Escola de Camins
Escola Tècnica Superior d'Enginyeria de Camins, Canals i Ports
UPC BARCELONATECH

Development of Reduced Order Models in application to thermal problems within the Kratos framework

Treball realitzat per:
José Raúl Bravo Martínez

Dirigit per:
Prof. Riccardo Rossi

Codirigit per:
Prof. Joaquín A. Hernandez Ortega

Màster en:
Computational Mechanics

Barcelona, 14-06-2019

Departament d 'Enginyeria Civil i Ambiental

TREBALL FINAL DE MÀSTER

Development of Reduced Order Models in application to thermal problems within the Kratos framework

BSc Jose Raul Bravo Martinez

Master thesis submitted under the supervision of
Prof. Riccardo Rossi

the co-supervision of
Prof. Joaquín A. Hernandez Ortega

in order to be awarded the Degree of
Master of Science in Computational Mechanics

Academic year
2017 – 2019

14/June/2019

Title: Development of Reduced Order Models in application to thermal problems within the Kratos framework

Author: BSc Jose Raul Bravo Martinez
Master of Science in Computational Mechanics
Academic year: 2017 – 2019

Abstract

Problems in science and engineering solved using the Finite Element Method (FEM) can produce large systems of equations that demand considerable computer power and/or a long time to be computed. One of the techniques to overcome this barrier are the so-called Reduced Order Models (ROM), which allow to obtain accurate representations of the solution of a large model, incurring a fraction of the computational burden.

The aim of this work is to implement a ROM application in the FEM program Kratos Multiphysics. More precisely, the Python interface, as well as the C++ scripts required to train and run ROM simulations using the Proper Orthogonal Decomposition (POD) within the Kratos framework, are presented. Validation of the methodology is performed for simple geometries in 2D and 3D for thermal problems. A study on the accuracy and performance of the implementation is also performed.

Keywords: Reduced Order Models, Proper Orthogonal Decomposition, Singular Value Decomposition, Finite Elements, Kratos Multiphysics

Acknowledgements

First off, I would like to acknowledge the **European Union** for its selfless aid to international students in Europe. Thank you for granting me the Erasmus Mundus Scholarship, which allowed me to complete this program, one of the best for Computational Mechanics in the world. This represented one of the most important and positively impacting experiences of my life.

I feel privileged for the quality of the professors I had during these two years of studies, both at Swansea University, Wales, and Universitat Politécnica de Catalunya, Spain. I would especially like to thank **Dr. Rubén Sevilla**, the director of the program in Swansea University, thank you for your guidance and professionalism. **Dr. Antonio Gil**, one of the most passionate professors I have had, thank you for your unconditional support. To my supervisors, **Prof. Riccardo Rossi** and **Prof. Joaquín Hernandez**, thank you for your patience and all the lessons taught during the months I worked in this project. I would also like to thank **Ms. Lelia Zielonka**, secretary of the masters, who was always there to help with anything, even before arriving in Europe.

One of the parts that I value the most from this experience has been the chance to interact with some of the most talented and sincerely good people I have met. To **Sai Praneeth**, **Prashanth Lakshmi Narasimhan**, and **Mohanad Agamy**; thank you for friendship, I learned lots from you. To **Shushu Qin**, the brightest mind I know, but also a very kind human being, and to **Arthur Lustman**, one of my dearest friends; both of whom were with me these two years, thank you for your continuous help, I would not have been able to do it without you.

Finally, to my family, specially to my parents **Rosa Martínez** and **Rubén Bravo**. Thank you for your unconditional support and love. All of my achievements are yours, because you taught me to set myself goals, and to work with humility and honesty to achieve them.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

For many years, experiments had been the tool to understand the mechanics dictating the behavior of phenomena in science. Industry, on the other hand, has used experiments to validate designs under working conditions. However, for many applications, it is either very expensive to carry out an experiment, for example a wind tunnel test; or it is impossible to do it, for example weather forecasting, or testing a new stent on a living person. Numerical simulations are an alternative to experiments based on mathematical modeling of the phenomenon to study, numerical solution of the discretized model, and analysis of the results.

With the increase in computer power, and the development of numerical methods in engineering; numerical simulations have gained presence in all the sectors of industry and in science. They provide a cheaper alternative, and allow to obtain a lot more data, when compared to an experiment[1].

However, the increase in complexity of the problems and in sophistication of the numerical techniques also increases the size of the problems to be solved by the computer. One of the techniques to overcome this barrier is to submit the corresponding data to a process commonly referred to as dimensionality reduction. This process attempts to extract a few dominant structures or modes from a larger data set. In the specific field of computational mechanics, the process of dimensionality reduction is more commonly known as model reduction [2], which is capable of obtaining accurate representations of the solution of a large model, incurring a fraction of the computational burden.

1.1 Objectives

Kratos is a powerful and large multiphysics software. It is open source and it counts with a modular design that has allowed the incorporation of many universities and companies into its further development.

The main objective of this work is the implementation from scratch in the software Kratos Multiphysics of a Reduced Order Model application based on the Proper Orthogonal Decomposition technique. All the required scripts have to be created in the programming languages of Kratos Multiphysics (C++ and Python).

The final ROM application must be able to solve linear and nonlinear problems, for thermal applications. A hyper-reduction method is to be employed to further reduce the size of the system; as well as techniques to perform the singular value decomposition in an efficient manner.

Chapter 2
Theoretical Background

This chapter presents the main theory used throughout this document. The Singular Value Decomposition is presented in an illustrative way. Then, more formalism is introduced, followed by the Proper Orthogonal Decomposition theory.

The final part of this chapter deals with an example of a finite element 1D bar. By contrasting the modal analysis technique with ROM, the bar example is indented to highlight the main features of the ROM applied to a linear and nonlinear problem. The code used for this example can be found in the appendix.

2.1 Singular Value Decomposition

Theorem 1 *Every matrix* $\boldsymbol{A} \in \mathbb{C}^{m\times n}$ *has a singular value decomposition with singular values* σ *and vectors* $\boldsymbol{v}$ *and* $\boldsymbol{u}$*, which are grouped respectively in the matrices* $\boldsymbol{U}$*,* $\boldsymbol{\Sigma}$ *and* $\boldsymbol{V^*}$ *[3].*

$$\boldsymbol{A} = \boldsymbol{U\Sigma V}^*$$

It can be shown that the SVD is related to the eigenvalue decomposition. However, notice the greater applicability of the SVD, since it does not require any characteristic on the matrix $\boldsymbol{A}$.

When dealing with real numbers only, Theorem 1 can be specialized, and the complex conjugate of the matrix $\boldsymbol{V}$ becomes simply its transpose. For the rest of this thesis, the form used for the singular value decomposition of a matrix $\boldsymbol{A}$ is:

$$\boldsymbol{A} = \boldsymbol{U\Sigma V}^T \tag{2.1}$$

where $\boldsymbol{U}$ is the matrix of left singular vectors, $\boldsymbol{\Sigma}$ is the diagonal matrix of singular values, and $\boldsymbol{V}^T$ is the matrix of right singular vectors.

In order to illustrate the fundamental idea of the SVD and demonstrate its wide applicability, the following example on decomposing a picture is presented:

2.1.1 Image Decomposition

An electronic image is nothing but a matrix containing the information of the pixels. In this example, the SVD of a JPEG image shown in figure 2.1 is taken. This picture is in grey scale, but the same procedure can be applied to color pictures, with some extra treatment.

The file was loaded to MATLAB and a SVD was performed. Then, one can select different amounts of "modes" to reconstruct the picture. Modes here refer to the columns of the matrix of left singular vectors $\boldsymbol{U}$.

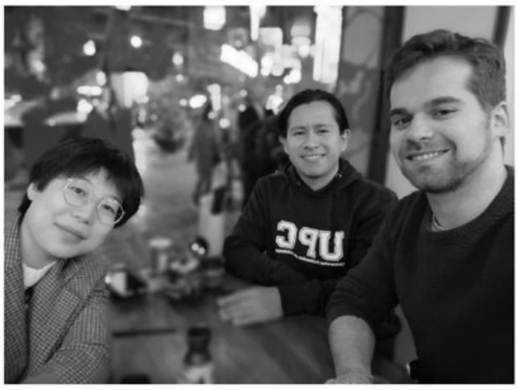
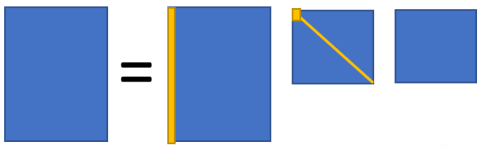
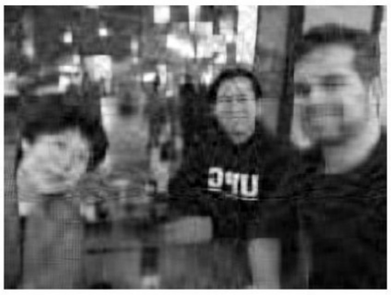


Figure 2.1: Original picture to obtain a decomposition from



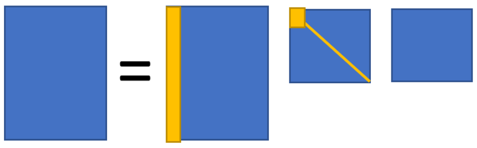
(a)



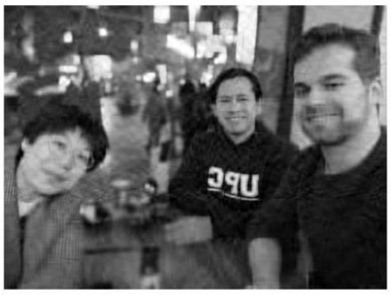
(b)

Figure 2.2: Reconstruction of picture using 5% of modes

Figures 2.3, 2.4, and 2.5 are obtained when reconstructing the original matrix via a truncated SVD. In all figures, a) is a representation of the decomposition and b) is the obtained pictured.

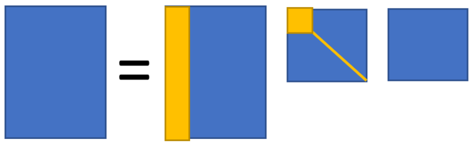


(a)

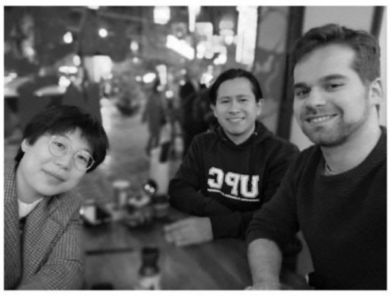


(b)

Figure 2.3: Reconstruction of picture using 10% of modes

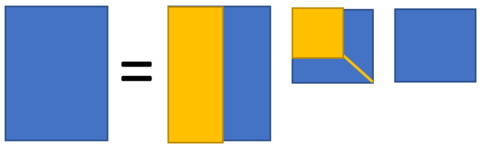


(a)

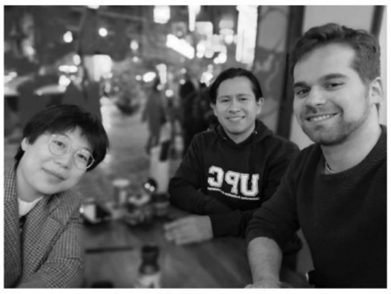


(b)

Figure 2.4: Reconstruction of picture using 25% of modes



(a)



(b)

Figure 2.5: Reconstruction of picture using 50% of modes

2.2 Proper Orthogonal Decomposition

Most concepts mentioned in this section, including the format of the equations are taken from the lectures on the POD and SVD from prof. Kutz, University of Washington [3].

The main problem is to solve a space and time dependent unknown, with boundary and initial conditions. Namely:

$$\begin{cases} \frac{d\boldsymbol{u}(t)}{dt} = L(\boldsymbol{u}(t)) + N(\boldsymbol{u}(t)) & \\ \boldsymbol{u} = \boldsymbol{u}_0 & \text{at } t = 0 \\ \boldsymbol{u} = \boldsymbol{u}_D & \text{in } \partial\Omega \end{cases} \tag{2.2}$$

Classical Finite Element Method (FEM) can be used to solve the problem. This is called the "High Fidelity" or "High Resolution" model. In this thesis the term full-order model is used to refer to it. On every time step, one must obtain the value of the unknown at the nodes, and store it as a column in an arrange called Snapshot Matrix. Namely:

$$\boldsymbol{X} = [\boldsymbol{u}(t=1), \boldsymbol{u}(t=2), ..., \boldsymbol{u}(t=p)] \in \mathbb{R}^{n\times p} \tag{2.3}$$

The model reduction consists on taking the Singular Value Decomposition of the snapshot matrix $\boldsymbol{X}$. And as was done in the case of the picture, only take some modes to approximate the solution:

$$\boldsymbol{u}(t) \approx \phi_r \boldsymbol{q}(t) \qquad \boldsymbol{q} \in \mathbb{R}^r \tag{2.4}$$

Where $r << n$.

