serial runs, or

```
mpirun -np 7 WMBP.exe -f dataFile > benchmark_output.txt
```

in case of parallel runs with a number of processors equal to 7.

Notice that these simulations can last for several days and require a minimum of 40 GB of RAM for serial runs. In addition, simulation may stop due to the failure of external library tetgen, trying to solve particularly complex geometric sub-problems, inherent to XFEM-based strategy. In this case, the user should restart the simulation from the last saved time step, following the instructions detailed in the Appendix 8.5.B.

8.4.4. Step 3: Post-processing

The numerical results of interest for this benchmark consist of the flow rate at the outlet. At each time istant, the software prints the computed outflow rate in the output file `benchmark_output.txt`. Hence, move to post-processing/ sub-directory and copy the output file `benchmark_output.txt` in it.

Post-processing is structured in two steps:

1. extract the time series of the outflow rate from the output file `benchmark_output.txt` using Python script `extract_flowResults.py` as follows:

```
python extract_flowResults.py benchmark_output.txt .
```

2. run the Python code `validation.py`

```
python validation.py
```

   to compute the mean flow rate Q^{sim} as in Equation 8.3, plot it with the HQ data curve, and compute the numerical error.

A Python installation (version $\geq$ 2.7) is required, together with Python packages numpy and matplotlib.

8.5. Appendix

8.5.A. SolverFile.xml

```
<ParameterList>
<!-- LinearSolver parameters -->
<Parameter name="Reuse Preconditioner" type="bool" value="false"/>
<Parameter name="Max Iterations For Reuse" type="int" value="80"/>
<Parameter name="Quit On Failure" type="bool" value="false"/>
<Parameter name="Silent" type="bool" value="false"/>
<Parameter name="Solver Type" type="string" value="AztecOO"/>

<!-- Operator specific parameters (AztecOO) -->
<ParameterList name="Solver: Operator List">

<!-- Trilinos parameters -->
<ParameterList name="Trilinos: AztecOO List">
<Parameter name="solver" type="string" value="gmres"/>
<Parameter name="conv" type="string" value="rhs"/>
<Parameter name="scaling" type="string" value="none"/>
<Parameter name="output" type="string" value="all"/>
<Parameter name="tol" type="double" value="1.e-6"/>
<Parameter name="max_iter" type="int" value="4000"/>
<Parameter name="kspace" type="int" value="2000"/>
<!-- az_aztec_defs.h -->
<!-- #define AZ_classic 0 /* Does double classic */ -->
<Parameter name="orthog" type="int" value="0"/>
<!-- az_aztec_defs.h -->
<!-- #define AZ_resid 0 -->
<Parameter name="aux_vec" type="int" value="0"/>
</ParameterList>
</ParameterList>
</ParameterList>
```

Code Listing 3: solverFile.xml

8.5.B. Simulation - Restart

In case of running error, a restart procedure has to be carried over to continue the simulation from the last saved timestep. At each time instant, the software exports the fluid and solid solutions in .h5 and .xmf format, that can be used to restart the job. For instance, at time iteration 39, the exported restart files are: `fsiRestartF_039` (fluid), `fsiRestartS1_039` (membrane) and `fsiRestartS2_039` (magnet).

The restart procedure consists of two steps:

1. Define the settings of `dataFile_restart` as in `dataFile`. In addition, the user has to specify: i) the restart time iteration [importer/initTimeIter] (e.g. to 39), and ii) the corresponding initial time [time_discretization/initialtime] (in s).
2. restart the simulation by typing on the command line:

   ```
   mpirun -np 7 WMBP.exe -f dataFile_restart > benchmark_output_restart.txt
   ```

   This can be run either in serial or in parallel, as in Section 8.4.3. Check that the restart files are present in the working directory for both the restart time iteration AND the previous one, e.g. `fsiRestart*_039` and `fsiRestart*_038`.

Notice that for the post-processing of a simulation that required a restart, the results need to be extracted also from output file `benchmark_output_restart.txt`. Hence, the user has to run:

```
python extract_flowResults.py benchmark_output_restart.txt
```

and join the results together in a unique file `flowResults.txt`.

Part IX.
Coupled parameterized reduced order modelling of thermo-mechanical phenomena arising in blast furnace

Nirav Shah, Michele Girfoglio, Patricia Barral, Peregrina Quintela, Gianluigi Rozza, Alejandro Lengomin

Abstract

The benchmark cases related to coupled thermomechanical phenomena arising in blast furnace have been addressed by using open-source libraries. In this document, we provide details about the numerical implementation of the benchmark tests to verify, validate and reproduce the results of numerical experiments by aiming towards smooth transition for next developers and students interested in detailed investigation of our work.

Keywords: Blast furnace hearth, Thermo-mechanical axisymmetric model, Finite element method, Benchmark verification, Moder order reduction, Proper orthogonal decomposition.

Latest release: https://doi.org/10.5281/zenodo.5171821

GitHub repository: https://github.com/ROMSOC/benchmars-thermomechanical-model

9.1. Introduction

In our previous work [80], we focused on the physical problem, mathematical formulation, statement of the benchmark problems and design of numerical experiments. We now provide the code for numerical experiments, using open source libraries, with the objective of validating or reproducing the results and smooth handover of numerical software to future developers as per best practices [81].

9.2. Prerequisites

We use python 3.6.9 as the programming language. In this project we use the libraries :

- FEniCS 2019.1.0 ([82],[83],[84],[85], www.fenicsproject.org)
- RBniCS 0.1.dev1 ([86],www.rbnicsproject.org)
- Matplolib 3.1.2 ([16],www.matplotlib.org)
- numpy 1.17.4 ([87],www.numpy.org)

```
from dolfin import * #FEniCS library
from mshr import * #mshr - mesh generation component of FEniCS
from rbnics import * #RBniCS library
import matplotlib.pyplot as plt #Matplotlib library
import numpy as np #Numpy library
```

The solutions are stored in .pvd format, which can later be viewed with Paraview (www.paraview.org).

9.3. Installation

Simply clone the public repository:

```
$ git clone https://github.com/ROMSOC/benchmark_thermomechanical_model
```

9.4. Running the benchmark cases

Source codes for input data are provided in the folder *source_files*. Source codes for running the benchmark are provided in folder *source*. After running the benchmark, results are stored in folder *result_files*.

Run required .py file e.g. *file_name.py* as,

```
python3 file_name.py
```

9.5. Benchmark cases

9.5.1. Reading the mesh

We first construct the polygonal domain using the mshr tool.

```
# Define domain
domain = Polygon([Point(0.,0.),Point(7.05,0.),
      Point(7.05,7.265),Point(5.3,7.265),
      Point(5.3,4.065),Point(4.95,4.065),
      Point(4.95,3.565),Point(4.6,3.565),
      Point(4.6,2.965),Point(4.25,2.965),
      Point(4.25,2.365),Point(0.,2.365)])
```

The domain is divided into triangular 30 subdomains. We define the mapping from the reference domain to the parametrized domain. As an example, we take the subdomain 1 whose coordinates on reference domain are $(0,0),(4.25,0),(0,2.365)$. It is deformed to the parametrized subdomain with coordinates $(0,0),(\mu_6,2),(0,\mu_0)$. The μ_6 and μ_0 are the 6th and 0th parameter of zero-indexed tuple Ξ. This mapping can be defined as,

```
  {
    ("0", "0"): ("0", "0"),
    ("4.25", "0"): ("mu[6]/2", "0"),
    ("0", "2.365"): ("0", "mu[0]")
  }, # subdomain 1
```

The RBniCS computes the mapping for each subdomain and uses it during affine transformation of operators. Similarly, the mappings and subdomains are created for other 29 subdomains.

Next, we set the subdomain marker:

```
# Loop over all mappings and set subdomain markers
for i, vertices_mapping in enumerate(vertices_mappings):
  print(i,vertices_mapping.keys())
  subdomain_i = Polygon([Point(*[float(coord) for coord in vertex]) for vertex in
                                                counterclockwise(vertices_mapping.keys())])
  domain.set_subdomain(i + 1, subdomain_i)
```

The mesh and subdomains are created based on subdomain markers.

```
# Create mesh
mesh = generate_mesh(domain, 30) #30 specifies the mesh size.

# Create subdomains
subdomains = MeshFunction("size_t", mesh, 2, mesh.domains())
```

We now set the boundary markers. The domain boundaries are shared across 20 subdomains, hence we define 20 classes and set markers for each of these boundaries. For example, for the boundary γ_s, we define the class

```
class Gamma_s(SubDomain):
  def inside(self, x, on_boundary):
    return x[0] < DOLFIN_EPS and on_boundary
```

and create an instance of this class and set boundary marker as 1.

```
gamma_s = Gamma_s()
gamma_s.mark(boundaries, 1)
```

Unlike γ_s, the bottom boundary γ_- is shared by 5 subdomains, we define 5 different classes and correspondingly 5 different markers.

```
class Gamma_minus1(SubDomain):
  def inside(self, x, on_boundary):
    return x[1] < DOLFIN_EPS and x[0] < (4.255 + DOLFIN_EPS) and on_boundary

class Gamma_minus2(SubDomain):
  def inside(self, x, on_boundary):
    return x[1] < DOLFIN_EPS and x[0] > (4.2 - DOLFIN_EPS) and x[0] < (4.65 + DOLFIN_EPS)
                                          and on_boundary

class Gamma_minus3(SubDomain):
  def inside(self, x, on_boundary):
    return x[1] < DOLFIN_EPS and x[0] > (4.55 - DOLFIN_EPS) and x[0] < (5.05 + DOLFIN_EPS)
                                           and on_boundary

class Gamma_minus4(SubDomain):
  def inside(self, x, on_boundary):
    return x[1] < DOLFIN_EPS and x[0] > (4.9 - DOLFIN_EPS) and x[0] < (5.4 + DOLFIN_EPS)
                                          and on_boundary

class Gamma_minus5(SubDomain):
  def inside(self, x, on_boundary):
    return x[1] < DOLFIN_EPS and x[0] > (5.25 - DOLFIN_EPS) and x[0] < (7.15 + DOLFIN_EPS)
                                           and on_boundary


gamma_minus1 = Gamma_minus1()
gamma_minus1.mark(boundaries, 2)
gamma_minus2 = Gamma_minus2()
gamma_minus2.mark(boundaries, 3)
gamma_minus3 = Gamma_minus3()
gamma_minus3.mark(boundaries, 4)
gamma_minus4 = Gamma_minus4()
gamma_minus4.mark(boundaries, 5)
gamma_minus5 = Gamma_minus5()
gamma_minus5.mark(boundaries, 6)
```

In a similar manner, markers are set for other boundaries and the mesh, the boundary markers, the subdomain markers and affine maps are stored.

```
# Save mesh data
os.system("mkdir ../input_data/mesh_data")
VerticesMappingIO.save_file(vertices_mappings, ".", "../data_files/mesh_data/
                                                  hearth_vertices_mapping.vmp")
File("../data_files/mesh_data/hearth.xml") << mesh
File("../data_files/mesh_data/hearth_physical_region.xml") << subdomains
File("../data_files/mesh_data/hearth_facet_region.xml") << boundaries
XDMFFile("../data_files/mesh_data/hearth.xdmf").write(mesh)
XDMFFile("../data_files/mesh_data/hearth_physical_region.xdmf").write(subdomains)
XDMFFile("../data_files/mesh_data/hearth_facet_region.xdmf").write(boundaries)
File("../data_files/mesh_data/hearth.pvd") << mesh
File("../data_files/mesh_data/hearth_physical_region.pvd") << subdomains
File("../data_files/mesh_data/hearth_facet_region.pvd") << boundaries
```