Finally, one solves for

$$\frac{d\boldsymbol{q}(t)}{dt} = \boldsymbol{\phi}_r^T L \phi_r^T \boldsymbol{q}(t) + \boldsymbol{\phi}_r^T N \phi_r^T \boldsymbol{q}(t) \tag{2.5}$$

2.3 Bar Example

Formulations taken mainly from [4]. A simple model of a 1-D Bar is designed to understand the concepts applied to the finite element context.

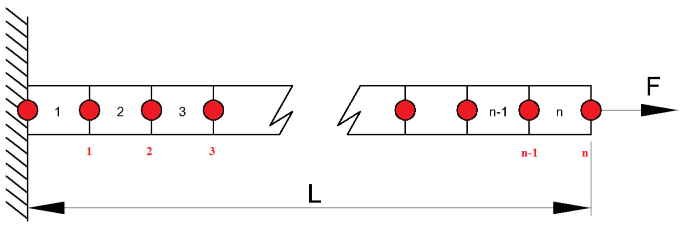


Figure 2.6: 1D bar system

The equation for the dynamic system is:

$$\boldsymbol{M}\ddot{\boldsymbol{u}} + \boldsymbol{C}\dot{\boldsymbol{u}} + \boldsymbol{K}\boldsymbol{u} = \boldsymbol{f} \tag{2.6}$$

Ignoring the matrix $\boldsymbol{C}$, related to damping effects, one is left with a system of the form:

$$\boldsymbol{M}\ddot{\boldsymbol{u}} + \boldsymbol{K}\boldsymbol{u} = \boldsymbol{f} \tag{2.7}$$

2.3.1 Time Integration Method: Newmark

In order to perform the time integration, the Newmark method is employed.

Newmark Method defines seven coefficients and uses them to update the solution in time. One must choose the value of the parameters α and β, and obtain the coefficients:

$$\alpha_0 = \frac{1}{\alpha dt^2} \quad \alpha_1 = \frac{\beta}{\alpha dt} \quad \alpha_2 = \frac{1}{\alpha dt} \quad \alpha_3 = \frac{1}{2\alpha} - 1 \tag{2.8}$$

$$\alpha_4 = \frac{\beta}{\alpha} - 1 \quad \alpha_5 = \frac{dt}{2}\left(\frac{\beta}{\alpha} - 2\right) \quad \alpha_6 = dt(1-\beta) \quad \alpha_7 = \beta dt$$

For $\alpha = 0.25$ and $\beta = 0.5$ numerical stability is ensured, as this is the trapezoidal rule or constant acceleration method. The accuracy is of $O(dt^2)$. The key is to solve the system for $\hat{\boldsymbol{K}}$ and $\hat{\boldsymbol{F}}$, as:

$$\hat{\boldsymbol{K}} = \boldsymbol{K} + \alpha_0 \boldsymbol{M} \tag{2.9}$$

$$\hat{\boldsymbol{F}} = \boldsymbol{F} + \boldsymbol{M}(\alpha_0 \boldsymbol{u_n} + \alpha_2 \dot{\boldsymbol{u}}_{\boldsymbol{n}} + \alpha_3 \ddot{\boldsymbol{u}}_{\boldsymbol{n}}) \tag{2.10}$$

The solution for displacement is obtained by solving the system:

$$\hat{\boldsymbol{K}}\boldsymbol{u}_{n+1} = \hat{\boldsymbol{F}} \tag{2.11}$$

The expression for velocity and acceleration are obtained as:

$$\ddot{\boldsymbol{u}}_{\boldsymbol{n+1}} = \alpha_0(\boldsymbol{u_{n+1}} - \boldsymbol{u_n}) - \alpha_2 \dot{\boldsymbol{u}}_{\boldsymbol{n}} - \alpha_3 \ddot{\boldsymbol{u}}_{\boldsymbol{n}} \tag{2.12}$$

$$\dot{\boldsymbol{u}}_{\boldsymbol{n+1}} = \dot{\boldsymbol{u}}_{\boldsymbol{n}} + \alpha_6 \ddot{\boldsymbol{u}}_{\boldsymbol{n}} + \alpha_7 \ddot{\boldsymbol{u}}_{\boldsymbol{n+1}} \tag{2.13}$$

Finally, the initial values are updated and the next time step is computed $(\cdot)_n = (\cdot)_{n+1}$. The algorithm to simulate the bar proceeds as follows:

```
Data: dt, NumberElements, $\alpha$ and $\beta$;
Obtain Newmark Coefficients;
Obtain Mass and Stiffness Matrices;
Initialize $\boldsymbol{u}$, $\dot{\boldsymbol{u}}$, and $\ddot{\boldsymbol{u}}$ to zero;
Obtain $\hat{\boldsymbol{K}}$;
while current time < final time do
    Calculate $\hat{\boldsymbol{F}}$;
    Solve the system $\hat{\boldsymbol{K}}\boldsymbol{u_{n+1}} = \hat{\boldsymbol{F}}$;
    Obtain $\ddot{\boldsymbol{u}}_{\boldsymbol{n+1}} = \alpha_0(\boldsymbol{u_{n+1}} - \boldsymbol{u_n}) - \alpha_2\dot{\boldsymbol{u}}_{\boldsymbol{n}} - \alpha_3\ddot{\boldsymbol{u}}_{\boldsymbol{n}}$ ;
    Obtain $\dot{\boldsymbol{u}}_{\boldsymbol{n+1}} = \dot{\boldsymbol{u}}_{\boldsymbol{n}} + \alpha_6\ddot{\boldsymbol{u}}_{\boldsymbol{n}} + \alpha_7\ddot{\boldsymbol{u}}_{\boldsymbol{n+1}}$;
    Update: $\boldsymbol{u_n} = \boldsymbol{u_{n+1}}$, $\dot{\boldsymbol{u}}_{\boldsymbol{n}} = \dot{\boldsymbol{u}}_{\boldsymbol{n+1}}$, $\ddot{\boldsymbol{u}}_{\boldsymbol{n}} = \ddot{\boldsymbol{u}}_{\boldsymbol{n+1}}$
end
```

Algorithm 1: Simulate a 1D bar using Newmark Method

Figure 2.7 shows the solution when running a script for 20 elements, dt=0.01, total time = 10, and a static force applied at the end of the bar of 0.2.

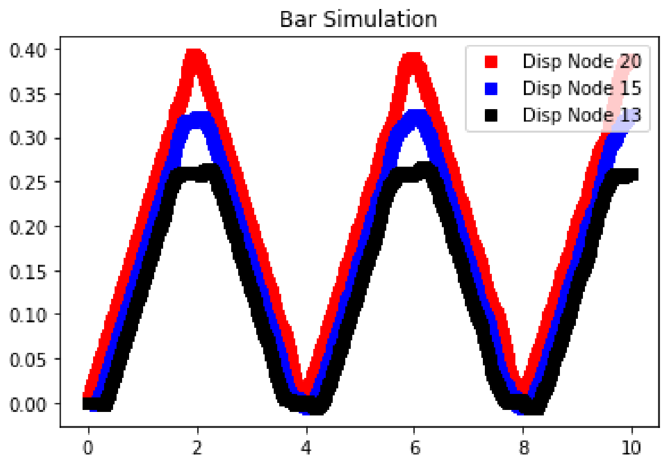


Figure 2.7: Solution for displacement of the 1D bar using Newmark method

2.3.2 Modal Analysis

One can easily obtain the eigenvalues and eigenvectors of this system, which in turn are the reduced basis for the analysis. The eigenvector matrix contains the modes, and the eigenvalues are an indicator of the influence of those modes on the response of the system.

Equation 2.7 is completely equivalent to:

$$(\boldsymbol{\phi}^T \boldsymbol{M}\boldsymbol{\phi}\ddot{\boldsymbol{u}}) + (\boldsymbol{\phi}^T \boldsymbol{K}\boldsymbol{\phi})\boldsymbol{u} = \boldsymbol{\phi}^T \boldsymbol{f} \tag{2.14}$$

Where $\boldsymbol{\phi}$ is the matrix of eigen vectors. If one takes only some eigenvectos, the resulting system is an approximation of smaller dimensions than the original. The reduced version of the matrices are:

$$\boldsymbol{M}^* = \phi^T \boldsymbol{M}\phi; \quad \boldsymbol{K}^* = \phi^T \boldsymbol{M}\phi; \quad \boldsymbol{f}^* = \phi^T \boldsymbol{F} \tag{2.15}$$

The reduced system is then:

$$\boldsymbol{M}^*\ddot{\boldsymbol{q}} + \boldsymbol{K}^*\boldsymbol{q} = \boldsymbol{f}^* \tag{2.16}$$

Algorithm 1 can be applied on the reduced system by only substituting the matrices, by the reduced versions and projecting back to the fine scale at the end of each time step, see Algorithm 2.

```
Data: dt, NumberElements, $\alpha$ and $\beta$;
Obtain Newmark Coefficients;
Obtain Mass and Stiffness Matrices;
Obtain Eigenvectors and Eigenvalues;
Obtain Reduced Matrices;
Initialize $\boldsymbol{q}$, $\dot{\boldsymbol{q}}$, and $\ddot{\boldsymbol{q}}$ to zero;
Obtain $\hat{\boldsymbol{K}}$ using $\boldsymbol{K}*and\boldsymbol{M}*$;
while current time < final time do
    Calculate $\hat{\boldsymbol{F}}$;
    Solve the system $\hat{\boldsymbol{K}}\dot{\boldsymbol{q}}_{n+1} = \hat{\boldsymbol{F}}$;
    Obtain $\ddot{\boldsymbol{q}}_{n+1} = \alpha_0(\boldsymbol{q}_{n+1} - \boldsymbol{q}_n) - \alpha_2\dot{\boldsymbol{q}}_n - \alpha_3\ddot{\boldsymbol{q}}_n$ ;
    Obtain $\dot{\boldsymbol{q}}_{n+1} = \dot{\boldsymbol{q}}_n + \alpha_6\ddot{\boldsymbol{q}}_n + \alpha_7\ddot{\boldsymbol{q}}_{n+1}$;
    Update: $\boldsymbol{q}_n = \boldsymbol{q}_{n+1}$, $\dot{\boldsymbol{q}}_n = \dot{\boldsymbol{q}}_{n+1}$, $\ddot{\boldsymbol{q}}_n = \ddot{\boldsymbol{q}}_{n+1}$;
    Project to fine basis: $\boldsymbol{u} = \phi\boldsymbol{q}$, $\dot{\boldsymbol{u}} = \phi\dot{\boldsymbol{q}}$, $\ddot{\boldsymbol{u}} = \phi\ddot{\boldsymbol{q}}$
end
```

Algorithm 2: Simulate a 1D bar using Newmark Method and Modal Analysis

One can select to use different amounts of modes in the simulation with modal analysis. The resulting displacement when using from 1 to 4 modes is compared to the original model in Figure 2.8.

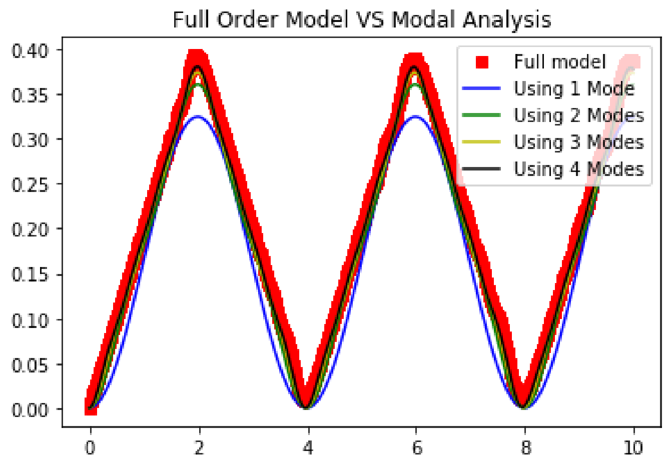


Figure 2.8: Solution obtained for the full-order model, compared to the reconstruction using different amounts of modes from eigen-value decomposition

The influence of the first modes is more important than the influence of the last modes. Moreover, the amplitude of the first modes is larger and their frequency is smaller. It is possible to visualize each of the modes by running a simulation using only the selected mode for the reduction. Figure 2.9 shows the modes for the modal analysis.

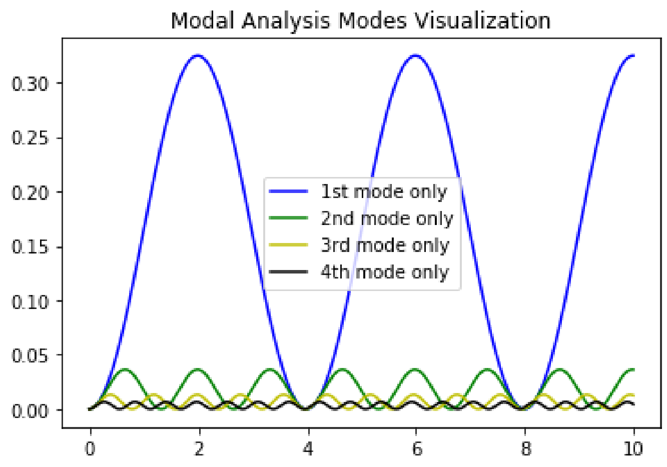


Figure 2.9: Visualization of the individual modes of modal analysis

2.3.3 Proper Orthogonal Decomposition

The first step, as already discussed, is running the simulation with the full-order model, and obtaining the snapshot matrix. The SVD is applied to the matrix, and a given amount of vectors of the matrix $\boldsymbol{U}$ are employed for projecting the system onto, and solving it.

The resulting system is exactly the same as Equation 2.16. And although the modes $\boldsymbol{\phi}$ are obtained following very different approaches, the same algorithm can be used to solve the system (Algorithm 2).

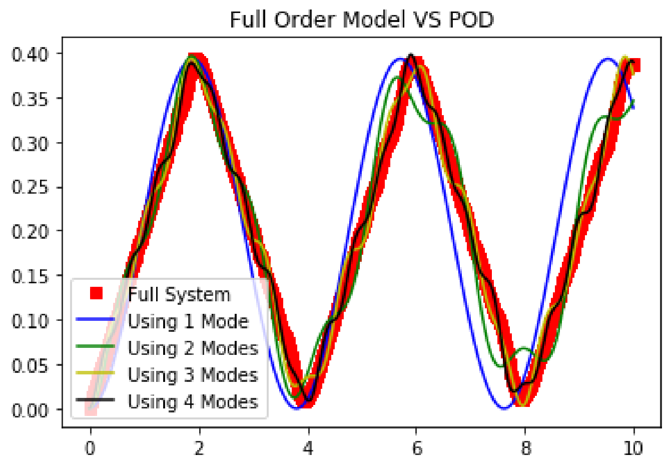


Figure 2.10: Solution for the full-order model, compared to the reconstruction using different amounts of modes from SVD

Similarly to modal analysis, the modes obtained from the SVD decrease in amplitude and increase in frequency. Notice however that the frequencies do not match as much as they do with modal analysis.

Notice that the reconstruction using only 1 SVD mode matches better the amplitude of the full-order model solution, than that of 1 Modal Analysis mode. However, the frequency of the full-order model is better captured with modal analysis than with SVD.

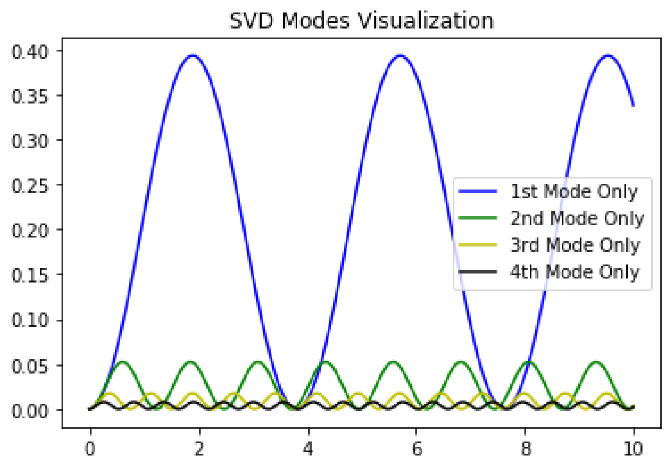


Figure 2.11: Visualization of the individual SVD modes

2.3.4 Nonlinear Bar

Under a scenario, in which the bar presents a non-linear response, the modal analysis reasoning fails. However, the POD is still valid element by element.

In order to solve the full-order model with a nonlinear response, it is convenient to rewrite the system in residual form:

$$\boldsymbol{M}\ddot{\boldsymbol{u}} - \boldsymbol{r}(\boldsymbol{x}, t) = 0; \qquad \boldsymbol{r} = \boldsymbol{f}_{ext}(\boldsymbol{u}, t) - \boldsymbol{K}(\boldsymbol{u})\boldsymbol{u} \tag{2.17}$$

To solve the nonlinearity, a Newton Raphson technique can be used, for which the tangent matrix has to be derived.

Defining a dynamic residual as:

$$\psi(\boldsymbol{u}_{n+1}) = \boldsymbol{r}_{n+1} - \boldsymbol{M}\ddot{\boldsymbol{u}}_{n+1} \tag{2.18}$$

The tangent matrix is obtained by differentiating ψ as:

$$\boldsymbol{K}_{tan}^{dyn} := -\frac{\partial \psi}{\partial \boldsymbol{u}_{n+1}} = -\frac{\partial \boldsymbol{r}_{n+1}}{\partial \boldsymbol{u}_{n+1}} + \frac{\partial \boldsymbol{M}\ddot{\boldsymbol{u}}_{n+1}}{\partial \boldsymbol{u}_{n+1}} \tag{2.19}$$

The first term in (2.19) is simply the static tangent operator $\boldsymbol{K}$. Mass, can be assumed to be constant and using:

$$\frac{\partial \ddot{\boldsymbol{u}}}{\partial \boldsymbol{u}_{n+1}} = \frac{1}{\alpha \Delta t^2}\boldsymbol{I} = \alpha_0 \boldsymbol{I} \tag{2.20}$$

The derivative of mass is readily computed. The dynamic tangent operator is then:

$$\boldsymbol{K}_{tan}^{dyn} = \boldsymbol{K}_{tan} + \alpha_0 \boldsymbol{M} \tag{2.21}$$

The algorithm for solving the non linear 1D bar is then:

```
Data: dt, NumberElements, $\alpha$ and $\beta$;
Obtain Newmark Coefficients;
Obtain the Mass Elemental Matrix;
Initialize $\boldsymbol{u}_n$, $\dot{\boldsymbol{u}}_{\boldsymbol{n}}$, $\ddot{\boldsymbol{u}}_{\boldsymbol{n}}$,$\boldsymbol{u}_{n+1}$, $\dot{\boldsymbol{u}}_{\boldsymbol{n+1}}$, and $\ddot{\boldsymbol{u}}_{n+1}$ to zero;
while current time < final time do
    Set $\dot{\boldsymbol{u}}_n = \dot{\boldsymbol{u}}_{n+1}$, $\ddot{\boldsymbol{u}}_n = \ddot{\boldsymbol{u}}_{\boldsymbol{n+1}}$, and $\boldsymbol{u}_{n+1} = \boldsymbol{u}_n + (\dot{\boldsymbol{u}}\Delta t)$;
    Obtain $\ddot{\boldsymbol{u}}_{\boldsymbol{n+1}} = \alpha_0(\boldsymbol{u}_{\boldsymbol{n+1}} - \boldsymbol{u}_{\boldsymbol{n}}) - \alpha_2\dot{\boldsymbol{u}}_{\boldsymbol{n}} - \alpha_3\ddot{\boldsymbol{u}}_{\boldsymbol{n}}$ ;
    Obtain $\dot{\boldsymbol{u}}_{\boldsymbol{n+1}} = \dot{\boldsymbol{u}}_n + \alpha_6\ddot{\boldsymbol{u}}_{n+1} + \alpha_7\ddot{\boldsymbol{u}}_{n+1}$;
    while residual > Tolerance do
        for Number of elements do
            Obtain the elemental external and internal forces $\boldsymbol{F}_{ext}$, $\boldsymbol{Ku}$;
            Obtain the elemental residual $\psi_{el} = \boldsymbol{F}_{ext} - \boldsymbol{Ku} - \boldsymbol{M}_{\boldsymbol{el}} * \ddot{\boldsymbol{u}}_{\boldsymbol{el n+1}}$;
            Obtain the elemental dynamic Tangent
             $\boldsymbol{K}_{\boldsymbol{el}}{}_{dyn}^{tan} = \boldsymbol{K}_{\boldsymbol{el}\,static} + \alpha_0\boldsymbol{M}_{\boldsymbol{el}}$;
            Assemble to $K_{dyn}$ and $\psi$
            $\Pi\boldsymbol{K}_{dyn}^{tan} = \boldsymbol{K}_{\boldsymbol{el}}{}_{dyn}^{tan}$;
            $\Pi\boldsymbol{\psi} = \boldsymbol{\psi}_{\boldsymbol{el}}$;
        end
        Correct $\boldsymbol{u}_{n+1}^{k+1} = \boldsymbol{u}_{n+1}^{k} + (K_{dyn}^{tan})^{-1}\boldsymbol{\psi}$;
        Obtain $\ddot{\boldsymbol{u}}_{\boldsymbol{n+1}}^{\boldsymbol{k+1}} = \alpha_0(\boldsymbol{u}_{\boldsymbol{n+1}}^{\boldsymbol{k}} - \boldsymbol{u}_{\boldsymbol{n}}) - \alpha_2\dot{\boldsymbol{u}}_{\boldsymbol{n}} - \alpha_3\ddot{\boldsymbol{u}}_{\boldsymbol{n}}$ ;
        Obtain $\dot{\boldsymbol{u}}_{\boldsymbol{n+1}}^{\boldsymbol{k+1}} = \dot{\boldsymbol{u}}_n + \alpha_6\ddot{\boldsymbol{u}}_n + \alpha_7\ddot{\boldsymbol{u}}_{n+1}^{k}$;
        Check Convergence;
    end
    Update: $\boldsymbol{u}_n = \boldsymbol{u}_{n+1}$,
end
```

Algorithm 3: Simulate a 1D bar using Newmark Method for a nonlinear bar

Figures 2.12, 2.13, and 2.14 show the solution obtained for displacement, velocity and acceleration respectively on a 1D bar under the nonlinear response assumption: $\boldsymbol{f}_{int} = \boldsymbol{u}^2 - 1$.

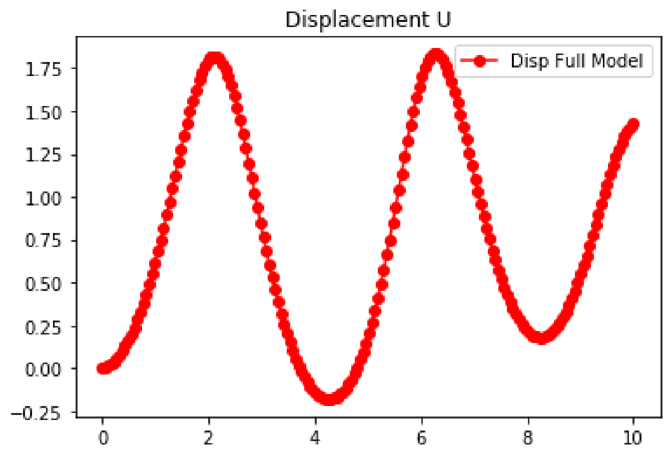


Figure 2.12: Results for displacement

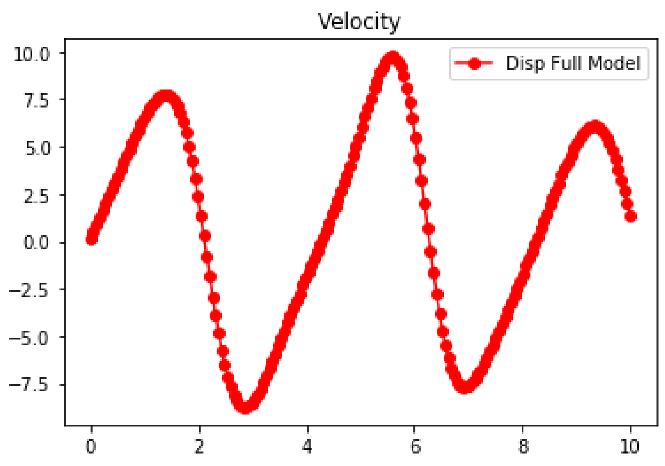


Figure 2.13: Results for velocity

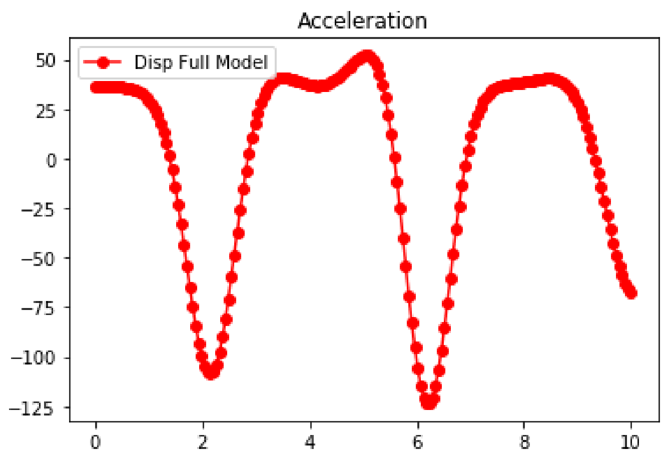


Figure 2.14: Results for acceleration

2.3.5 Nonlinear Bar ROM

The POD can be applied to the nonlinear system by taking the snapshot matrix from the full-order model solution and from it, applying the SVD, and obtaining the reduced basis $\boldsymbol{\phi}$.