At the beginning of any benchmark test, we first read the mesh, the subdomains and also read the boundary markers. The volume of each subdomains and length of each boundary are measured. Notice that all boundary markers with same boundary condition are combined.

```
# Read the mesh file from specified path.
mesh = Mesh("../../benchmarks/data_files/mesh_data/hearth.xml")
domains = MeshFunction('size_t',mesh,mesh.topology().dim()) # Read domain marker
subdomains = MeshFunction("size_t", mesh, "../../benchmarks/data_files/mesh_data/
                                           hearth_physical_region.xml") # Read
                                           subdomain markers
boundaries = MeshFunction("size_t", mesh, "../../benchmarks/data_files/mesh_data/
                                           hearth_facet_region.xml") # Read boundary
                                           markers
dx = Measure('dx', domain = mesh, subdomain_data = domains) # Volume measure
ds = Measure('ds', domain = mesh, subdomain_data = boundaries) # Boundary measure
n = as_vector(FacetNormal(mesh)) # Edge unit normal vector

d_bottom = ds(2) + ds(3) + ds(4) + ds(5) + ds(6) # Markers of bottom boundary \gamma_{-}
d_out = ds(7) + ds(8) + ds(9) + ds(10) + ds(11) # Markers of outer boundary \gamma_{out}
d_sf = ds(13) + ds(14) + ds(15) + ds(16) + ds(17) + ds(18) + ds(19) + ds(20) # Markers of
                                            inner boundary \gamma_{sf}
```

9.5.2. Thermal model

We define the norm for $\psi \in H^1_r(\omega)$,

```
#Computation of H^1_r(\omega) norm
def compute_h1r_norm(psi,mesh):
  r = SpatialCoordinate(mesh)[0]
  dx = Measure('dx', domain = mesh)
  a = inner(psi,psi)*r*dx + inner(grad(psi),grad(psi))*r*dx
  A = assemble(a)
  return sqrt(A)
```

Next, we define the range of polynomials degree for measuring $p-$convergence and an object to store the relative error.

```
error_T_vector = [] #List to store error in temperature w.r.t. polynomial degree
p = range(1,4) # List of polynomial degrees
```

Next, we specify the relevant physical parameters, which are than later used in weak form.

```
k = 10. # Thermal conductivity
h_fluid = 200. # Convection coefficient on \gamma_{sf}
h_right = 2000. # Convection coefficient on \gamma_{out}
h_bottom = 2000. # Convection coefficient on \gamma_{-}
```

For every polynomial degree, we define the relevant "Lagrange" function space of a particular degree. We also define the solution field and test function in this space.

```
# Define function space
VT = FunctionSpace(mesh,"CG",i) # Function space for temperature
psi, T_ = TestFunction(VT), TrialFunction(VT) # Evaluate trial and test function
T = Function(VT, name = "temperature increase")
x = list()
x.append(Expression("x[0]", element=VT.ufl_element())) #r coordinate
x.append(Expression("x[1]", element=VT.ufl_element())) #y coordinate
```

We then define and solve the equation in weak formulation.

```
# solving weak form of energy equation
a_T = k * inner(grad(psi),grad(T_)) * x[0] * dx + \
  h_fluid * psi * T_ * x[0] * d_sf + h_right * psi * T_ * x[0] * d_out + \
  h_bottom * psi * T_ * x[0] * d_bottom # Bilinear side
l_T = h_fluid * psi * (x[0] * x[0] * x[1] + k/h_fluid * ( 2 * x[0] * x[1] * n[0] + x[0] *
                                                    x[0] * n[1] ) ) * x[0] * d_sf + \
  h_right * psi * (x[0]*x[0]*x[1]+2*x[0]*x[1]*k/h_right) * x[0] * d_out + \
  h_bottom * psi * (x[0] * x[0] * x[1] - x[0] * x[0] * k / h_bottom) * x[0] * d_bottom + \
  -4 * k * x[1] * psi * x[0] * dx + psi * k * x[0] * x[0] * x[0] * ds(12) # Linear side
solve(a_T == l_T, T) # Solve the variational form
```

After performing the computations, we store the data in format compatible with paraview for further visualization. Also, we plot the p-convergence.

```
# Plotting and visualization
File("../../benchmarks/result_files/thermal_model/temperature_computed.pvd") << T
File("../../benchmarks/result_files/thermal_model/temperature_analytical.pvd") <<
                                                    T_analytical
error_temperature = Function(VT) #Function for Spatial distribution of temperature
                                                    absolute error
error_temperature.vector()[:] = abs(T_analytical.vector().get_local() - T.vector().
                                                    get_local())
File("../../benchmarks/result_files/thermal_model/temperature_absolute_error.pvd") <<
                                                    error_temperature

# Plotting and printing convergence tests
plt.figure(figsize=[10,8])
a = plt.semilogy([1,2,3],error_T_vector,marker='o',linewidth=4)
plt.xticks([1,2,3],fontsize=18)
plt.yticks(fontsize=18)
plt.xlabel('Polynomial degree',fontsize=24)
plt.ylabel('Relative error',fontsize=24)
plt.axis('tight')
plt.savefig("../../benchmarks/result_files/thermal_model/convergence_test")
plt.show()
```

9.5.3. Mechanical model

We define the norm for $\overrightarrow{\phi} \in \mathbb{U}$.

```
#Computation of \mathbb{U} norm
def compute_U_norm(phi,mesh):
  x = SpatialCoordinate(mesh)
  dx = Measure('dx', domain = mesh)
  a = inner(phi,phi)*x[0]*dx + inner(grad(phi),grad(phi))*x[0]*dx + (phi[0]**2)/x[0]*dx
  A = assemble(a)
  return sqrt(A)
```

Next, we define the axisymmetric stress and strain tensor.

```
# Axisymmetric strain tensor definition. Alternative could be to express strain as vector
                                                    using Voigt notation.
def eps(u):
  return \
    sym(as_tensor([[u[0].dx(0), u[0].dx(1), 0. ],\
    [u[1].dx(0), u[1].dx(1), 0.],\
    [0., 0., u[0]/x[0]]]))
```

```
# Axisymmetric stress tensor definition. Alternative could be to express stress as vector
                                                    using Voigt notation.
def sigma(u):
  return lmbda * tr(eps(u)) * Identity(3) + 2.0 * mu * eps(u)
```

Since, our aim is to assess $p-$convergence, we define the range of polynomial degrees and measure corresponding relative errors.

```
error_u_vector = [] #List for absolute error in displacement
p = range(1,4) #Polynomial degrees
```

We introduce the physical properties.

```
E = Constant(5e9) # Young's modulus
nu = Constant(0.2) # Poisson's ratio
mu = E/2/(1+nu) # Lam\'e parameter
lmbda = E*nu/(1+nu)/(1-2*nu) # Lam\'e parameter
```

We now define the function space for displacement and stress. We also initialize variables for the test function and the solution field.

```
# Define function space
VM = VectorFunctionSpace(mesh,"CG",i) # Function space for displacement
x = Expression(("x[0]","x[1]"), element=VM.ufl_element())
VS = FunctionSpace(mesh,"CG",max(i-1,1))
# Function space for Von Mises stress NOTE: when i=1, the VS is of degree 1 and not 0.
phi, u_ = TestFunction(VM), TrialFunction(VM)
u = Function(VM, name = "Displacement") # u[0] = u_r and u[1] = u_y
```

We define the variables related to the source term and the boundary data.

```
# Dirichlet boundary data
bcs_M = [DirichletBC( VM.sub(0), Constant(0.), 'x[0] < DOLFIN_EPS and on_boundary'), \
  DirichletBC( VM.sub(1), Constant(0.), 'near(x[1],0) and on_boundary')]
# Set Dirichlet boundary. Note that only functions which satisfy zero normal displacement
                                                    on \gamma_s \cup \gamma_- are admissible.

# Source term and relevant boundary data
f0_r = - (2*E*nu*1e-4*x[0]/(1-2*nu)/(1+nu)+2*E*1e-4*x[0]/(1+nu))
f0_y = - (4*E*1e-4*x[1]/(1+nu)+4*E*1e-4*x[1]*nu/(1-2*nu)/(1+nu))

g_plus_r = 2*E*1e-4*x[0]*x[1]/(1+nu)
g_plus_y = E / (1-2*nu) / (1+nu) * (2*nu*1e-4*x[1]*x[1]+(1-nu)*1e-4*x[0]*x[0])

g_minus_r = -g_plus_r

g_sf_r = E / (1-2*nu) / (1+nu) * (1e-4 * x[1] * x[1] + nu * 1e-4 * x[0] * x[0]) * n[0] +
                                                    2 * E * 1e-4 * x[0] * x[1] / (1 + nu) * n[1]
g_sf_y = 2 * E * 1e-4 * x[0] * x[1] / (1 + nu) * n[0] + E / (1-2*nu) / (1+nu) * (2 * nu *
                                                    1e-4 * x[1] * x[1] + (1 - nu) * 1e-4 * x[0]
                                                    * x[0]) * n[1]

g_out_r = E / (1-2*nu) / (1+nu) * (1e-4 * x[1] * x[1] + nu * 1e-4 * x[0] * x[0])
g_out_y = 2*E*1e-4*x[0]*x[1]/(1+nu)
```

The equation in weak form is then defined and solved for computing displacement. Based on this displacement, the Von Mises stress is computed.

```
# solving weak form of momentum equation
# Bilinear form
```

```
a_M = inner(sigma(u_),eps(phi)) * x[0] * dx
#linear form
l_M = (phi[0] * f0_r + phi[1] * f0_y) * x[0] * dx + (phi[0] * g_plus_r + phi[1] *
                                                    g_plus_y) * x[0] * ds(12) + \
(phi[0] * g_minus_r) * x[0] * d_bottom + (phi[0] * g_sf_r + phi[1] * g_sf_y) * x[0] *
                                                    d_sf + \
(phi[0] * g_out_r + phi[1] * g_out_y) * x[0] * d_out
solve(a_M == l_M, u, bcs_M)


# Von Mises stress computed displacement
sigma_dev = sigma(u) - tr(sigma(u)) / 3 * Identity(3)
sigma_vm = sqrt(3 * inner( sigma_dev, sigma_dev) / 2) # Von mises stress
# Von Mises stress analytical displacement
sigma_dev_analytical = sigma(u_analytical) - tr(sigma(u_analytical)) / 3 * Identity(3)
sigma_vm_analytical = sqrt(3 * inner( sigma_dev_analytical, sigma_dev_analytical) / 2) #
                                                    Von mises stress

# Compute H^1_r norm of error
error_u = compute_U_norm(u_analytical-u,mesh)/compute_U_norm(u_analytical,mesh)
error_u_vector.append(error_u)
print("Relative error in U-norm : ",str(error_u))
```