Algorithm 4 presents the steps necessary to carry out the ROM simulation:

```
Data: dt, NumberElements, $\alpha$ and $\beta$, Snapshot Matrix;
Obtain Newmark Coefficients;
Obtain the Mass Elemental Matrix;
Obtain the SVD of the Snapshot Matrix;
Truncate matrix $\boldsymbol{U}$ and use it as basis;
Initialize $\boldsymbol{q}_n$, $\dot{\boldsymbol{q}}_{\boldsymbol{n}}$, $\ddot{\boldsymbol{q}}_{\boldsymbol{n}}$,$\boldsymbol{q}_{n+1}$, $\dot{\boldsymbol{q}}_{\boldsymbol{n+1}}$, and $\ddot{\boldsymbol{q}}_{n+1}$ to zero;
while current time < final time do
    Set $\dot{\boldsymbol{q}}_n = \dot{\boldsymbol{q}}_{n+1}$, $\ddot{\boldsymbol{q}}_n = \ddot{\boldsymbol{q}}_{\boldsymbol{n+1}}$, and $\boldsymbol{q}_{n+1} = \boldsymbol{q}_n + (\dot{\boldsymbol{q}}\Delta t)$;
    Obtain $\ddot{\boldsymbol{q}}_{\boldsymbol{n+1}} = \alpha_0(\boldsymbol{q}_{\boldsymbol{n+1}} - \boldsymbol{q}_{\boldsymbol{n}}) - \alpha_2\dot{\boldsymbol{q}}_{\boldsymbol{n}} - \alpha_3\ddot{\boldsymbol{q}}_{\boldsymbol{n}}$ ;
    Obtain $\dot{\boldsymbol{q}}_{\boldsymbol{n+1}} = \dot{\boldsymbol{q}}_n + \alpha_6\ddot{\boldsymbol{q}}_{n+1} + \alpha_7\ddot{\boldsymbol{q}}_{n+1}$;
    while residual > Tolerance do
        for Number of elements do
            Obtain the elemental reduction basis $\boldsymbol{\phi}_{el}$;
            Obtain $\boldsymbol{u}_{\boldsymbol{el}n+1} = \boldsymbol{\phi}\boldsymbol{q}_{n+1}$;
            Obtain $\dot{\boldsymbol{u}}_{\boldsymbol{el}n+1} = \boldsymbol{\phi}\dot{\boldsymbol{q}}_{n+1}$;
            Obtain the elemental external and internal forces $\boldsymbol{F}_{ext}$, $\boldsymbol{K}\boldsymbol{u}_n$;
            Obtain the elemental reduction basis $\boldsymbol{\phi}_{el}$;
            Obtain the elemental residual $\boldsymbol{psi}_{el} = \boldsymbol{F}_{ext} - \boldsymbol{K}\boldsymbol{u} - \boldsymbol{M}_{\boldsymbol{el}} * \ddot{\boldsymbol{u}}_{\boldsymbol{eln+1}}$;
            Obtain the ROM elemental residual $\psi_{ROM} += \boldsymbol{\phi}^T\boldsymbol{\psi}_{\boldsymbol{el}}\boldsymbol{\phi}$;
            Obtain the elemental dynamic Tangent
            $\boldsymbol{K}_{\boldsymbol{el}}{}^{tan}_{dyn} = \boldsymbol{K}_{\boldsymbol{el}static} + \alpha_0\boldsymbol{M}_{\boldsymbol{el}}$;
            Obtain the ROM elemental dynamic Tangent
            $\boldsymbol{K}_{\boldsymbol{ROM}}{}^{tan}_{dyn} = \boldsymbol{\phi}^T\boldsymbol{K}_{\boldsymbol{el}}{}^{tan}_{dyn}\boldsymbol{\phi}$;
        end
        Correct $\boldsymbol{q}^{k+1}_{n+1} = \boldsymbol{q}^{k}_{n+1} + (\boldsymbol{K}^{tan}_{dyn})^{-1}\boldsymbol{\psi}$;
        Obtain $\ddot{\boldsymbol{q}}^{\boldsymbol{k+1}}_{\boldsymbol{n+1}} = \alpha_0(\boldsymbol{q}^{\boldsymbol{k}}_{\boldsymbol{n+1}} - \boldsymbol{q}_{\boldsymbol{n}}) - \alpha_2\dot{\boldsymbol{q}}_{\boldsymbol{n}} - \alpha_3\ddot{\boldsymbol{q}}_{\boldsymbol{n}}$ ;
        Obtain $\dot{\boldsymbol{q}}^{\boldsymbol{k+1}}_{\boldsymbol{n+1}} = \dot{\boldsymbol{q}}_n + \alpha_6\ddot{\boldsymbol{q}}_n + \alpha_7\ddot{\boldsymbol{q}}^{k}_{n+1}$;
        Check Convergence;
    end
    Update: $\boldsymbol{q}_n = \boldsymbol{q}_{n+1}$,
end
```

Algorithm 4: Simulate a 1D bar using Newmark Method for ROM nonlinear bar

Figures 2.15, 2.16, and 2.17 show the a comparison of the solutions obtained for the full-order model and ROM.

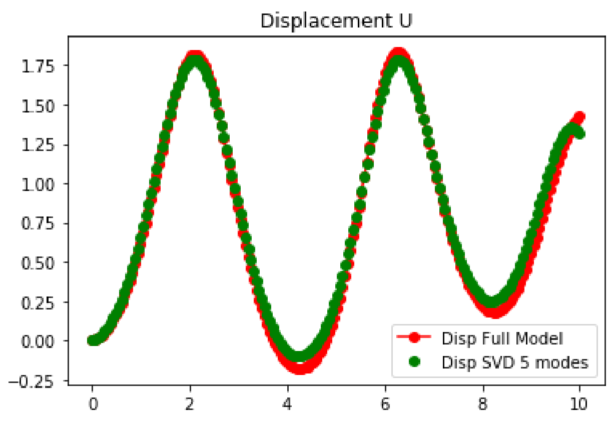


Figure 2.15: Comparison of results for displacement

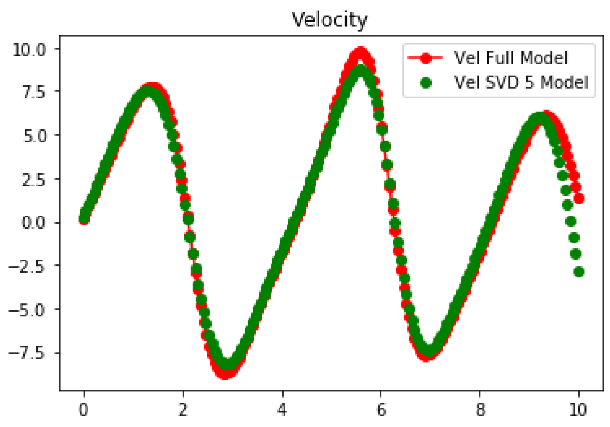


Figure 2.16: Comparison of results for velocity

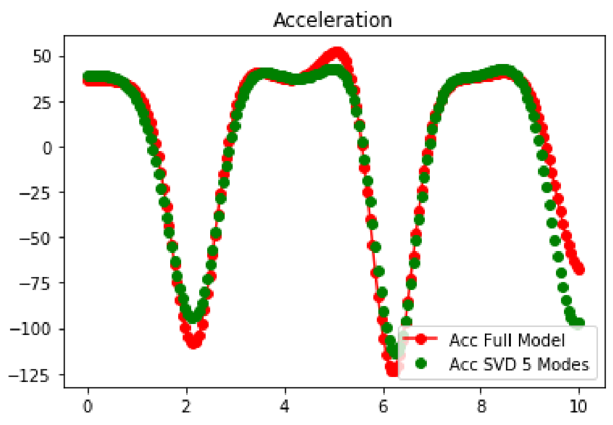


Figure 2.17: Comparison of results for acceleration

2.3.6 Frobenius Norm

The selection of the number of modes to accurately perform the simulations can be done following different criteria [5]. In this thesis, the Frobenius Norm is used. It is defined as:

$$||\boldsymbol{A}||_{FR} = \sqrt{\sum_i \sum_j A_{ij}^2} \tag{2.22}$$

In the context of Reduced Order Models, the values of the diagonal matrix $\boldsymbol{\Sigma}$, which are such that $\sigma_i > \sigma_{i+1}$, can be used to obtain the minimum number of modes to represent the behavior of the full-order system. Such a measure is of the following form:

$$\sqrt{\frac{\sum_{i=N+1}^{end} \sigma_i^2}{\sum_{i=1}^{N} \sigma_i^2}} \leq tol \tag{2.23}$$

Where N is the number of modes to take.

In the bar example, the plot of the matrix $\boldsymbol{\Sigma}$ is shown in Figure 2.18

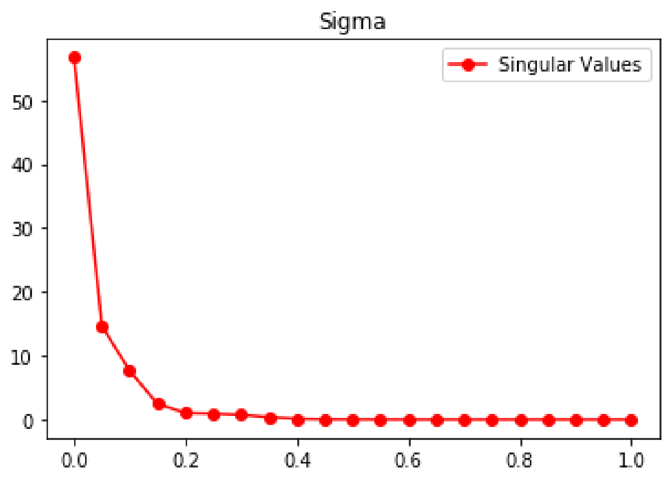


Figure 2.18: Singular values for the nonlinear bar system

Notice that there are as many singular values as degrees of freedom in the full scale system. Small singular values correspond to modes that are not only not influencing the system, but polluting the representation of it [6]. This is demonstrated in the results sections.

By applying 2.23, one can see that the minimum number of modes to use is 6, provided a tolerance of $1e-6$.

The Python codes for the 1D bar can be found in the appendix.

Chapter 3
Methodology

In this chapter, the software and tools employed in this thesis are described. The two programs most widely used during the development of this thesis have been Kratos Multiphysics and GiD.

On the one hand, the role GiD had for this thesis has been a pre- and postprocessor. It has been used to obtain the geometries, and observe the results, but no development or implementation was undertaken on it.

On the other hand, Kratos Multiphysics, the solver, has been extended, taking advantage of its existing structures, to be able run ROM simulations. The general structure of the code, and the implementations performed are explained in this chapter.

Other tools explained in this chapter include COMPSs, a program developed at the Barcelona Supercomputer Center for parallel computing . This has been used to create training cases for the ROM.

3.1 GiD

The Software GiD is a pre- and postprocessor developed at CIMNE. In this thesis it is used to create the geometries, mesh them, and to display the results obtained in Kratos. The selection of GiD over similar programs, is due to its complete compatibility and support for Kratos.

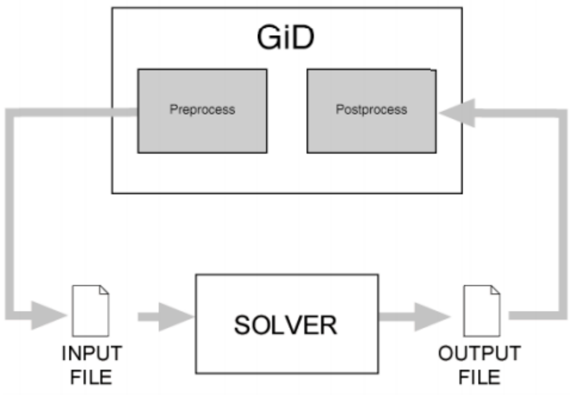


Figure 3.1: Sketch of the GiD on the working flow

From GiD one can lauch the "Kratos Problemtype". This is the graphical user interface of Kratos-Multiphysics that the user of the software can enter to create a model, solve it using Kratos, and visualize the results.

In this thesis, the Kratos application of interest is the Convection-Diffusion application, both 2D and 3D.

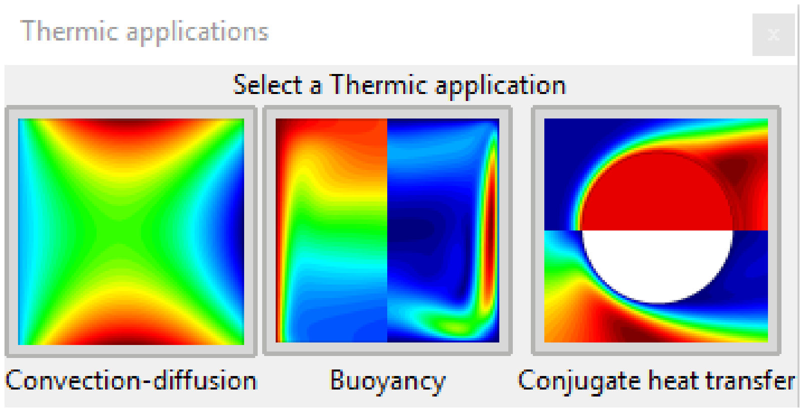


Figure 3.2: Thermic Applications available in Kratos

Inside the Convection-Diffusion application interface, one can follow all the steps to prepare a model: generation of geometry, imposition of boundary conditions, meshing, etc. Alternatively, it is also possible to press the "Example" button, which builds a test case. After that, when one presses "solve" all the necessary files to run a Kratos problem are created (MainKratos.py, specific_problem.mdpa, and ProjectParameter.json). The software GiD runs the MainKratos.py file and the results files are generated. One can then switch to the post-process interface and visualize the results.

Figure 3.3 shows the results obtained when solving the example problem for the convection diffusion application in 2D. The boundary conditions for this example are prescribed temperature on the left side, prescribed flux on the upper and lower sides, and prescribed radiation condition on the right side. The same model is solved at the end of the results section, using the ROM solver implemented (statically).

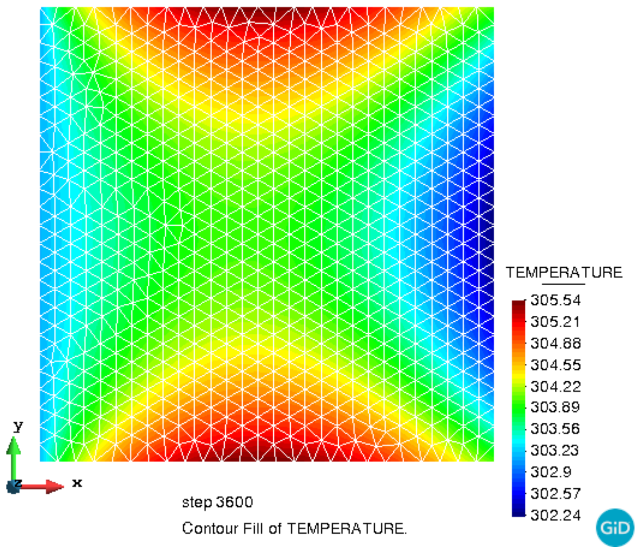


Figure 3.3: Example of visualization of a model in GiD

3.2 Kratos Multiphysics

Kratos is a powerful FEM code written in C++. It is open source and counts with an extensive Python interface. It started as a project at the International Centre for Numerical Methods in Engineering (CIMNE) in 2002. Over the years, many companies and universities incorporated to the development of Kratos. Today, Kratos Multiphysics is a mature and large code with more than 2.5 million lines, and it is supported by many companies and universities, among which are: The Technical University of Munich (TUM), Airbus, Siemens, Altair. The code is available in GitHub[1].

3.2.1 General Structure

Kratos is coded in an Object Oriented fashion. Object orientation consists on developing objects that count with a number of attributes and an interface to interact with them. The design of Kratos aims for modularity and extensibility, as well as efficiency, good performance and team-work-promptness.

The main classes of Kratos can be seen in figure 3.4

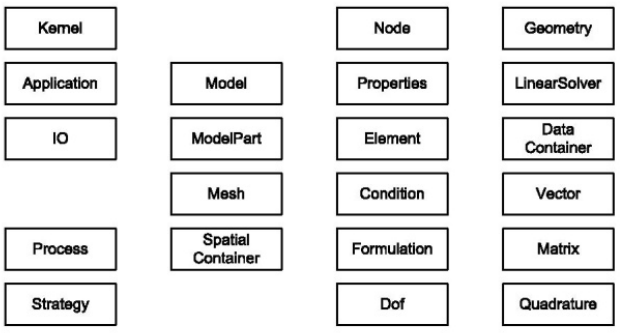


Figure 3.4: Main classes in Kratos

The design is based on the Finite Element Method basic structure. The selection of such a layout is due to the fact that the original idea when developing Kratos was to create a multidisciplinary FEM code; moreover, people working in the first versions were in general already familiar with FEM [7] [8].

Kratos employs a multi-layer structure, in which each layer interacts only with objects in its layer or in layers below it. This feature allows for users with different programming skills and different needs, to interact with the least amount of layers possible. Figure 3.5 shows the basic layer structure in Kratos.

[1]https://github.com/KratosMultiphysics/Kratos

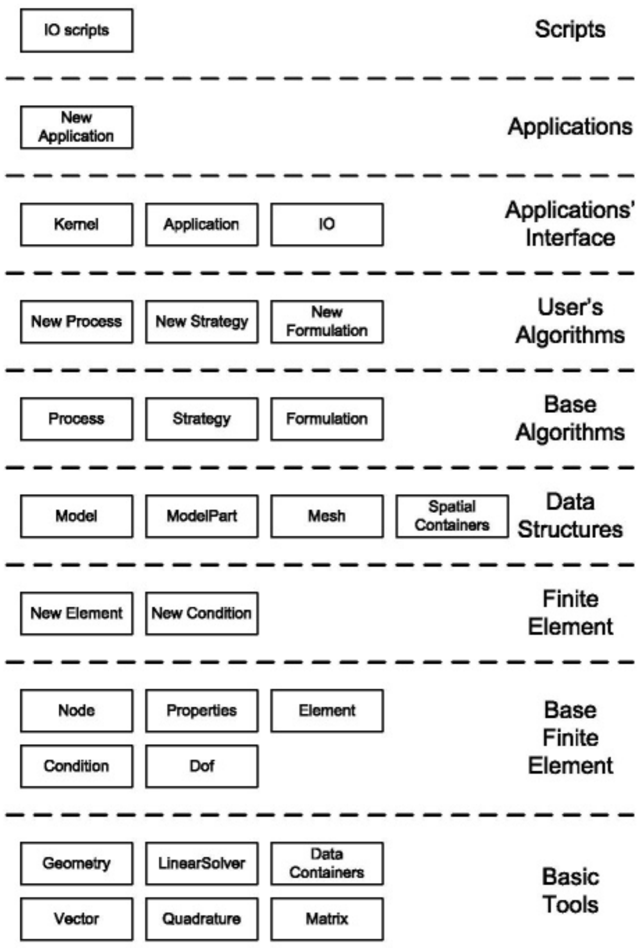


Figure 3.5: Layer structure in Kratos. Programming expertise decreases from bottom to fop, while FEM skills increase

3.3 Parts of a Kratos Problem

Several files are created automatically when running a Kratos example from the GiD interface. Some of those files are only needed by the GUI, however each Kratos problem must count at least with the three files shown in figure 3.6.

MainKratos.py
ProjectParameters.json
Part_name.mdpa

Figure 3.6: Files for a Kratos problem

These files contain specific data about the problem to be solved with Kratos; like the geometry, material, or tolerances. This architecture allows for changes of parameters to be performed quickly without having to enter and re-compile Kratos. These three main files are explained further in the next sections.

Another important component is the Python solver. Each Kratos problem uses a Python solver, however unlike the MainKratos.py, ProjectParameters.json, and Part_name.mdpa; the Python solver is not usually contained in the folder of the specific problem. Rather, it is in the Kratos application folder, and unless one needs to modify how the problem is being solved (which is the case here), the general user of Kratos does not modify it.

3.3.1 MainKratos.py

The main tasks the MainKratos.py performs are:

- Load Kratos and the applications needed
- Read the problem data, project parameters and create the model part in Kratos: mesh, elements, materials, variables, degrees of freedom...
- Define the BuilderSolver used
- Call the solver
- Write a file with the mesh and results

The MainKratos.py that is produced by GiD is simple and it performs only the mentioned taks. On top of that, one can edit the MainKratos.py to perform modifications in the model before running, select a solver different to the default one, print a given value at a given node, or save to a file the nodal solution values.

For the implementation of ROM in Kratos, this flexibility has been exploited, as is discussed in the section 3.5.

3.3.2 ProjectParameters.json

The ProjectParameters.json file contains information on the tolerances, start and end time for the simulation, the name of the model, the processes, and any other parameter.

The term "processes" here means scripts in Python that are called, not from the MainKratos.py; but from the ProjectParameters.json. The most common processes are:

- Assign scalar quantity: This is used for example to impose a fixed temperature or a flux on a certain face.
- Assign vector quantity: In this thesis, these kinds of quantities are not dealt with, but this process can be used to impose a certain displacement.
- Apply thermal face: This is the process used when applying a radiation condition on a certain face. It takes as inputs ambient temperature, emissivity and convection coefficient.
- Generate the GiD or VTK output: The creation of the output files to be read with GiD or Paraview, for example. Other outputs like HDF5 are also possible.

3.3.3 Part_name.mdpa

The .mdpa file (the extension name comes from "model part") contains information on the nodal coordinates and connectivities. It also declares the elements and conditions, as well as the materials assigned to them.

In Kratos, the terms Elements and Conditions are used to define similar entities. Elements are self explanatory, since they match the classical finite element definition. Conditions, on the other hand, are not so obvious. In Kratos, Conditions refer to elements of one spatial dimension less than the number of spatial dimensions of the problem at hand. That is, the face or line elements on the boundaries of the domain.

Many types of elements and conditions are defined in Kratos, which vary not only in their geometry, but also in the methods to calculate their contribution to the global system. In this thesis the elements and conditions used are declared in the Part_name.mdpa; and they are Laplacian Elements, and ThermalFace Conditions.

3.3.4 Python Solver

The PythonSolver is responsible for everything related to the physics of the problem (e.g. how to setup the system of equations). It declares the Kratos Variables(e.g. "TEMPERATURE", "DISPLACEMENT_X"), sets the Builder and Solver and Solution strategy to follow.

During the development of this thesis project, a simple static Python solver was created, and new functionalities were added as they were needed and more understanding of Kratos was achieved. This static solver was called to run the problems to train the ROM; and obtain the snapshots.

A Python rom solver was also created. This solver calls the specific rom builder and solver inside kratos, which is coded in C++. More details on the C++ sections of the Kratos that were used in this thesis are mentioned in the next sections.

3.4 Inside Kratos

The Python interface is useful to develop and modify models in an quick and flexible way. However, as it is the case for any interpreted language, it has the disadvantage of not being fast.

The computations in Kratos are performed by the compiled C++ code, therefore being very fast and efficient in comparison. Two very important classes in Kratos are the Builder and Solver and the Solution Strategy, which are described in the following subsections.

3.4.1 Builder and Solver

The builder and solver class is responsible for looping over the elements, assemble the global system and solving it.

One way to implement ROM in Kratos is to take advantage of this existing structure, and although other ways to do it exist, they would also be more intrusive. Therefore given the resources available, in this thesis the BuilderAndSolver class is the place where the ROM was implemented.

Static solver : ResidualBasedBlockBuilderAndSolver.h

This is the builder and solver that is used by the static_solver.py. It is also used by many Kratos applications, as it follows the classical computation of the stiffness matrix $\boldsymbol{K}$, and forcing vector $\boldsymbol{f}$ by looping on the elements. It contains a method to apply Dirichlet boundary conditions, and to calculate the RHS, which is useful for residual calculations during convergence-dependent processes like solving a nonlinear equation.