The data is stored in format compatible to paraview. We also plot the polynomial degree vs the relative error.

```
# Post-processing and visualization
File("../../benchmarks/result_files/mechanical_model/displacement_computed.pvd") << u
File("../../benchmarks/result_files/mechanical_model/displacement_analytical.pvd") <<
                                                    u_analytical
File("../../benchmarks/result_files/mechanical_model/displacement_error.pvd") << project(
                                                    u_analytical-u,VM)

error_stress = Function(VS) #Function for absolute error in stress tensor
error_stress.vector()[:] = abs(project(sigma_vm,VS).vector().get_local() - project(
                                                    sigma_vm_analytical,VS).vector().get_local()
                                                    )
File("../../benchmarks/result_files/mechanical_model/von_mises_stress_computed.pvd") <<
                                                    project(sigma_vm,VS)
File("../../benchmarks/result_files/mechanical_model/von_mises_stress_analytical.pvd") <<
                                                     project(sigma_vm_analytical,VS)
File("../../benchmarks/result_files/mechanical_model/von_mises_stress_error.pvd") <<
                                                    project(error_stress,VS)

#Convergence tests
plt.figure(figsize=[10,8])
a = plt.semilogy([1,2,3],error_u_vector,marker='o',linewidth=4)
plt.xticks([1,2,3],fontsize=18)
plt.yticks(fontsize=18)
plt.xlabel('Polynomial degree',fontsize=24)
plt.ylabel('Relative error',fontsize=24)
plt.axis('tight')
plt.savefig("../../benchmarks/result_files/mechanical_model/convergence_test")
plt.show() # To show the plots

print("Relative error in U norm: "+ str(error_u_vector))
```

9.5.4. Coupled model

Similar to the Thermal model (Section 9.5.2), we first solve the energy equation in weak form.

```
# Define function space
VT = FunctionSpace(mesh,"CG",3) # Function space for temperature
psi, T_ = TestFunction(VT), TrialFunction(VT) # Evaluate trial and test function
T = Function(VT, name = "temperature increase")

# Known analytical solution, Thermal material properties and Boundary data
T_analytical = Expression('x[0]*x[0]*x[1]',degree = 3)
VT_analytical = FunctionSpace(mesh,"CG",3) #Space for analytical solution
T_analytical = project(T_analytical,VT_analytical)
k = 10. # Thermal conductivity
h_fluid = 200. # Convection coefficient on \gamma_{sf}
h_right = 2000. # Convection coefficient on \gamma_{out}
h_bottom = 2000. # Convection coefficient on \gamma_{-}
x = list()
x.append(Expression("x[0]", element=VT.ufl_element())) #r coordinate
x.append(Expression("x[1]", element=VT.ufl_element())) #y coordinate

# solving weak form of energy equation
a_T = k * inner(grad(psi),grad(T_)) * x[0] * dx + \
  h_fluid * psi * T_ * x[0] * d_sf + h_right * psi * T_ * x[0] * d_out + \
  h_bottom * psi * T_ * x[0] * d_bottom # Bilinear side
l_T = h_fluid * psi * (x[0] * x[0] * x[1] + k/h_fluid * ( 2 * x[0] * x[1] * n[0] + x[0] *
                                                    x[0] * n[1] ) ) * x[0] * d_sf + \
  h_right * psi * (x[0]*x[0]*x[1]+2*x[0]*x[1]*k/h_right) * x[0] * d_out + \
  h_bottom * psi * (x[0] * x[0] * x[1] - x[0] * x[0] * k / h_bottom) * x[0] * d_bottom + \
  -4 * k * x[1] * psi * x[0] * dx + psi * k * x[0] * x[0] * x[0] * ds(12) # Linear side
solve(a_T == l_T, T) # Solve the variational form
```

Also, similar to mechanical model (section 9.5.3) , we define the $\mathbb{U}$ norm, stress and strain tensor. Additionally, we define the thermomechanical stress tensor.

```
# Define \mathbb{U} norm
def compute_U_norm(phi,mesh):
  x = SpatialCoordinate(mesh)
  a = inner(phi,phi)*x[0]*dx + inner(grad(phi),grad(phi))*x[0]*dx + (phi[0]**2/x[0])*dx
  A = assemble(a)
  return sqrt(A)

# Axisymmetric strain tensor definition. Alternative could be to express strain as vector
                                                    using Voigt notation.
def eps(u):
  return \
    sym(as_tensor([[u[0].dx(0), u[0].dx(1), 0. ],\
    [u[1].dx(0), u[1].dx(1), 0.],\
    [0., 0., u[0]/x[0]]]))

# Axisymmetric thermo-mechanical stress tensor definition. Alternative could be to
                                                express as vector using Voigt notation.
def sigma(u,T):
  return lmbda * tr(eps(u)) * Identity(3) + 2.0 * mu * eps(u) - (2 * mu + 3 * lmbda) *
                                                alpha * (T - T_0) * Identity(3)

# Axisymmetric mechanical stress tensor definition. Alternative could be to express as
                                                vector using Voigt notation.
def sigma2(u):
  return lmbda * tr(eps(u)) * Identity(3) + 2.0 * mu * eps(u)
```

Next, the physical data are specified.

```
T_0 = 298 # Reference temperature for zero thermal stress
E = Constant(5e9) # Young's modulus
```

```
nu = Constant(0.2) # Poisson's ratio
mu = E/2/(1+nu) # Lame\'e parameter
lmbda = E*nu/(1+nu)/(1-2*nu) # Lame\'e parameter
alpha = Constant(1e-6) # Thermal expansion coefficient
```

Similar to the mechanical model (section 9.5.3), we solve the weak form.

```
# Define function space for displacement
VM = VectorFunctionSpace(mesh,"CG",i) # Function space for displacement
x = Expression(("x[0]","x[1]"), element=VM.ufl_element())
phi, u_ = TestFunction(VM), TrialFunction(VM)
u = Function(VM, name = "Displacement") # u[0] = u_r and u[1] = u_y
VS = FunctionSpace(mesh,"CG",max(i-1,1)) # Function space for shear component of stress

# Dirichlet boundary data
bcs_M = [DirichletBC( VM.sub(0), Constant(0.), 'x[0] < DOLFIN_EPS and on_boundary'),
                                                DirichletBC( VM.sub(1), Constant(0.), 'near(
                                                x[1],0) and on_boundary')]

#Boundary and source terms
f0_r = - (2*E*nu*1e-4*x[0]/(1-2*nu)/(1+nu)+2*E*1e-4*x[0]/(1+nu)-2*E*x[0]*x[1]*alpha/(1-2*
                                                nu))
f0_y = - (4*E*1e-4*x[1]/(1+nu)+4*E*1e-4*x[1]*nu/(1-2*nu)/(1+nu)-E*x[0]*x[0]*alpha/(1-2*nu
                                                ))

g_plus_r = 2*E*1e-4*x[0]*x[1]/(1+nu)
g_plus_y = E / (1-2*nu) / (1+nu) * (2*nu*1e-4*x[1]*x[1]+(1-nu)*1e-4*x[0]*x[0]) - E*alpha/
                                                (1-2*nu)*(x[0]*x[0]*x[1] - T_0)

g_minus_r = -g_plus_r

g_sf_r = (E / (1-2*nu) / (1+nu) * (1e-4 * x[1] * x[1] + nu * 1e-4 * x[0] * x[0]) - E*
                                                alpha/(1-2*nu)*(x[0]*x[0]*x[1] - T_0)) * n[0
                                                ] + 2 * E * 1e-4 * x[0] * x[1] / (1 + nu) *
                                                n[1]
g_sf_y = 2 * E * 1e-4 * x[0] * x[1] / (1 + nu) * n[0] + (E / (1-2*nu) / (1+nu) * (2 * nu
                                                * 1e-4 * x[1] * x[1] + (1 - nu) * 1e-4 * x[0
                                                ] * x[0])- E*alpha/(1-2*nu)*(x[0]*x[0]*x[1]
                                                - T_0)) * n[1]

g_out_r = E / (1-2*nu) / (1+nu) * (1e-4 * x[1] * x[1] + nu * 1e-4 * x[0] * x[0]) - E*
                                                alpha/(1-2*nu)*(x[0]*x[0]*x[1] - T_0)
g_out_y = 2*E*1e-4*x[0]*x[1]/(1+nu)

# solving weak form of momentum equation
# This is not bilinear side as terms related to thermal stress are included.
a_M1 = inner(sigma(u_,T),eps(phi)) * x[0] * dx
# This is not linear side as terms related to thermal stress are not included.
l_M1 = (phi[0] * f0_r + phi[1] * f0_y) * x[0] * dx + (phi[0] * g_plus_r + phi[1] *
                                                g_plus_y) * x[0] * ds(12) + \
(phi[0] * g_minus_r) * x[0] * d_bottom + (phi[0] * g_sf_r + phi[1] * g_sf_y) * x[0] *
                                                d_sf + \
(phi[0] * g_out_r + phi[1] * g_out_y) * x[0] * d_out
F = a_M1 - l_M1
a_M = lhs(F) # Now a_M is bilinear form
l_M = rhs(F) # Now l_M is linear form
solve(a_M == l_M, u, bcs_M) # Solve equation

# Compute \mathbb{U} norm of error
error_u = compute_U_norm(u_analytical-u,mesh)/compute_U_norm(u_analytical,mesh)
error_u_vector.append(error_u)
```

```
print("Relative error in U-norm : ",str(error_u))
```

We compute the relevant stress fields.

```
# Von Mises stress for computed displacement
sigma_dev = sigma(u,T) - tr(sigma(u,T)) / 3 * Identity(3)
sigma_vm = sqrt(3 * inner( sigma_dev, sigma_dev) / 2) # Von mises stress
# Von Mises stress for analytical displacement
sigma_dev_analytical = sigma(u_analytical,T) - tr(sigma(u_analytical,T)) / 3 * Identity(3
                                                )
sigma_vm_analytical = sqrt(3 * inner( sigma_dev_analytical, sigma_dev_analytical) / 2) #
                                                Von mises stress
# Spherical stress for computed displacement
sigma_spherical = tr(sigma(u,T)) / 3
# Spherical stress for analytical displacement
sigma_spherical_analytical = tr(sigma(u_analytical,T)) / 3
# Spherical mechanical stress for computed displacement
sigma_spherical_non_thermal = tr(sigma2(u)) / 3
```

Finally, we store the solution field for further visualization.

```
# Post-processing and visualization
File("../../benchmarks/result_files/coupled_model/Temperature_computed.pvd") << T
File("../../benchmarks/result_files/coupled_model/Temperature_analytical.pvd") <<
                                                T_analytical
File("../../benchmarks/result_files/coupled_model/Teperature_error.pvd") << project(T-
                                                T_analytical,VT)

File("../../benchmarks/result_files/coupled_model/displacement_computed.pvd") << u
File("../../benchmarks/result_files/coupled_model/displacement_analytical.pvd") <<
                                                u_analytical
File("../../benchmarks/result_files/coupled_model/displacement_absolute_error.pvd") <<
                                                project(u_analytical - u,VM)

File("../../benchmarks/result_files/coupled_model/von_mises_computed_coupling.pvd") <<
                                                project(sigma_vm,VS)
File("../../benchmarks/result_files/coupled_model/von_mises_analytical_coupling.pvd") <<
                                                project(sigma_vm_analytical,VS)
error_stress_von_mises = Function(VS)
error_stress_von_mises.vector()[:] = abs(project(sigma_vm,VS).vector().get_local() -
                                                project(sigma_vm_analytical,VS).vector().
                                                get_local())
File("../../benchmarks/result_files/coupled_model/von_mises_stress_error_coupling.pvd") <
                                                < project(error_stress_von_mises,VS)

File("../../benchmarks/result_files/coupled_model/difference_in_spherical_stress.pvd") <<
                                                 project(sigma_spherical -
                                                sigma_spherical_non_thermal,VS)
File("../../benchmarks/result_files/coupled_model/thermal_part_of_stress.pvd") << project
                                                (-(2 * mu + 3 * lmbda) * alpha * (T - T_0),
                                                VS)
error_stress_spherical = Function(VS)
error_stress_spherical.vector()[:] = abs(project(sigma_spherical,VS).vector().get_local()
                                                 - project(sigma_spherical_non_thermal - (2
                                                * mu + 3 * lmbda) * alpha * (T - T_0),VS).
                                                vector().get_local())
File("../../benchmarks/result_files/coupled_model/absolute_error_spherical_stress.pvd") <
                                                < error_stress_spherical

#Convergence tests
plt.figure(figsize=[10,8])
```

```
a = plt.semilogy([1,2,3],error_u_vector,marker='o',linewidth=4)
plt.xticks([1,2,3],fontsize=18)
plt.yticks(fontsize=18)
plt.xlabel('Polynomial degree',fontsize=24)
plt.ylabel('Relative error',fontsize=24)
plt.axis('tight')
plt.savefig('../../benchmarks/result_files/coupled_model/
                                                Convergence_coupling_displacement_benchmark_comparison
                                                .png')
plt.show()
```

9.5.5. Reduced basis method

For the affine geometric parametrization, we use 2 decorators : one for the affine shape parametrization and the other for transfer of operators between reference domain and parametrized domain.

```
@PullBackFormsToReferenceDomain() #Decorator for operator transformation between
                                                parameterized domain to reference domain
@AffineShapeParametrization("../../benchmarks/data_files/mesh_data/
                                                hearth_vertices_mapping.vmp") #Decorator for
                                                 shape parametrization with mapping defined
                                                in specified file
```

To compute the temperature field required to compute displacement for coupling model, we use another decorator :

```
@ExactParametrizedFunctions() #Decorator for computing temperature field required for
                                                linear side
```

Considering that the solution computed by finite element method is used as benchmark for assessing the accuracy of the reduced basis method, we refer to the solution computed by finite element method as “Truth solution”.

9.5.5.1. Thermal system

We first define the class *HearthThermal*, inherited from *EllipticCoerciveProblem*, for the thermal system.

```
class HearthThermal(EllipticCoerciveProblem):
```

The default initialization involves all the parameters related to the problem.

```
  # Default initialization of members
  def __init__(self, V, **kwargs):
    # Call the standard initialization
    EllipticCoerciveProblem.__init__(self, V, **kwargs)
    # ... and also store FEniCS data structures for assembly
    assert "subdomains" in kwargs
    assert "boundaries" in kwargs
    assert "mesh" in kwargs
    assert "h_cf" in kwargs
    assert "h_out" in kwargs
    assert "h_bottom" in kwargs
    self.subdomains, self.boundaries = kwargs["subdomains"], kwargs["boundaries"]
    self.u = TrialFunction(V)
    self.v = TestFunction(V)
    self.dx = Measure("dx")(subdomain_data=subdomains)
    self.ds = Measure("ds")(subdomain_data=boundaries)
    self.subdomains = subdomains
```

```
    self.boundaries = boundaries
    self.reference_mesh = kwargs["mesh"]
    self.h_cf = kwargs["h_cf"]
    self.h_out = kwargs["h_out"]
    self.h_bottom = kwargs["h_bottom"]
    self.x0 = Expression("x[0]", element=V.ufl_element())
```

Firstly, the affine mulitplicative terms and next the weak formulation are defined.

```
  # Return theta multiplicative terms of the affine expansion of the problem.
  def compute_theta(self, term):
    mu = self.mu
    if term == "a":
      theta_a0 = mu[10]
      theta_a1 = 1.0
      return (theta_a0, theta_a1)
    elif term == "f":
      theta_f0 = 1.0
      return (theta_f0, )
    else:
      raise ValueError("Invalid term for compute_theta().")

  # Return forms resulting from the discretization of the affine expansion of the problem
                                                        operators.
  def assemble_operator(self, term):
    u = self.u
    v = self.v
    reference_mesh = self.reference_mesh
    dx = self.dx
    ds = self.ds
    h_cf = self.h_cf
    h_out = self.h_out
    h_bottom = self.h_bottom
    r = self.x0
    d_bottom = ds(2) + ds(3) + ds(4) + ds(5) + ds(6)
    d_out = ds(7) + ds(8) + ds(9) + ds(10) + ds(11)
    d_sf = ds(13) + ds(14) + ds(15) + ds(16) + ds(17) + ds(18) + ds(19) + ds(20)
    if term == "a":
      a0 = inner(grad(u), grad(v))*r*dx
      a1 = h_bottom*u*v*r*d_bottom + h_out*u*v*r*d_out + h_cf*u*v*r*d_sf
      return (a0, a1)
    elif term == "f":
      f0 = h_bottom*313*v*r*d_bottom + h_out*313*v*r*d_out + h_cf*1773*v*r*d_sf
      return (f0, )
    elif term == "inner_product":
      x0 = u*v*r*dx + inner(grad(u), grad(v))*r*dx
      return (x0,)
    else:
      raise ValueError("Invalid term for assemble_operator().")
```

Using the *HearthThermal* class, we now perform POD-Galerkin approximation of the thermal problem. The function space is defined first. Next, an instance of *HearthThermal* class, *hearth_problem_thermal* is created and model order reduction is performed.

```
# 2A. Create Finite Element space (Lagrange P1)
VT = FunctionSpace(mesh, "Lagrange", 1) # For temperature

# 3A. Allocate an object of the Hearth class
hearth_problem_thermal = HearthThermal(VT, subdomains=subdomains, boundaries=boundaries,
                                                    mesh=mesh, h_cf=200., h_out=2000., h_bottom=
```

```
                                                2000.)
#specify and set range of each parameter
mu_range = [(2.3,2.4), (0.5,0.7), (0.5,0.7), (0.4,0.6), (3.05,3.35), (13.5,14.5), (8.3,8.
                                                7), (8.8,9.2), (9.8,10.2), (10.4,10.8), (9.8
                                                ,10.2), (2.08e9,2.08e9), (1.39e9,1.39e9), (
                                                1e-6,1e-6)]
hearth_problem_thermal.set_mu_range(mu_range)

# 4A. Prepare reduction with a POD-Galerkin method
#NOTE : truth_problem attribute is FEM problem and reduced_problem is RB problem
pod_galerkin_method_thermal = PODGalerkin(hearth_problem_thermal)
pod_galerkin_method_thermal.set_Nmax(100) #Maximum size of reduced basis space
pod_galerkin_method_thermal.set_tolerance(1e-4) #Maximum eigenvalue tolerance
```

Using *pod_galerkin_method_thermal*, we perform the offline phase for the thermal system.

```
# 5A. Perform the offline phase
pod_galerkin_method_thermal.initialize_training_set(1000) #Initialize training set with
                                                specified number of training parameters
reduced_hearth_problem_thermal = pod_galerkin_method_thermal.offline() #Perform offline
                                                phase
```

Next, we perform the error analysis, compute the time taken for truth solution and reduced basis solution.

```
# 7A. Perform an error analysis
pod_galerkin_method_thermal.initialize_testing_set(50) #Initialize error analysis with
                                                specified number of parameters
pod_galerkin_method_thermal.error_analysis() #Perform error analysis

# 8A1. Perform a speedup analysis - Compute time for truth solutions
pod_galerkin_method_thermal.initialize_testing_set(50) #Initialize truth time computation
                                                 with specified number of parameters
testing_set_speedup_analysis = pod_galerkin_method_thermal.testing_set

pod_galerkin_method_thermal._patch_truth_solve(True) #To enable cahce reading

truth_timer = Timer("parallel") #Timer for computation of FEM solution
time_thermal_truth = np.empty(len(testing_set_speedup_analysis)) #Storage of time taken
                                                for solving FEM equation. It is a vector of
                                                size of number of speedup analysis
                                                parameters

# Iteration over speedup analysis parameters for measuring time taken for FEM solution
for (mu_index, mu_test) in enumerate(testing_set_speedup_analysis):
  print(TextLine(str(mu_index), fill="#"))
  pod_galerkin_method_thermal.truth_problem.set_mu(mu_test) #Set the parameter
  truth_timer.start()
  pod_galerkin_method_thermal.truth_problem.solve() #Solve the FEM problem
  truth_time_thermal = truth_timer.stop()
  print("Truth time thermal : ",truth_time_thermal)
  time_thermal_truth[mu_index] = truth_time_thermal #Save time taken for truth solve

np.save("time_thermal_truth",time_thermal_truth) #Save time taken for computation of FEM
                                                solution

pod_galerkin_method_thermal._undo_patch_truth_solve(True) #To enable cache reading

# 8A2. Perform a speedup analysis - Compute time for reduced solutions
pod_galerkin_method_thermal._patch_truth_solve(True) #To disable cache reading
reduced_timer = Timer("serial") #Timer for computation of RB solution
max_basis_function = reduced_hearth_problem_thermal.N #Size of reduced basis space
```

```
time_thermal_reduced = np.empty([max_basis_function,len(testing_set_speedup_analysis)]) #
                                                Storage of time taken for solving RB
                                                equation. It is a matrix of size size of
                                                reduced basis space \times number of speedup
                                                 analysis parameters

# Iteration over speedup analysis parameters for measuring time for RB solution
for basis_size in range(1,max_basis_function+1):
  for (mu_index, mu_test) in enumerate(testing_set_speedup_analysis):
    print(TextLine(str(mu_index), fill="#"))
    pod_galerkin_method_thermal.reduced_problem.set_mu(mu_test) #Set parameter
    reduced_timer.start()
    pod_galerkin_method_thermal.reduced_problem.solve(basis_size) #Solve the RB problem
    rb_time_thermal = reduced_timer.stop()
    print("Reduced time thermal : ",rb_time_thermal)
    time_thermal_reduced[basis_size-1,mu_index] = rb_time_thermal #Save time taken for RB
                                                solve

pod_galerkin_method_thermal._undo_patch_truth_solve(True) #To disable cache reading

np.save("time_thermal_reduced",time_thermal_reduced) #Save time taken for computation for
                                                 RB solution
```

For any new parameter *online_mu*, the truth solution and the reduced basis solution can be performed using these classes.

```
pod_galerkin_method_thermal.reduced_problem.set_mu(online_mu) #Set parameter
T_rb = pod_galerkin_method_thermal.reduced_problem.solve() #Reduced problem solve
pod_galerkin_method_thermal.reduced_problem.export_solution(filename="
                                                reference_domain_thermal_rb") #Save solution
                                                 for visualization with paraview
T_rb = pod_galerkin_method_thermal.reduced_problem.basis_functions * T_rb #RB solution
                                                projected back to FEM space
pod_galerkin_method_thermal.truth_problem.set_mu(online_mu) #Set parameter
T = pod_galerkin_method_thermal.truth_problem.solve() #FEM problem solve
pod_galerkin_method_thermal.truth_problem.export_solution(filename="reference_domain_fem"
                                                ) #Save solution for visualization with
                                                paraview
pod_galerkin_method_thermal.truth_problem.mesh_motion.move_mesh() #Deform mesh as per
                                                geometric parameters
File("HearthThermal/reference_domain_thermal_spatial_error.pvd") << project(T-T_rb,VT) #
                                                Spatial error
pod_galerkin_method_thermal.truth_problem.mesh_motion.reset_reference() #Restore mesh to
                                                reference configuration
```

9.5.5.2. Mechanical system

We first define the class *HearthMechanical*, inherited from *EllipticCoerciveProblem*, for the mechanical system.

```
class HearthMechanical(EllipticCoerciveProblem):
```