ROM solver: ROMBuilderAndSolver.h

The ROMBuilderAndSolver is derived from the base class BuilderAndSolver. Many characteristics of the base are kept, however, the global system is not assembled in the classical way. The system is composed of a dense matrix with dimensions equal to the number of modes selected. See figure 3.7.

Details on the implementation of ROM in the ROMBuilderAndSolver.h are given in section 3.5.3.

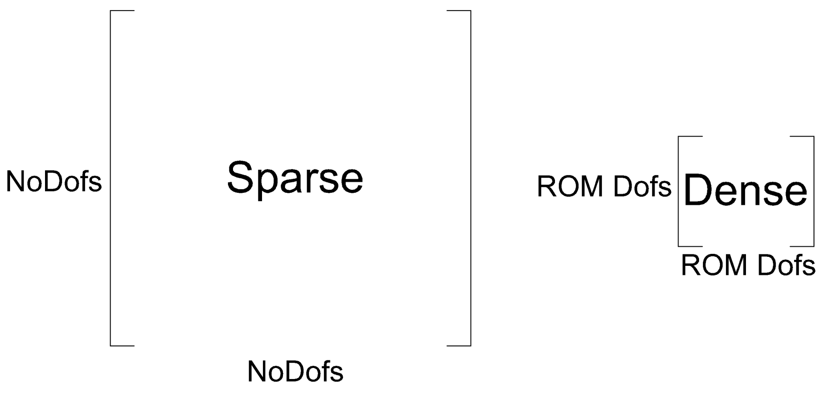


Figure 3.7: System size using the classical Builder and Solver, and the ROM one.

3.4.2 Solution Strategy

The solution strategy is the class that calls the builder and solver. It can be either linear, in which case the solution of the algebraic system is performed once; or nonlinear, for which iterations are performed until a suitable convergence criterion is converged.

In this thesis, the same solution strategy is used for both, the full-order model and the ROM. Such solution strategy is linear or nonlinear depending on the case studied. The builder and solver used is the only variation between the two models.

Both, the "builder and solver" and the "solution strategy" to use are declared at Python level by the solver.

3.5 Implementing ROM in Kratos

The process to have a reduced order model working in Kratos consists of many steps. First, one must obtain from the full-order model the information of the variable of interest in each node. For this thesis, the variable of interest is temperature.

There are some approaches for building the snapshot matrix. For dynamic problems, the information is captured at every time step, as it was done in the 1D bar example. For static problems, like the ones that are treated in this thesis using Kratos, the information is to be obtained for different values of the boundary conditions.

The section on Creating the Snapshots describes how the matrix of snapshots is created from Kratos, after which a SVD of it is to be taken, a further explanation is presented in the section Performing the SVD. While the selection of the number of modes and how they are imported back to Kratos is explained in the section Selecting the Modes.

Finally, section 3.5.3 explains how the ROM is implemented in Kratos. Which is done at the Builder and Solver level.

3.5.1 Creating the Snapshots

The format chosen to obtain and store the snapshot matrix is HDF5. HDF5 is a data format that is well suited to work with large and complex data.

There exists already an HDF5 application in Kratos, and taking advantage of it, an HDF5 process was created in a Python script. So that everything one needs to add to a given simulation´s ProjectParameters.json are the following lines:

```
{
    "Python_module" : "create_snapshots_hdf5_process",
    "kratos_module" : "KratosMultiphysics",
    "Parameters"    : {
    "model_part_name" : "ThermalModelPart",
    "variable_name"   : "TEMPERATURE"
    }
}
```

Three different strategies were followed to obtain the snapshot matrix. The first one was to set a pseudo time to modify the boundary conditions at each time step.

A second approach was to manually enter in the ProjectParameters.json the boundary conditions values, run one instance of the simulation, then enter again in the Project-Parameters.json to change the parameter, and run a second simulation. This is suitable when the number of simulation needed for training is not large, as it is the case for linear problems.

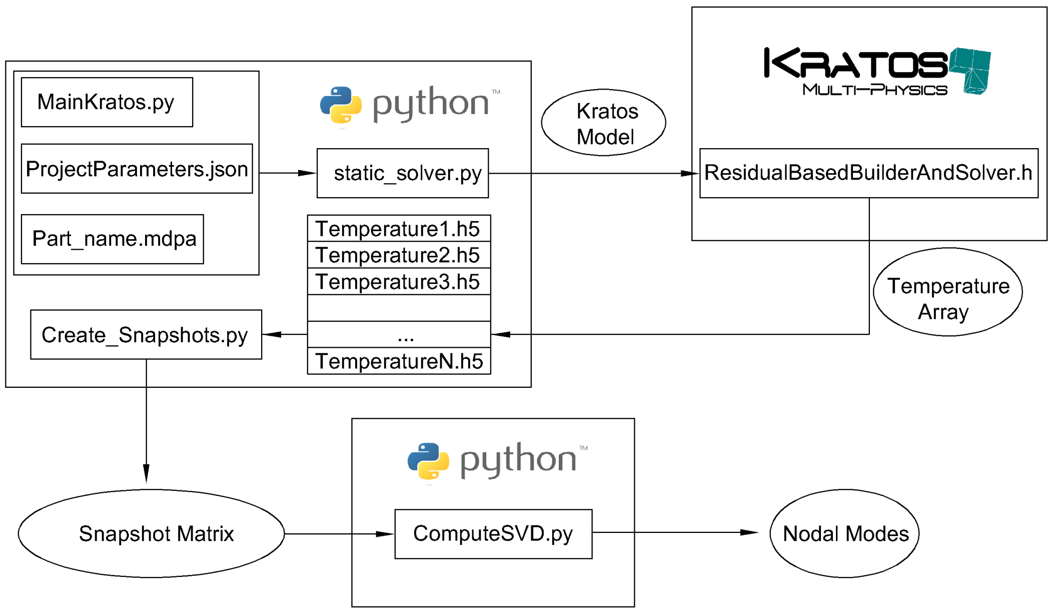


Figure 3.8: Work flow of the creation of reduction basis.

The third way for the creation of the snapshots is employed for nonlinear problems, where the number of simulations required to train the model is orders of magnitude larger than for linear ones. In this case, it is clear the importance of a tool to set and run simulations more efficiently. In this thesis the training of large amounts of simulations is done via the program COMPSs.

COMPSs

COMPS Superscalar (COMPSs) is a framework developed at the Barcelona Supercomputing Center (BSC). Its purpose is to facilitate the running in parallel of codes programmed following a sequential paradigm, by taking care of data distribution.

COMPSs is only available for Linux, and it was used to launch the training cases for the nonlinear problems studied.

When using COMPSs, the simulation is sent to many processors, and only at the end, one knows the information of them all. Figure 3.12, at the end of the chapter, demonstrates this idea.

Therefore, for the nonlinear simulation, only one HDF5 file is created containing the snapshot matrix with as many rows as DOFs and as many columns as cases studied.

3.5.2 Selecting the Modes

The way to select the number of modes to take into consideration in the simulation is via the Frobenius norm introduced already in section 2. The equation (Eq. 2.23, repeated here) dictating the number of modes to use is:

$$\sqrt{\frac{\sum_{i=N+1}^{end} \sigma_i^2}{\sum_{i=1}^{N} \sigma_i^2}} \leq tol$$

That is, one has to take enough modes to ensure that the sum of the square of the singular values from N+1 to the end, divided by the sum of the square of the singular values from 1 to N, where N is the number of modes to use, is less than a tolerance.

A Python script was created to build the reduced basis and save it in a NodalModes.json file. Then a modified version of MainKratos.py, say MainKratosROM.py, including the function ModifyInitialGeometry(), reads the NodalModes.json file, and imports the corresponding mode vector to each node in the geometry.

Notice some differences with the 1D bar example. Here, the ROM basis is imported to each node, while in the bar example the basis was multiplied directly by the elemental matrices. The building of the elemental reduced matrix is done in the Builder and Solver.

Another important point to have in mind is that, since the quantity of interest for the cases studied is a scalar, a vector is imported to each node. If the quantity of interest was vectorial, a matrix would be imported to each node, with as many columns as dimensions has the vectorial nodal quantity.

Figure 3.8 shows a sketch of the work-flow followed in order to obtain the snapshot matrix and the modes.

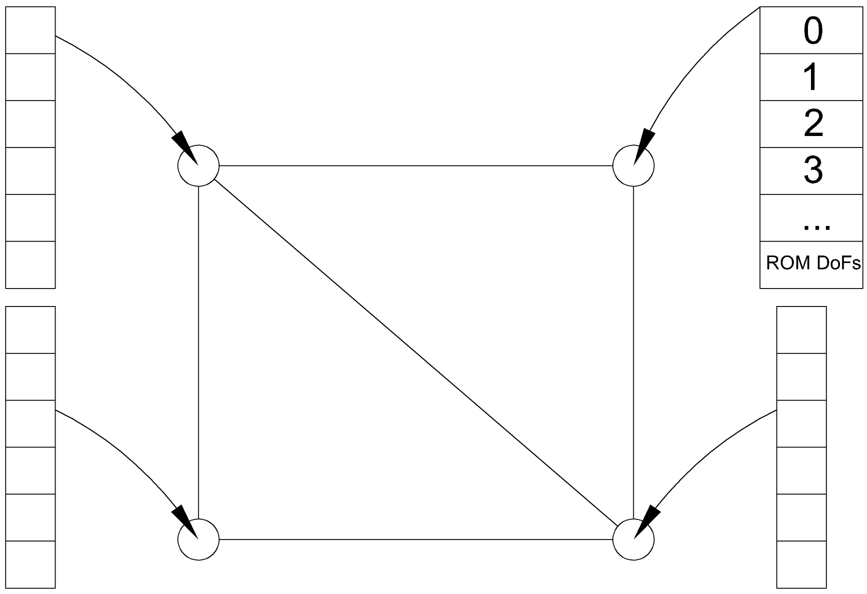


Figure 3.9: The modes vectors are imported to the corresponding nodes by the MainKratosROM.py

3.5.3 Implementation in the Builder and Solver

The ROM builder and solver was created taking advantage of the existing structure in Kratos.

A simple sketch of the class created is shown in the following chart:

ROMBuilderAndSolver.h

- **Inherits from:** BuilderAndSolver
- **Inherits by:** None
- **Attributes:**
  - A_{rom} The reduced LHS matrix
  - b_{rom} The reduced RHS vector
  - x_{rom} The reduced solution vector
  - Dx The solution vector
- **Method:**
  - *AssembleContribution()*
  - *ProjectToFineBasis()*
  - *BuildAndSolve()*

The ROM builder and solver is working as any other builder and solver inside Kratos. It is called by the solution strategy, and given the information of the Elements and Conditions it solves an algebraic system and returns a vector with the nodal solution. The process to produce such solution is, however, not standard.

AssembleContribution()
This function takes the elemental contribution to the LHS and RHS and performs the multiplication by the elemental reduced basis.

$$\boldsymbol{A^e}_{ROM} = \boldsymbol{\phi^e}^T \boldsymbol{A^e} \boldsymbol{\phi^e} \tag{3.1}$$

$$\boldsymbol{b^e}_{ROM} = \boldsymbol{\phi^e}^T \boldsymbol{b^e} \tag{3.2}$$

Then it assembles the elemental contribution to the global system. For this case, the assembling consists on the sum of all the elemental matrices. The global matrix has the same dimension as the elemental ones.

$$\boldsymbol{A}_{ROM} += \boldsymbol{A^e}_{ROM} \tag{3.3}$$

$$\boldsymbol{b}_{ROM} += \boldsymbol{b^e}_{ROM} \tag{3.4}$$

A very important aspect of the implementation is the way to treat the Fixed degrees of freedom. Usually, ROM is performed only on the free degrees of freedom. In this Kratos implementation the fixed degrees of freedom are not considered when constructing the elemental matrix $\boldsymbol{\phi}$ from the nodal basis vectors.

All of the elements have been assigned a reduced basis vector, including those with prescribed Dirichlet BCs. Using the Kratos flag that tells whether or not a certain degree of freedom is fixed, one can add the nodal basis vector to the elemental matrix, or add a vector of zeros, if the degree of freedom is fixed. This ensures to obtain a non-singular matrix $\boldsymbol{A}_{ROM}$.