We specify the default initialization for this class.

```
  # Default initialization of members
  def __init__(self, V, **kwargs):
    # Call the standard initialization
    EllipticCoerciveProblem.__init__(self, V, **kwargs)
    # ... and also store FEniCS data structures for assembly
```

```
    assert "subdomains" in kwargs
    assert "boundaries" in kwargs
    assert "mesh" in kwargs
    self.normal = as_vector(FacetNormal(kwargs["mesh"]))
    self.subdomains, self.boundaries = kwargs["subdomains"], kwargs["boundaries"]
    self.u = TrialFunction(V)
    self.v = TestFunction(V)
    self.dx = Measure("dx")(subdomain_data=subdomains)
    self.ds = Measure("ds")(subdomain_data=boundaries)
    self.subdomains = subdomains
    self.boundaries = boundaries
    self.x0 = Expression("x[0]", element=V.sub(0).ufl_element())
    self.x1 = Expression("x[1]", element=V.sub(1).ufl_element())
```

Next, the affine multiplicative terms, weak forms and strain tensors are defined.

```
  # Return theta multiplicative terms of the affine expansion of the problem.
  def compute_theta(self, term):
    mu = self.mu
    if term == "a":
      theta_a0 = mu[11]
      theta_a1 = 2*mu[12]
      return (theta_a0, theta_a1, )
    elif term == "f":
      theta_f0 = 1.0
      return (theta_f0, )
    else:
      raise ValueError("Invalid term for compute_theta().")

  # Return strain tensor
  def strain(self,u):
    r = self.x0
    return sym(as_tensor([[u[0].dx(0), u[0].dx(1), 0. ], [u[1].dx(0), u[1].dx(1), 0.], [0
                                                  ., 0., u[0]/r]]))

  # Return forms resulting from the discretization of the affine expansion of the problem
                                                  operators.
  def assemble_operator(self, term):
    u = self.u
    v = self.v
    dx = self.dx
    ds = self.ds
    r = self.x0
    x1 = self.x1
    n = self.normal
    d_bottom = ds(2) + ds(3) + ds(4) + ds(5) + ds(6)
    d_out = ds(7) + ds(8) + ds(9) + ds(10) + ds(11)
    d_sf = ds(13) + ds(14) + ds(15) + ds(16) + ds(17) + ds(18) + ds(19) + ds(20)
    if term == "a":
      a0 = (u[0].dx(0)+u[1].dx(1)+u[0]/r)*(v[0].dx(0)+v[1].dx(1)+v[0]/r)*r*dx
      a1 = (u[0].dx(0)*v[0].dx(0) + u[1].dx(1)*v[1].dx(1) + (u[0]*v[0])/(r)**2 + 0.5*(u[0].
                                                  dx(1)+u[1].dx(0))*(v[0].dx(1)+v[1].dx(0))) *
                                                   r * dx
      return (a0, a1,)
    elif term == "f":
      f0 = - dot( v, 7460*9.81*(7.265-x1)*n) * r * d_sf
      return (f0,)
    elif term == "inner_product":
      x0 = inner(u,v) * r * dx + inner(self.strain(u),self.strain(v)) * r * dx
      return (x0,)
    elif term == "dirichlet_bc":
```

```
    bc0 = [DirichletBC(self.V.sub(0), Constant(0.), self.boundaries, 1),
      DirichletBC(self.V.sub(1), Constant(0.), self.boundaries, 2),
      DirichletBC(self.V.sub(1), Constant(0.), self.boundaries, 3),
      DirichletBC(self.V.sub(1), Constant(0.), self.boundaries, 4),
      DirichletBC(self.V.sub(1), Constant(0.), self.boundaries, 5),
      DirichletBC(self.V.sub(1), Constant(0.), self.boundaries, 6),]
    return (bc0,)
  else:
    raise ValueError("Invalid term for assemble_operator().")
```

The function space and an instance *hearth_problem_mechanical* of *HearthMechanical* class are defined.

```
# 2B. Create Finite Element space (Lagrange P1)
VM = VectorFunctionSpace(mesh,"Lagrange",1) # For mechanical

# 3B. Allocate an object of the HearthThermoMechanical class
hearth_problem_mechanical = HearthMechanical(VM, subdomains=subdomains, boundaries=
                                             boundaries, mesh=mesh)
#specify and set range of each parameter
mu_range = [(2.3,2.4), (0.5,0.7), (0.5,0.7), (0.4,0.6), (3.05,3.35), (13.5,14.5), (8.3,8.
                                             7), (8.8,9.2), (9.8,10.2), (10.4,10.8), (10
                                             .,10.), (1.9e9,2.5e9), (1.2e9,1.8e9), (1e-6,
                                             1e-6)]
hearth_problem_mechanical.set_mu_range(mu_range)
```

Next, reduction with POD-Galerkin method is initialised and offline phase is performed.

```
# 4B. Prepare reduction with a POD-Galerkin method
#NOTE : truth_problem attribute is FEM problem and reduced_problem is RB problem
pod_galerkin_method_mechanical = PODGalerkin(hearth_problem_mechanical)
pod_galerkin_method_mechanical.set_Nmax(100) #Maximum size of reduced basis space
pod_galerkin_method_mechanical.set_tolerance(1e-4) #Maximum eigenvalue tolerance

# 5B. Perform the offline phase
pod_galerkin_method_mechanical.initialize_training_set(1000) #Initialize training set
                                             with specified number of training parameters
reduced_hearth_problem_mechanical = pod_galerkin_method_mechanical.offline() #Perform
                                             offline phase
```

Error analysis is performed and time taken for the computation of truth solution and reduced basis solution are measured.

```
# 7B. Perform an error analysis
pod_galerkin_method_mechanical.initialize_testing_set(50) #Initialize error analysis set
                                             with specified number of parameters
pod_galerkin_method_mechanical.error_analysis() #Perform error analysis

# 8B1. Perform a speedup analysis - Compute time for truth solutions
pod_galerkin_method_mechanical.initialize_testing_set(50) #Initialize speedup analysis
                                             set with specified number of parameters
testing_set_speedup_analysis = pod_galerkin_method_mechanical.testing_set

pod_galerkin_method_mechanical._patch_truth_solve(True) # To disable cache reading

truth_timer = Timer("parallel") #Timer for computation of FEM solution
time_mechanical_truth = np.empty(len(testing_set_speedup_analysis)) #Storage of time
                                             taken for solving FEM equation. It is a
                                             vector of size of number of speedup analysis
                                              parameters

# Iteration over speedup analysis parameters for measuring time taken for FEM solution
```

```
for (mu_index, mu_test) in enumerate(testing_set_speedup_analysis):
  print(TextLine(str(mu_index), fill="#"))
  pod_galerkin_method_mechanical.truth_problem.set_mu(mu_test) #Set the parameter
  truth_timer.start()
  pod_galerkin_method_mechanical.truth_problem.solve() #Solve the FEM problem
  truth_time_mechanical = truth_timer.stop()
  print("Truth time mechanical : ",truth_time_mechanical)
  time_mechanical_truth[mu_index] = truth_time_mechanical #Save time taken for truth solve

pod_galerkin_method_mechanical._undo_patch_truth_solve(True) #To enable cache reading

np.save("time_mechanical_truth",time_mechanical_truth) #Save numpy array of time taken
                                                       for FEM solution

# 8B2. Perform a speedup analysis - Compute time for reduced solutions
pod_galerkin_method_mechanical._patch_truth_solve(True) #To disable cache reading

reduced_timer = Timer("serial") #Timer for computation of reduced solution
max_basis_function = reduced_hearth_problem_mechanical.N # Size of reduced basis space
time_mechanical_reduced = np.empty([max_basis_function,len(testing_set_speedup_analysis)]
                                                       ) #Storage of time taken for solving RB
                                                       equation. It is a matrix of size size of
                                                       reduced basis space \times number of speedup
                                                        analysis parameters

# Iteration over speedup analysis parameters for measuring time taken for RB solution
for basis_size in range(1,max_basis_function+1):
  for (mu_index, mu_test) in enumerate(testing_set_speedup_analysis):
    print(TextLine(str(mu_index), fill="#"))
    pod_galerkin_method_mechanical.reduced_problem.set_mu(mu_test) #Set the parameter
    reduced_timer.start()
    pod_galerkin_method_mechanical.reduced_problem.solve(basis_size) #Solve the RB problem
    rb_time_mechanical = reduced_timer.stop()
    print("Reduced time mechanical : ",rb_time_mechanical)
    time_mechanical_reduced[basis_size-1,mu_index] = rb_time_mechanical #Save time taken
                                                       for reduced basis solution

pod_galerkin_method_mechanical._undo_patch_truth_solve(True) #To enable cache reading

np.save("time_mechanical_reduced",time_mechanical_reduced) #Save numpy array of time
                                                       taken for RB solution
```

For any new parameter *online_mu*, the truth solution and the reduced basis solution are computed as:

```
# 6B. Perform an online solve
pod_galerkin_method_mechanical.reduced_problem.set_mu(online_mu) #Set parameter
u_rb = pod_galerkin_method_mechanical.reduced_problem.solve() #Reduced problem solve
pod_galerkin_method_mechanical.reduced_problem.export_solution(filename="
                                                       reference_domain_mechanical_rb") #Save
                                                       solution for visualization with paraview
u_rb = pod_galerkin_method_mechanical.reduced_problem.basis_functions * u_rb #RB solution
                                                        projected back to FEM space
pod_galerkin_method_mechanical.truth_problem.set_mu(online_mu) #Set parameter
u = pod_galerkin_method_mechanical.truth_problem.solve() #FEM problem solve
pod_galerkin_method_mechanical.truth_problem.export_solution(filename="
                                                       reference_domain_fem") #Save solution for
                                                       visualization with paraview
pod_galerkin_method_mechanical.truth_problem.mesh_motion.move_mesh() #Deform mesh as per
                                                       geometric parameters
File("HearthMechanical/reference_domain_mechanical_spatial_error.pvd") << project(u-u_rb,
                                                       VM) #Spatial error
```

```
pod_galerkin_method_mechanical.truth_problem.mesh_motion.reset_reference() #Restore mesh
                                                to reference configuration
```

9.5.5.3. Coupling system

For the coupling system, we define the class *HearthThermoMechanical*.

```
class HearthThermoMechanical(EllipticCoerciveProblem):
```

The default initialization is specified,

```
  # Default initialization of members
  def __init__(self, V, **kwargs):
    # Call the standard initialization
    EllipticCoerciveProblem.__init__(self, V, **kwargs)
    # ... and also store FEniCS data structures for assembly
    assert "subdomains" in kwargs
    assert "boundaries" in kwargs
    assert "mesh" in kwargs
    assert "hearth_problem_thermal" in kwargs
    assert "ref_temperature" in kwargs
    self.subdomains, self.boundaries = kwargs["subdomains"], kwargs["boundaries"]
    self.u = TrialFunction(V)
    self.v = TestFunction(V)
    self.dx = Measure("dx")(subdomain_data=subdomains)
    self.ds = Measure("ds")(subdomain_data=boundaries)
    self.subdomains = subdomains
    self.boundaries = boundaries
    self.hearth_problem_thermal = kwargs["hearth_problem_thermal"]
    self.T_0 = kwargs["ref_temperature"]
    self.x0 = Expression("x[0]", element=V.sub(0).ufl_element())
```

Similar to mechanical system, we define the affine mutiplicative terms, weak formulation and strain tensor.

```
  # Return theta multiplicative terms of the affine expansion of the problem.
  def compute_theta(self, term):
    mu = self.mu
    if term == "a":
      theta_a0 = mu[11]
      theta_a1 = 2*mu[12]
      return (theta_a0, theta_a1,)
    elif term == "f":
      theta_f0 = (2 * mu[11] + 3 * mu[12]) * mu[13]
      return (theta_f0,)
    else:
      raise ValueError("Invalid term for compute_theta().")

  # Return strain tensor
  def strain(self,u):
    r = self.x0
    return sym(as_tensor([[u[0].dx(0), u[0].dx(1), 0. ], [u[1].dx(0), u[1].dx(1), 0.], [0
                                                ., 0., u[0]/r]]))

  # Return forms resulting from the discretization of the affine expansion of the problem
                                                operators.
  def assemble_operator(self, term):
    u = self.u
    v = self.v
    dx = self.dx
    ds = self.ds
```

```
    T_0 = self.T_0
    T = self.hearth_problem_thermal._solution
    r = self.x0
    d_bottom = ds(2) + ds(3) + ds(4) + ds(5) + ds(6)
    d_out = ds(7) + ds(8) + ds(9) + ds(10) + ds(11)
    d_sf = ds(13) + ds(14) + ds(15) + ds(16) + ds(17) + ds(18) + ds(19) + ds(20)
    if term == "a":
      a0 = (u[0].dx(0)+u[1].dx(1)+u[0]/r)*(v[0].dx(0)+v[1].dx(1)+v[0]/r)*r*dx
      a1 = (u[0].dx(0)*v[0].dx(0) + u[1].dx(1)*v[1].dx(1) + (u[0]*v[0])/(r)**2 + 0.5*(u[0].
                                                dx(1)+u[1].dx(0))*(v[0].dx(1)+v[1].dx(0))) *
                                                 r * dx
      return (a0, a1,)
    elif term == "f":
      f0 = (T-T_0) * (v[0].dx(0) + v[1].dx(1) + v[0]/r) * r * dx
      return (f0,)
    elif term == "inner_product":
      x0 = inner(u,v) * r * dx + inner(self.strain(u),self.strain(v)) * r * dx
      return (x0,)
    elif term == "dirichlet_bc":
      bc0 = [DirichletBC(self.V.sub(0), Constant(0.), self.boundaries, 1),
        DirichletBC(self.V.sub(1), Constant(0.), self.boundaries, 2),
        DirichletBC(self.V.sub(1), Constant(0.), self.boundaries, 3),
        DirichletBC(self.V.sub(1), Constant(0.), self.boundaries, 4),
        DirichletBC(self.V.sub(1), Constant(0.), self.boundaries, 5),
        DirichletBC(self.V.sub(1), Constant(0.), self.boundaries, 6),]
      return (bc0,)
    else:
      raise ValueError("Invalid term for assemble_operator().")
```

We define an instance of *HearthThermoMechanical* class and initialise reduction with POD-Galerkin method.

```
# 3C. Allocate an object of the HearthThermoMechanical class
hearth_problem_thermo_mechanical = HearthThermoMechanical(VM, subdomains=subdomains,
                                                boundaries=boundaries, mesh=mesh,
                                                hearth_problem_thermal=
                                                hearth_problem_thermal, ref_temperature=T_0)
#specify and set range of each parameter
mu_range = [(2.3,2.4), (0.5,0.7), (0.5,0.7), (0.4,0.6), (3.05,3.35), (13.5,14.5), (8.3,8.
                                                7), (8.8,9.2), (9.8,10.2), (10.4,10.8), (9.8
                                                ,10.2), (1.9e9,2.5e9), (1.2e9,1.8e9), (0.8e-
                                                6,1.2e-6)]
hearth_problem_thermo_mechanical.set_mu_range(mu_range)

# 4C. Prepare reduction with a POD-Galerkin method
#NOTE : truth_problem attribute is FEM problem and reduced_problem is RB problem
pod_galerkin_method_thermo_mechanical = PODGalerkin(hearth_problem_thermo_mechanical)
pod_galerkin_method_thermo_mechanical.set_Nmax(100) #Maximum size of reduced basis space
pod_galerkin_method_thermo_mechanical.set_tolerance(1e-4) #Maximum eigenvalue tolerance
```

Next, we perform the offline phase.

```
# 5C. Perform the offline phase
pod_galerkin_method_thermo_mechanical.initialize_training_set(1000) #Initialize training
                                                set with specified number of training
                                                parameters
reduced_hearth_problem_thermo_mechanical = pod_galerkin_method_thermo_mechanical.offline
                                                () #Perform offline phase
```

Before performing the reduction of thermal problem, we perform the truth solution computation related operations. This is due to the reason that, we want to use temperature field computed by finite element method for

the truth solution of displacement. We compute the truth solution at few parameters to perform error analysis and to measure time taken for computing the truth solution.

```
# 6C. Perform a truth solve : Reference domain
online_mu = ( 2.365, 0.6, 0.6, 0.5, 3.2, 14.10, 8.50, 9.2, 9.9, 10.6, 10., lame1, lame2,
                                                    1e-6)
pod_galerkin_method_thermo_mechanical.truth_problem.set_mu(online_mu)
u_ref = pod_galerkin_method_thermo_mechanical.truth_problem.solve()
pod_galerkin_method_thermo_mechanical.truth_problem.export_solution(filename="
                                                    reference_domain_fem")

# 6C. Perform a truth solve : Parametrized domain
online_mu = ( 2.365, 0.6, 0.6, 0.45, 3.2, 14.10, 8.30, 9.2, 9.9, 10.6, 10., lame1, lame2,
                                                     1e-6)
pod_galerkin_method_thermo_mechanical.truth_problem.set_mu(online_mu)
u_par = pod_galerkin_method_thermo_mechanical.truth_problem.solve()
pod_galerkin_method_thermo_mechanical.truth_problem.export_solution(filename="
                                                    parametric_domain_fem")

# 7C1. Perform an error analysis - Compute truth solutions
pod_galerkin_method_thermo_mechanical.initialize_testing_set(50) #Initialize error
                                                    analysis set with specified number of
                                                    parameters
testing_set_error_analysis = pod_galerkin_method_thermo_mechanical.testing_set

truth_solution_thermo_mechanical = list()

# Iteration over error analysis parameters for measuring time taken for FEM solution
for (mu_index, mu_test) in enumerate(testing_set_error_analysis):
  print(TextLine(str(mu_index), fill="#"))
  pod_galerkin_method_thermo_mechanical.truth_problem.set_mu(mu_test) #Set parameter
  truth_solution_thermo_mechanical.append(pod_galerkin_method_thermo_mechanical.
                                                    truth_problem.solve()) #Solve and store FEM
                                                    solution

# 8C1. Perform a speedup analysis - Compute time for truth solutions
pod_galerkin_method_thermo_mechanical.initialize_testing_set(50) #Initialize truth
                                                    solution with specified number of parameters
testing_set_speedup_analysis = pod_galerkin_method_thermo_mechanical.testing_set

pod_galerkin_method_thermo_mechanical._patch_truth_solve(True) #To disable cache reading

truth_timer = Timer("parallel") #Timer for computation of FEM solution
time_thermo_mechanical_truth = np.empty(len(testing_set_speedup_analysis)) #Storage of
                                                    time taken for solving FEM equation. It is a
                                                     vector of size of number of speedup
                                                    analysis parameters

# Iteration over speedup analysis parameters for measuring time taken for RB solution
for (mu_index, mu_test) in enumerate(testing_set_speedup_analysis):
  print(TextLine(str(mu_index), fill="#"))
  pod_galerkin_method_thermo_mechanical.truth_problem.set_mu(mu_test) #Set the parameter
  truth_timer.start()
  pod_galerkin_method_thermo_mechanical.truth_problem.solve() #Solve the RB problem
  truth_time_thermo_mechanical = truth_timer.stop()
  print("Truth time thermomechanical : ",truth_time_thermo_mechanical)
  time_thermo_mechanical_truth[mu_index] = truth_time_thermo_mechanical #Save time taken
                                                    for reduced basis solution

pod_galerkin_method_thermo_mechanical._undo_patch_truth_solve(True) #To disable cache
                                                    reading
```

```
np.save("time_thermo_mechanical_truth",time_thermo_mechanical_truth) #Save numpy array of
                                                    time taken for RB solution
```