ProjectToFineBasis()
This functions takes care of returning the nodal value of the unknown $\boldsymbol{T}$ on each step, starting from the reduced variable $\boldsymbol{x}_{ROM}$.

$$\boldsymbol{T} = \boldsymbol{\phi} \boldsymbol{x}_{ROM} \tag{3.5}$$

BuildAndSolve()
The main function of the class. Figure 3.10 shows the flowchart of it. Notice that the system solved is the reduced system, but at each step, the solution is projected back to the fine scale.

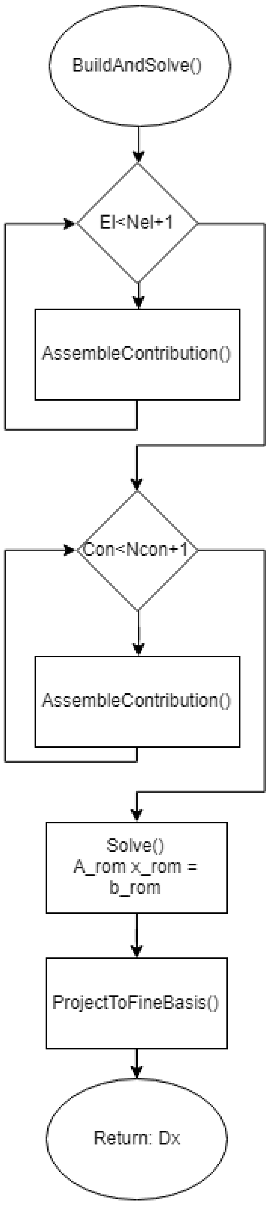


Figure 3.10: Flowchart of the BuildAndSolve() function in the ROM builder and solver

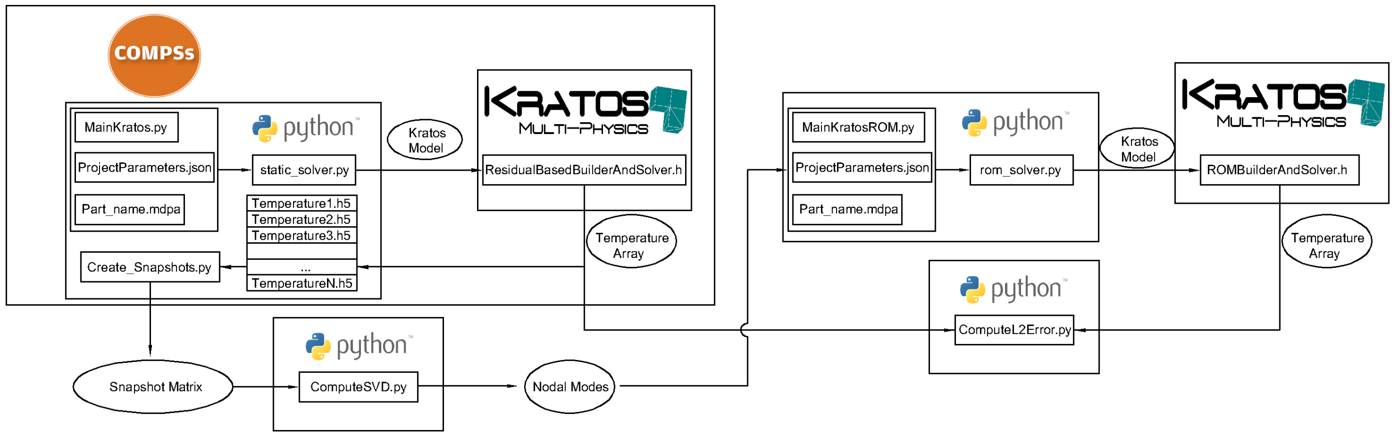


Figure 3.11: The complete workflow to train and run the ROM in Kratos

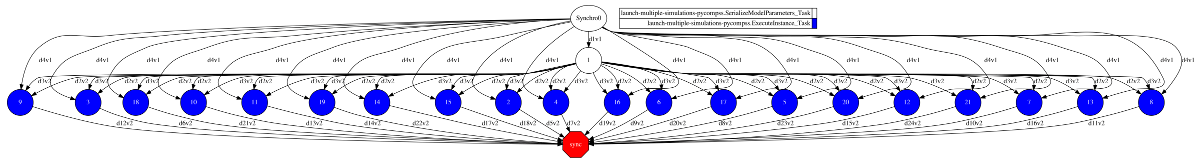

Synchro0
launch-multiple-simulations-pycompss.SerializeModelParameters_Task
launch-multiple-simulations-pycompss.ExecuteInstance_Task
sync

Figure 3.12: Sketch of the parallelism of the execution by COMPSs. The blue circles are the individual simulations, while the red hexagon is the synchronization point

Chapter 4
Results

This section presents the results obtained when running test examples on the ROM solver. There are two sections in this chapter dedicated to linear and nonlinear problems.

Linear and nonlinear problems are clearly different in the way to solve the equations, either the linear system arising from the discretization of the model is to be solved once, or iterations are needed until a convergence criterion is fulfilled. On top of that, in the ROM context, a much larger set of data for training is needed in the case of nonlinear problems, as compared to that needed for linear problems. As discussed in section 3, in order to perform the training of the nonlinear problems, the program COMPSs has been used to obtain the snapshot matrix. See figure 3.11.

The models presented are two 3D linear models, as well as a nonlinear 2D model. The absence of a nonlinear 3D model is due to lack of computer power to perform the training, a cluster is required for this task. The 3D nonlinear model is a pending task as is mentioned in the section 6.

4.1 Linear Problems

Linear problems are simple to treat in any context. For ROM, it can be said that the training cases needed are relatively few. Moreover, the exact solution is to be retrieved when using the right amount of modes.

The importance of starting the study of the ROM with linear models, is to precisely being able to observe such properties; before moving to larger and more complex applications.

The geometries studied are a cubic domain with Dirichlet boundary conditions on all its faces, and a simplified radiator that incorporates Neumann boundary conditions. The comparison is made of the full-scale model against the ROM model.

4.1.1 Cube with Dirichlet BCs

The first example is a cubic domain $\Omega = [0,1]^3$ with Dirichlet boundary conditions on all the faces $\partial\Omega = \Gamma_D$. The equation solved is:

$$\begin{cases} \nabla \cdot (\kappa \nabla T) = 0 & \text{in } \Omega, \\ T = T_D(t) & \text{on } \Gamma_D, \end{cases} \tag{4.1}$$

Where the function to impose the Dirichlet boundary condition is:

$$T_D(t) = x^2 + y^2 + z^2 - 2t \tag{4.2}$$

Clearly, t is not time, since the problem being solved is a Poisson equation. However, giving this information to Kratos in the ProjectParameters.json, and computing a given amount of "time steps" allows to obtain several simulations and arrange the results (via the create_snapshot_hdf5_process) into the snapshot matrix.

The information of the Kratos model is shown in table 4.1.

Cube with Dirichlet BCs		
	Number	Type
Nodes	4020	-
Elements	18144	LaplacianElement3D4N

Table 4.1: Characteristics of the finite element model: Cube

In order to train the ROM for this simulation, 10 "time steps" are computed. In reality, although one is dealing with a 3D model, with a quadratic distribution of temperatures, when taking the singular value decomposition of the snapshot matrix, it is expected to see that there is only one driving parameter, that is, one dominant mode, and the rest of the modes should be small in comparison.

After running the 10 simulations, obtaining the snapshot matrix, and taking the SVD; the resulting singular values are:

Matrix of singular values SIGMA:

$$\textbf{diag}(1952.5\ ,\ 54.3\ ,\ 2.8e^{-6}\ ,\ 1.43e^{-11}\ ,\ 2.3e^{-12}\ ,\ 1.1e^{-12}\ ,\ 3.4e^{-13}\ ,\ 5.5e^{-14}\ ,\ 4.0e^{-14}\ ,\ 0.0)$$

Figure 4.1 shows the plot of the singular values σ_i, of the diagonal matrix $\mathbf{\Sigma}$.

Following the Frobenius norm (Eq. 2.23) with a tolerance of $1e^{-6}$, two modes are to be taken.

Afterwards, a simulation was run in Kratos using the same parameters as for the full-order model, but using the ROM builder and solver described in section 3.5.3 with different amounts of modes. Figure 4.2 shows a comparison of the simulations of the full-order model, and ROM using 1 mode.

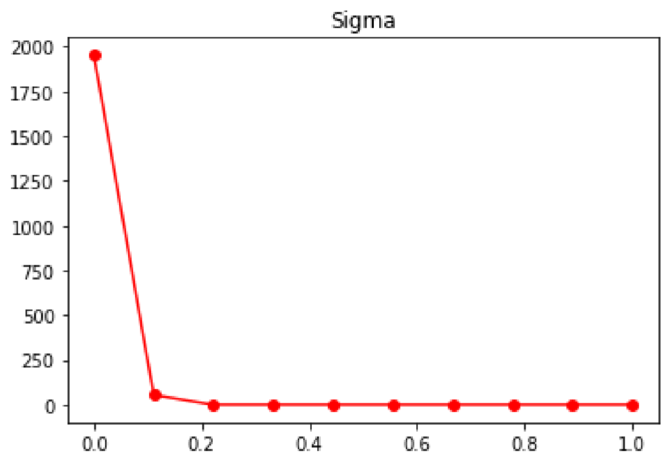
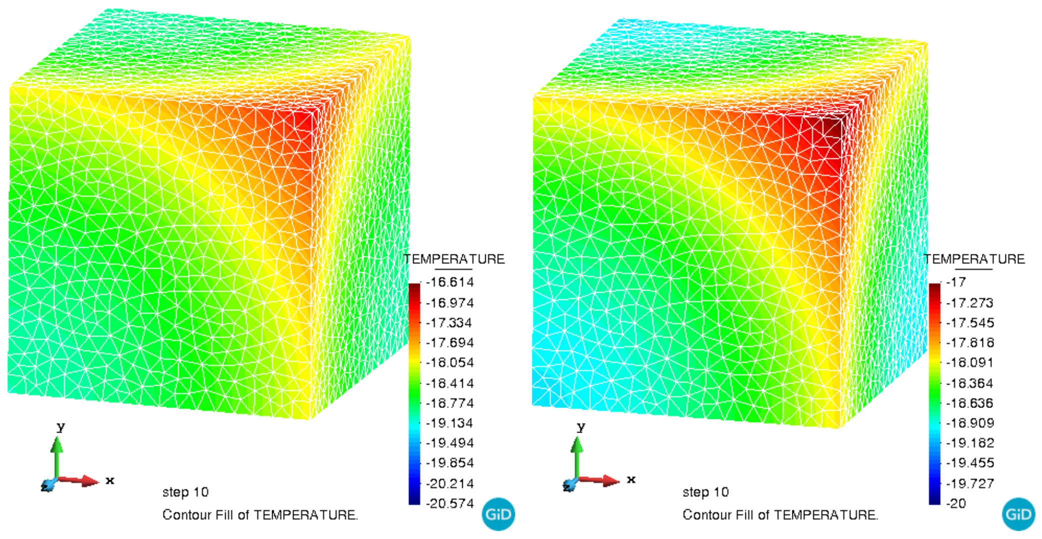


Figure 4.1: Singular values for Cube model



(a) ROM, 1 mode (b) Full-Order Model

Figure 4.2: Comparison ROM vs full-order Model

The error is measured using the L2 norm:

$$||T_F - T_{ROM}||_\Omega$$

Which is:

$$\sqrt{\frac{\int_\Omega (T_F(x,y) - T_{ROM}(x,y))^2 \, d\Omega}{\int_\Omega d\Omega}}$$

Which in the discrete case, can be expressed as:

$$\sqrt{\frac{\sum A_i \left(T_{Fi} - T_{ROMi}\right)^2}{\sum A_i}} \tag{4.3}$$

Where T_F is the nodal temperature in the full-order model, T_{ROM} is the nodal temperature in the ROM model, and A_i is the nodal area of influence.

Table 4.2 shows the L2 error of the simulation by employing different amounts of modes.