Now, since the operations related to the truth solution are performed, we can reduce the thermal system. The reduced basis solution of temperature field is used for computing reduced solution at few parameters. Also we compute the reduced basis solution for error analysis and time taken for computing the reduced basis solution.

```
#6C. Perform an online solve : Reference domain
online_mu_reference = ( 2.365, 0.6, 0.6, 0.5, 3.2, 14.10, 8.50, 9.2, 9.9, 10.6, 10.,
                                                    lame1, lame2, 1e-6)
online_mu = online_mu_reference
pod_galerkin_method_thermo_mechanical.reduced_problem.set_mu(online_mu)
u_rb = pod_galerkin_method_thermo_mechanical.reduced_problem.solve()
pod_galerkin_method_thermo_mechanical.reduced_problem.export_solution(filename="
                                                    reference_domain_thermomechanical_rb")
u_rb = pod_galerkin_method_thermo_mechanical.reduced_problem.basis_functions * u_rb
pod_galerkin_method_thermo_mechanical.truth_problem.mesh_motion.move_mesh()
File("HearthThermoMechanical/reference_domain_thermomechanical_spatial_error.pvd") <<
                                                    project(u_ref-u_rb,VM)
pod_galerkin_method_thermo_mechanical.truth_problem.mesh_motion.reset_reference()

# 6C. Perform an online solve : Parametrized domain
online_mu_parametrized = ( 2.365, 0.6, 0.6, 0.45, 3.2, 14.10, 8.30, 9.2, 9.9, 10.6, 10.,
                                                    lame1, lame2, 1e-6)
online_mu = online_mu_parametrized
pod_galerkin_method_thermo_mechanical.reduced_problem.set_mu(online_mu)
u_rb = pod_galerkin_method_thermo_mechanical.reduced_problem.solve()
pod_galerkin_method_thermo_mechanical.reduced_problem.export_solution(filename="
                                                    parametric_domain_thermomechanical_rb")
u_rb = pod_galerkin_method_thermo_mechanical.reduced_problem.basis_functions * u_rb
pod_galerkin_method_thermo_mechanical.truth_problem.mesh_motion.move_mesh()
File("HearthThermoMechanical/parametric_domain_thermomechanical_spatial_error.pvd") <<
                                                    project(u_par-u_rb,VM)
pod_galerkin_method_thermo_mechanical.truth_problem.mesh_motion.reset_reference()

# 7C2. Perform an error analysis - Compute reduced basis solution
dx = Measure("dx")(subdomain_data=subdomains) #Volume measure
r = Expression("x[0]", element=VM.sub(0).ufl_element()) #

max_basis_function = reduced_hearth_problem_thermo_mechanical.N # Size of reduced basis
                                                    space
error_thermo_mechanical = np.empty([max_basis_function,len(testing_set_error_analysis)])
                                                    # Numpy array of size of reduced basis space
                                                     \times number of error analysis parameters
                                                    for storing error
# Iteration over error analysis parameters for measuring time taken for RB solution
for basis_size in range(1,max_basis_function+1):
  for (mu_index, mu_test) in enumerate(testing_set_error_analysis):
    print(TextLine(str(mu_index), fill="#"))
    pod_galerkin_method_thermo_mechanical.reduced_problem.set_mu(mu_test) #Set parameter
    rb_dofs = pod_galerkin_method_thermo_mechanical.reduced_problem.solve(basis_size) #
                                                    Compute reduced basis degrees of freddom
    rb_solution = reduced_hearth_problem_thermo_mechanical.basis_functions[:basis_size] *
                                                    rb_dofs #RB solution projected back to FEM
                                                    space
    # Absolute and relative error measurement
    absolute_error = assemble(inner(truth_solution_thermo_mechanical[mu_index] -
                                                    rb_solution,truth_solution_thermo_mechanical
                                                    [mu_index] - rb_solution) * r * dx + inner(
                                                    hearth_problem_thermo_mechanical.strain(
                                                    truth_solution_thermo_mechanical[mu_index] -
```

```
                                                rb_solution),
                                               hearth_problem_thermo_mechanical.strain(
                                               truth_solution_thermo_mechanical[mu_index] -
                                                rb_solution)) * r * dx)
    error_thermo_mechanical[basis_size-1,mu_index] = np.sqrt(absolute_error / assemble(
                                               inner(truth_solution_thermo_mechanical[
                                               mu_index],truth_solution_thermo_mechanical[
                                               mu_index]) * r * dx + inner(
                                               hearth_problem_thermo_mechanical.strain(
                                               truth_solution_thermo_mechanical[mu_index]),
                                                hearth_problem_thermo_mechanical.strain(
                                               truth_solution_thermo_mechanical[mu_index]))
                                                * r * dx))

np.save("HearthThermoMechanical/error_analysis/error_thermo_mechanical",
                                               error_thermo_mechanical)

# 8C2. Perform a speedup analysis - Compute time for reduced solutions
pod_galerkin_method_thermo_mechanical._patch_truth_solve(True) #To disable cache reading

reduced_timer = Timer("serial") #Timer for computation of RB solution
time_thermo_mechanical_reduced = np.empty([max_basis_function,len(
                                               testing_set_speedup_analysis)]) #Storage of
                                               time taken for solving RB equation. It is a
                                               matrix of size size of reduced basis space \
                                               times number of speedup analysis parameters

# Iteration over speedup analysis parameters for measuring time taken for RB solution
for basis_size in range(1,max_basis_function+1):
  for (mu_index, mu_test) in enumerate(testing_set_speedup_analysis):
    print(TextLine(str(mu_index), fill="#"))
    pod_galerkin_method_thermo_mechanical.reduced_problem.set_mu(mu_test) #Set parameter
    reduced_timer.start()
    pod_galerkin_method_thermo_mechanical.reduced_problem.solve(basis_size) #Solve the RB
                                               problem
    rb_time_thermo_mechanical = reduced_timer.stop()
    print("Reduced time thermomechanical : ",rb_time_thermo_mechanical)
    time_thermo_mechanical_reduced[basis_size-1,mu_index] = rb_time_thermo_mechanical #
                                               Save time taken for reduced basis solution

pod_galerkin_method_thermo_mechanical._undo_patch_truth_solve(True) # To disable cache
                                               reading

np.save("time_thermo_mechanical_reduced",time_thermo_mechanical_reduced) #Save numpy
                                               array of time taken for RB solution
```

9.6. License

- FEniCS and RBniCS are freely available under the GNU LGPL, version 3.
- Matplotlib only uses BSD compatible code, and its license is based on the PSF license. Non-BSD compatible licenses (e.g., LGPL) are acceptable in matplotlib toolkits.

Accordingly, this code is freely available under the GNU LGPL, version 3.

9.7. Disclaimer

In downloading this SOFTWARE you are deemed to have read and agreed to the following terms: This SOFTWARE has been designed with an exclusive focus on civil applications. It is not to be used for any illegal, deceptive, misleading or unethical purpose or in any military applications. This includes ANY APPLICA-

TION WHERE THE USE OF THE SOFTWARE MAY RESULT IN DEATH, PERSONAL INJURY OR SEVERE PHYSICAL OR ENVIRONMENTAL DAMAGE. Any redistribution of the software must retain this disclaimer. BY INSTALLING, COPYING, OR OTHERWISE USING THE SOFTWARE, YOU AGREE TO THE TERMS ABOVE. IF YOU DO NOT AGREE TO THESE TERMS, DO NOT INSTALL OR USE THE SOFTWARE.

References

[1] M. Bannenberg, P. Barral, A. Bärmann, J.-D. Benamou, F. Bianco, A. Binder, G. Chazareix, S. Dittmer, D. F. Comesaña, M. Girfoglio, M. Günther, J. C. Gutiérrez Pérez, L. Hauberg-Lotte, W. Ijzerman, O. Jadhav, T. Kluth, A. Lengomin, P. Maass, G. Marconi, A. Martin, M. Martinolli, V. Mehrmann, P. P. Monticone, U. Morelli, A. Nayak, A. Obereder, D. Otero Baguer, L. Polverelli, A. Prieto, P. Quintela, R. Ramlau, C. Riccardo, G. Rozza, G. Rukhaia, N. Shah, G. Stabile, B. Stadler, J. Staszek, and C. Vergara, "Software-based representation of selected benchmark hierarchies equipped with publically available data," Jun. 2020, project Deliverable (D5.2), Version 2.0. [Online]. Available: https://doi.org/10.5281/zenodo.3888145

[2] M. Yudytskiy, "Wavelet methods in adaptive optics," Ph.D. dissertation, Johannes Kepler University Linz, 2014.

[3] B. Stadler, R. Biasi, M. Manetti, and R. Ramlau, "Real-time implementation of an iterative solver for atmospheric tomography," 2020.

[4] B. Stadler, R. Biasi, and R. Ramlau, "Feasibility of standard and novel solvers in atmospheric tomography for the ELT," in *Proc. AO4ELT6*, 2019.

[5] R. Ramlau and B. Stadler, "An augmented wavelet reconstructor for atmospheric tomography," *Electron. Trans. Numer. Anal.*, vol. 54, pp. 256–275, 2021.

[6] S. Marburg and B. Nolte, *Computational Acoustics of Noise Propagation in Fluids - Finite and Boundary Element Methods*. Springer-Verlag Berlin Heidelberg, 2008.

[7] J.-P. Berenger, "A perfectly matched layer for the absorption of electromagnetic waves," *Journal of Computational Physics*, vol. 114, no. 2, pp. 185 – 200, 1994.

[8] X. Sagartzazu, L. Hervella-Nieto, and J. M. Pagalday, "Review in sound absorbing materials," *Archives of Computational Methods in Engineering*, vol. 15, no. 3, pp. 311–342, Sep 2008.

[9] J. Allard and N. Atalla, *Propagation of Sound in Porous Media: Modelling Sound Absorbing Materials 2e*. Wiley, 2009.

[10] A. Bermúdez, L. Hervella-Nieto, A. Prieto, and R. Rodríguez, "Perfectly matched layers for time-harmonic second order elliptic problems," *Archives of Computational Methods in Engineering*, vol. 17, no. 1, pp. 77–107, Mar 2010.

[11] A. Bermúdez, L. Hervella-Nieto, A. Prieto, and R. Rodríguez, "An optimal perfectly matched layer with unbounded absorbing function for time-harmonic acoustic scattering problems," *Journal of Computational Physics*, vol. 223, no. 2, pp. 469 – 488, 2007. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0021999106004487

[12] "Salome documentation." [Online]. Available: https://www.salome-platform.org/user-section/documentation/current-release

[13] H. P. Langtangen and A. Logg, *Solving PDEs in Python*. Springer, 2017.

[14] "Fenics documentation." [Online]. Available: https://fenicsproject.org/documentation/

[15] A. S. Nayak, A. Prieto, and D. Fernández-Comeseña, "Implementing acoustic scattering simulations for external geometries within a porous enclosure," in *Software-based representation of selected benchmark hierarchies equipped with publically available data*. Zenodo, Jun. 2020, pp. 2–18, rOMSOC Deliverable D5.2, Version 2.0. [Online]. Available: https://doi.org/10.5281/zenodo.3888145

[16] J. D. Hunter, "Matplotlib: A 2d graphics environment," *Computing in Science & Engineering*, vol. 9, no. 3, pp. 90–95, 2007.

[17] U. Ayachit, *The ParaView Guide: A Parallel Visualization Application*. USA: Kitware, Inc., 2015.

[18] "Paraview tutorials." [Online]. Available: https://www.paraview.org/Wiki/The_ParaView_Tutorial

[19] T. Glimm and V. Oliker, "Optical design of single reflector systems and the monge–kantorovich mass transfer problem," *Journal of Mathematical Sciences*, vol. 117, pp. 4096–4108, 09 2003.

[20] X.-J. Wang, "On the design of a reflector antenna ii," *Calculus of Variations and Partial Differential Equations*, vol. 20, no. 3, pp. 329–341, Jul 2004. [Online]. Available: https://doi.org/10.1007/s00526-003-0239-4

[21] C. Villani, *Optimal Transport: Old and New*, ser. Grundlehren der mathematischen Wissenschaften. Springer Berlin Heidelberg, 2008. [Online]. Available: https://books.google.fr/books?id=NZXiNAEACAAJ

[22] G. Peyré and M. Cuturi, "Computational Optimal Transport," *ArXiv e-prints*, Mar. 2018.

[23] M. Cuturi, "Sinkhorn distances: Lightspeed computation of optimal transport," in *Advances in Neural Information Processing Systems 26*, C. J. C. Burges, L. Bottou, M. Welling, Z. Ghahramani, and K. Q. Weinberger, Eds. Curran Associates, Inc., 2013, pp. 2292–2300. [Online]. Available: http://papers.nips.cc/paper/4927-sinkhorn-distances-lightspeed-computation-of-optimal-transport.pdf

[24] C. Léonard, "A survey of the schrödinger problem and some of its connections with optimal transport," 2013.

[25] B. Schmitzer, "Stabilized Sparse Scaling Algorithms for Entropy Regularized Transport Problems," *arXiv e-prints*, p. arXiv:1610.06519, Oct 2016.

[26] A. M. Oberman and Y. Ruan, "An efficient linear programming method for optimal transportation," 2015.

[27] R. J. Berman, "The Sinkhorn algorithm, parabolic optimal transport and geometric Monge-Ampere equations," *ArXiv e-prints*, Dec. 2017.

[28] R. S. Womersley, "Efficient Spherical Designs with Good Geometric Properties," *ArXiv e-prints*, Sep. 2017.

[29] J. Feydy, T. Séjourné, F.-X. Vialard, S.-i. Amari, A. Trouvé, and G. Peyré, "Interpolating between Optimal Transport and MMD using Sinkhorn Divergences," Oct. 2018, working paper or preprint. [Online]. Available: https://hal.archives-ouvertes.fr/hal-01898858

[30] J.-D. Benamou, W. L. Ijzerman, and G. Rukhaia, "An Entropic Optimal Transport Numerical Approach to the Reflector Problem," Apr. 2020, working paper or preprint. [Online]. Available: https://hal.archives-ouvertes.fr/hal-02539799

[31] M. Bannenberg, A. Ciccazzo, and M. Günther, "Reduced order multirate schemes for coupled differential-algebraic systems," *Applied Numerical Mathematics*, 2021.

[32] ——, "Coupling of model order reduction and multirate techniques for coupled dynamical systems," *Applied Mathematics Letters*, vol. 112, p. 106780, 2021.

[33] M. Bannenberg, F. Kasolis, M. Günther, and M. Clemens, "Maximum entropy snapshot sampling for reduced basis modelling," —, 2020.

[34] H. Broer and F. Takens, *Dynamical systems and chaos*. Springer-Verlag New York, 2011.

[35] F. Kasolis, D. Zhang, and M. Clemens, "Recurrent quantification analysis for model reduction of nonlinear transient electro-quasistatic field problems," in *International Conference on Electromagnetics in Advanced Applications (ICEAA 2019)*, 2019, pp. 14–17.

[36] H. A. Tong, *Dimension estimation and models*. World Scientific, 1993, vol. 1.

[37] B. van der Waerden, *Mathematical Statistics*. Springer, 1969.

[38] A. Verhoeven, "Redundancy reduction of IC models : by multirate time-integration and model order reduction," Ph.D. dissertation, Department of Mathematics and Computer Science, 2008.

[39] G. Wanner and E. Hairer, *Solving ordinary differential equations II*. Springer Berlin Heidelberg, 1996, vol. 375.

[40] C. Hachtel, A. Bartel, M. Günther, and A. Sandu, "Multirate implicit Euler schemes for a class of differential–algebraic equations of index-1," *J. Comput. Appl. Math.*, p. 112499, 2019.

[41] A. Verhoeven, J. Ter Maten, M. Striebel, and R. Mattheij, "Model order reduction for nonlinear ic models," in *IFIP Conference on System Modeling and Optimization*. Springer, 2007, pp. 476–491.

[42] A. Chatterjee, "An introduction to the proper orthogonal decomposition," *Curr. Sci.*, vol. 78, pp. 808–817, 2000.

[43] G. Berkooz, P. Holmes, and J. Lumley, "The proper orthogonal decomposition in the analysis of turbulent flows," *Annu. Rev. Fluid Mech.*, vol. 25, no. 1, pp. 539–575, 1993.

[44] L. Sirovich, "Turbulence and the dynamics of coherent structures. Part I: coherent structures," *Quart. Appl. Math.*, vol. 45, no. 3, pp. 561–571, 1987.

[45] A. Binder, O. Jadhav, and V. Mehrmann, "Model order reduction for the simulation of parametric interest rate models in financial risk analysis," *J. Math. Industry*, vol. 11, pp. 1–34, 2021.

[46] J. Edmonds, "Matroids and the greedy algorithm," *Math. Program.*, vol. 1, pp. 127–136, 1971.

[47] European Commission, "Commission delegated regulation (EU) 2017/653," *Off. J. EU*, vol. 1, pp. 1–52, 2017.

[48] ——, "Commission delegated regulation (EU) 1286/2014," *Off. J. EU*, vol. 1, pp. 1–23, 2014.

[49] S. Shreve, *Stochastic Calculus and Finance*, 1st ed. New York, US: Springer-Verlag, 2004.

[50] M. Aichinger and A. Binder, *A Workout in Computational Finance*, 1st ed. West Sussex, UK: John Wiley and Sons Inc., 2013.

[51] H. Engl, "Calibration problems–an inverse problems view," *Wilmott*, pp. 16–20, 2007.

[52] MathConsult, "Calibration of interest rate models," MathConsult GmbH, Linz, Austria, Report, 2009.

[53] M. Rathinam and L. Petzold, "A new look at proper orthogonal decomposition," *SIAM J. Numer. Anal.*, vol. 41, no. 5, pp. 1893–1925, 2003.

[54] M. Williams, P. Schmid, and J. Kutz, "Hybrid reduced-order integration with proper orthogonal decomposition and dynamic mode decomposition," *SIAM J. Multiscale Model. Simul.*, vol. 11, no. 2, pp. 522–544, 2013.

[55] R. Pinnau, "Model reduction via proper orthogonal decomposition," in *Model Order Reduction: Theory, Research Aspects and Applications*, W. Schilders, H. van der Vorst, and J. Rommes, Eds. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 95–109.

[56] G. James, D. Witten, T. Hastie, and R. Tibshirani, *An Introduction to Statistical Learning*, 1st ed. New York, NY: Springer-Verlag, 2013.

[57] J. R. Quinlan, "Induction of decision trees," *Machine Learning*, vol. 1, pp. 81–106, 1986.

[58] A. Forrester, A. Sóbester, and A. Keane, *Engineering design via surrogate modelling: a practical guide*, 1st ed. West Sussex, UK: John Wiley and Sons Inc., 2008.

[59] D. Jones, "A taxonomy of global optimization methods based on response surfaces," *J. Global Optim.*, vol. 21, no. 4, pp. 345–383, 2001.

[60] S. Raff, "Routing and scheduling of vehicles and crews. The state of the art." *Computers & Operations Research*, vol. 10, no. 2, pp. 63–211, Jan. 1983. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/0305054883900308

[61] L. Klimeš and J. Štětina, "A rapid gpu-based heat transfer and solidification model for dynamic computer simulations of continuous steel casting," *Journal of Materials Processing Technology*, vol. 226, pp. 1–14, 2015.

[62] I. Samarasekera and J. Brimacombe, "The influence of mold behavior on the production of continuously cast steel billets," *Metallurgical Transactions B*, vol. 13, no. 1, pp. 105–116, 1982.

[63] U. E. Morelli, P. Barral, P. Quintela, G. Rozza, and G. Stabile, "A numerical approach for heat flux estimation in thin slabs continuous casting molds using data assimilation," *International Journal for Numerical Methods in Engineering*, vol. n/a, no. n/a, 2021.

[64] F. D. Moura Neto and A. J. da Silva Neto, *An Introduction to Inverse Problems with Applications*. Springer Publishing Company, Incorporated, 2012.

[65] R. Fletcher and C. M. Reeves, "Function minimization by conjugate gradients," *The Computer Journal*, vol. 7, no. 2, pp. 149–154, 01 1964.

[66] P. Ranut, "Optimization and inverse problems in heat transfer," Ph.D. dissertation, Universitá degli Studi di Udine, Via delle Scienze, 206, 33100 Udine UD, Italy, 2012.

[67] M. D. Buhmann, *Radial basis functions: theory and implementations*. Cambridge university press, 2003, vol. 12.

[68] G. Prando, "Non-parametric bayesian methods for linear system identification," Ph.D. dissertation, Universitá di Padova, Via 8 Febbraio 1848, 2, 35122 Padova PD, Italy, 2016.

[69] G. Stabile, S. Hijazi, A. Mola, S. Lorenzi, and G. Rozza, "POD-Galerkin reduced order methods for CFD using Finite Volume Discretisation: vortex shedding around a circular cylinder," *Communications in Applied and Industrial Mathematics*, vol. 8, no. 1, pp. 210–236, (2017).

[70] ITHACA-FV, https://mathlab.sissa.it/ithaca-fv, accessed: 2020-10-26.

[71] F. Moukalled, L. Mangani, and M. Darwish, *The Finite Volume Method in Computational Fluid Dynamics: An Advanced Introduction with OpenFOAM and Matlab*, 1st ed. Springer Publishing Company, Incorporated, 2015.

[72] OpenCFD, *OpenFOAM - The Open Source CFD Toolbox - User's Guide*, 1st ed., OpenCFD Ltd., United Kingdom, 11 2007.

[73] C. N. Botterbusch, S. Lucquin, P. Monticone, J. B. Drevet, A. Guignabert, and P. Meneroud, "Implantable pump system having an undulating membrane," May 15 2018, uS Patent 9,968,720.

[74] M. Perschall, J. B. Drevet, T. Schenkel, and H. Oertel, "The progressive wave pump: numerical multiphysics investigation of a novel pump concept with potential to ventricular assist device application," *Artificial organs*, vol. 36, no. 9, 2012.

[75] E. Burman and M. A. Fernández, "An unfitted Nitsche method for incompressible fluid–structure interaction using overlapping meshes," *Computer Methods in Applied Mechanics and Engineering*, vol. 279, pp. 497–514, 2014.

[76] A. Hansbo and P. Hansbo, "An unfitted finite element method, based on Nitsche's method, for elliptic interface problems," *Computer methods in applied mechanics and engineering*, vol. 191, no. 47-48, pp. 5537–5552, 2002.

[77] M. Martinolli, J. Biasetti, S. Zonca, L. Polverelli, and C. Vergara, "Extended finite element method for fluid-structure interaction in wave membrane blood pump," *International Journal for Numerical Methods in Biomedical Engineering*, vol. 37, no. 7, p. e3467, 2021. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/cnm.3467

[78] S. Zonca, C. Vergara, and L. Formaggia, "An unfitted formulation for the interaction of an incompressible fluid with a thick structure via an XFEM/DG approach," *SIAM Journal on Scientific Computing*, vol. 40, no. 1, pp. B59–B84, 2018.

[79] C. Geuzaine and J.-F. Remacle, "Gmsh: A 3-D finite element mesh generator with built-in pre-and post-processing facilities," *International journal for numerical methods in engineering*, vol. 79, no. 11, pp. 1309–1331, 2009.

[80] P. Barral, M. Girfoglio, A. Lengomin, P. Quintela, and N. Shah, "Coupled parameterized reduced order modelling of thermo-mechanical phenomena arising in blast furnace," Jun. 2020, project Deliverable (D5.2), Version 2.0. [Online]. Available: https://doi.org/10.5281/zenodo.3888145

[81] J. Fehr, C. Himpe, S. Rave, and J. Saak, "Sustainable research software hand-over," *Journal of Open Research Software*, vol. 9, 2021. [Online]. Available: http://dx.doi.org/10.5334/jors.307

[82] M. S. Alnæs, J. Blechta, J. Hake, A. Johansson, B. Kehlet, A. Logg, C. Richardson, J. Ring, M. E. Rognes, and G. N. Wells, "The fenics project version 1.5," *Archive of Numerical Software*, vol. 3, no. 100, 2015.

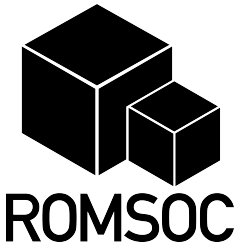
[83] A. Logg, K.-A. Mardal, G. N. Wells *et al.*, *Automated Solution of Differential Equations by the Finite Element Method*. Springer, 2012.

[84] A. Logg, G. N. Wells, and J. Hake, *DOLFIN: a C++/Python Finite Element Library*. Springer, 2012, ch. 10.

[85] A. Logg and G. N. Wells, "Dolfin: Automated finite element computing," *ACM Transactions on Mathematical Software*, vol. 37, no. 2, 2010.

[86] J. S. Hesthaven, G. Rozza, and B. Stamm, *Certified Reduced Basis Methods for Parametrized Partial Differential Equations*, ser. SpringerBriefs in Mathematics. Springer International Publishing, 2015.

[87] C. R. Harris, K. J. Millman, S. J. van der Walt, R. Gommers, P. Virtanen, D. Cournapeau, E. Wieser, J. Taylor, S. Berg, N. J. Smith, R. Kern, M. Picus, S. Hoyer, M. H. van Kerkwijk, M. Brett, A. Haldane, J. F. del Río, M. Wiebe, P. Peterson, P. Gérard-Marchant, K. Sheppard, T. Reddy, W. Weckesser, H. Abbasi, C. Gohlke, and T. E. Oliphant, "Array programming with NumPy," *Nature*, vol. 585, no. 7825, pp. 357–362, Sep. 2020. [Online]. Available: https://doi.org/10.1038/s41586-020-2649-2



The ROMSOC project

September 1, 2021

ROMSOC-D5.3-0.1

Horizon 2020