L2 Error	
Number of Modes	Error
1	0.240
2	$1.645e^{-08}$
5	$6.318e^{-08}$
10	$6.320e^{-08}$

Table 4.2: Error of the ROM with respect to the full-order model in L2 taking different amounts of modes

4.1.2 Simplified Radiator with Dirichlet and Neumann BCs

The second linear example is a 3D geometry that reassembles a radiator. The values to be changed in this geometry are the heat flux applied on three of the faces, while the temperature is prescribed fixed on four faces. The rest of the faces of the geometry have a zero prescribed normal flux. That is:

$$\begin{cases} \nabla \cdot (\kappa \nabla T) = 0 & \text{in } \Omega, \\ T = T_D & \text{on } \Gamma_D, \\ \boldsymbol{q} \cdot \boldsymbol{n} = q_{1,2,3} & \text{on } \Gamma_{N1,2,3}, \\ \boldsymbol{q} \cdot \boldsymbol{n} = 0 & \text{on } \partial\Omega \setminus (\Gamma_D \cup \Gamma_{N1,2,3}), \end{cases} \tag{4.4}$$

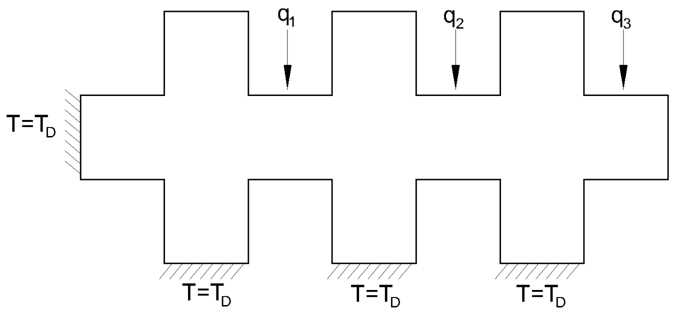


Figure 4.3: Radiator sketch

The information of the Kratos model is shown in table 4.3.

Radiator with Dirichlet and Neumann BCs		
	Number	Type
Nodes	3094	-
Elements	14121	LaplacianElement3D4N
Conditions	518	ThermalFace3D3N

Table 4.3: Characteristics of the finite element model: Radiator

Eight simulations were run in the full-order model using different boundary conditions. The imposed temperature was kept fixed to a given value, while the values for the imposed heat flux q_1, q_2, and q_3 were varied with the combinations shown in table 4.4.

From these simulations, the following matrix of singular values was obtained:

Matrix of singular values SIGMA:

$$\mathbf{diag}(15,201.40 \ , 688.77 \ , 216.96 \ , 90.00 \ , 2.74e^{-04} \ , 4.10e^{-05} \ , 1.95e^{-05} \ , 0.00)$$

Figure 4.4 shows the plot of the singular values σ_i, of the diagonal matrix $\mathbf{\Sigma}$.

The Frobenius norm (Eq. 2.23) with a tolerance of $1e^{-6}$ indicates that 4 modes are to be used. Using the recommended amount of modes, one can run the test cases indicated in Table 4.5.

Training Cases			
Case	q_1	q_2	q_3
1	0	0	0
2	q_1	0	0
3	q_1	q_2	0
4	q_1	q_2	q_3
5	q_1	0	q_3
6	0	q_2	0
7	0	0	q_3
8	0	q_2	q_3

Table 4.4: Training cases for the linear problem of the Radiator with Dirichlet and Neumann BCs.

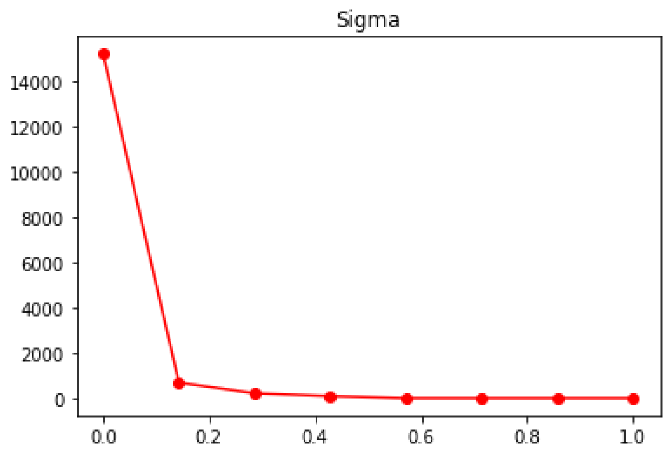


Figure 4.4: Singular values Radiator model

Test Cases			
Case	q_1	q_2	q_3
1	$q_1/2$	$q_2/2$	$q_3/2$
2	$q_1/5$	$4q_2/5$	q_3
3	$2q_1$	$2q_2$	$2q_3$
4	$-q_1$	$-q_2$	$-q_3$

Table 4.5: Cases to test the example Radiator

The error committed using the recommended amount of modes (4) was calculated using Equation 4.3, for each one of the test cases shown in Table 4.5.

L2 Error	
Case	Error
1	$2.470e^{-06}$
2	$4.245e^{-06}$
3	$5.435e^{-06}$
4	$6.377e^{-06}$

Table 4.6: Error of the ROM with respect to the Full-Order model in L2 for the different test cases in Table 4.5.

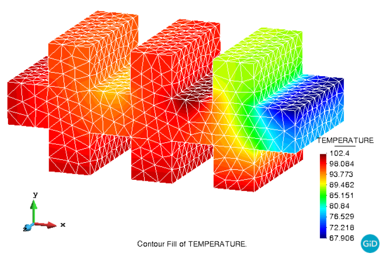
Speed of computations

A set of simulations were run for the same problem, using three different levels of mesh refinement. The data of the meshes can be seen in Table 4.7.

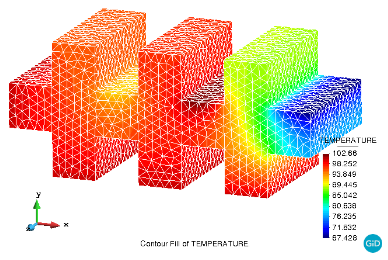
Cube with Dirichlet BCs			
	Mesh 1	Mesh 2	Mesh 3
Nodes	3094	9819	22188
Elements	14121	48618	114614
Conditions	518	1190	2030

Table 4.7: Data of the refinement of the Radiator model

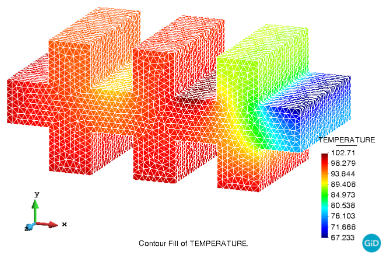
For reference, the meshes are shown in Figure 4.5.



(a)



(b)



(c)

Figure 4.5: Different meshes for Radiator model

The time that each of the different sections of the simulation takes was recorded. Some data of the computer in which the simulations were run are: Kratos Multiphysics version 7.0 in Debug mode, running on Windows on an 8GB RAM Laptop with processor AMD A10 at 1.8GHz. The time obtained for the simulations is shown in Table 4.8.

	Cube with Dirichlet BCs		
	SystemMatrixResize	SystemConstruction	Solve
Mesh 1			
ROM	0.0001 s	1.5 s	0.00025 s
Full-order	3.5 s	5.5 s	0.2 s
Mesh 2			
ROM	0.0004 s	5.5 s	0.00025 s
Full-order	12.4 s	17.6 s	0.9 s
Mesh 3			
ROM	0.0005 s	13.0 s	0.0004 s
Full-order	28.0 s	42.0 s	1.3 s

Table 4.8: Time comparison. ROM vs full-order model for the Radiator model

4.2 Non-Linear Problems

As mentioned in section 3.4.2, the Solution Strategy class is the one that calls the Builder and Solver. The solution strategy used for nonlinear problems is the Newton-Raphson strategy with suitable convergence criterion. Both, the full-scale model and the ROM use the same Solution Strategy, and both do converge. However, unlike for linear problems, the results are expected to differ when using the full-scale or the ROM solver [9]. The difference can be observed in Table 4.12, and a further discussion on this can be found on Section 5.1.

On top of that, the necessary training cases have been run on a virtual Linux machine in the writer´s personal laptop. In order to further investigate the performance of the code, and to simulate more complicated geometries in 3D, more computer power is needed.

4.2.1 Square with Dirichlet, Neumann and Radiation BCs

The nonlinear example considered is solved on a square domain $\Omega = [0,1]^2$ whose boundary conditions are imposed temperature on one face, imposed flux on two faces, and a Stefan-Boltzmann radiation condition on the last face. The problem is:

$$\begin{cases} \nabla \cdot (\kappa \nabla T) = 0 & \text{in } \Omega, \\ T = T_D & \text{on } \Gamma_D, \\ \boldsymbol{q} \cdot \boldsymbol{n} = q_1 & \text{on } \Gamma_{N1}, \\ \boldsymbol{q} \cdot \boldsymbol{n} = q_2 & \text{on } \Gamma_{N2}, \\ \boldsymbol{q} \cdot \boldsymbol{n} = q^{SB}(T) = \sigma\epsilon(T^4 - T_\infty^4) & \text{on } \Gamma_R, \end{cases} \tag{4.5}$$

Where ϵ = emissivity, σ = SB constant, and T_∞ = ambient temperature.

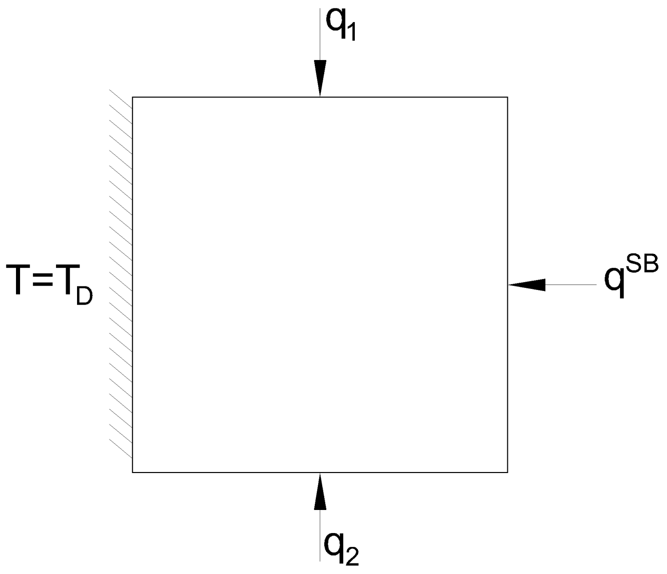


Figure 4.6: Sketch of square with radiation

The information of the Kratos model is shown in table 4.9.

Square with Dirichlet, Neumann and Radiation BCs		
	Number	Type
Nodes	251	-
Elements	444	LaplacianElement2D3N
Conditions	56	ThermalFace2D2N

Table 4.9: Characteristics of the finite element model: Square radiation

In order to train the ROM, many case are needed. In particular, the training parameters that were used for the problem at hand are shown next:

$$\begin{cases} q_1 & = [-2000.0, -1500.0, -1000.0, 0.0, 1000.0, 1500.0, 2000.0] \\ q_2 & = [-2000.0, -1500.0, -1000.0, 0.0, 1000.0, 1500.0, 2000.0] \\ \text{Ambient Temperature} & = [5.0, 10.0, 15.0, 20.0] \\ \text{Imposed Temperature} & = [50.0, 75.0, 100.0, 125.0] \end{cases}$$

As can be seen, all possible combination of conditions produce 784 training cases. Therefore, it was not viable to run them manually. The COMPSs tool explained in section 3.5.1 was used to set and run the cases. The snapshot matrix obtained is shown in Figure 4.7. Recall that the columns are the nodal temperature values.

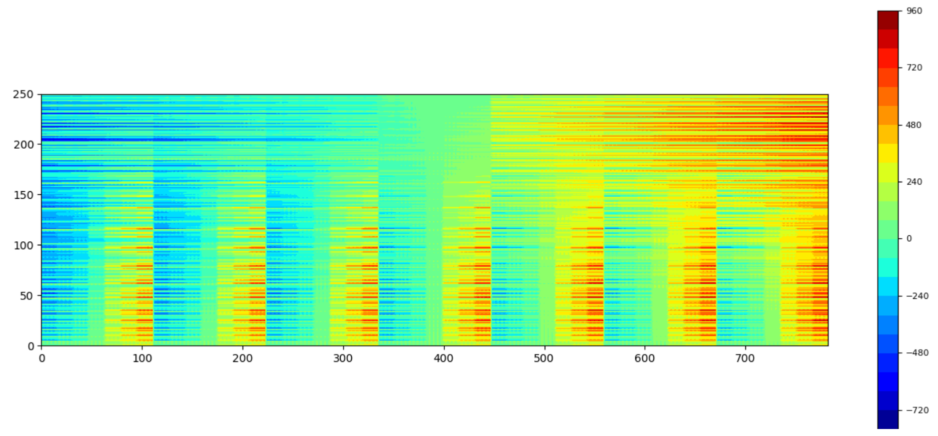


Figure 4.7: Snapshots matrix Coarse Square model

The SVD of this matrix was taken. Figure 4.8 shows the plot of the singular values σ_i, of the diagonal matrix $\mathbf{\Sigma}$.

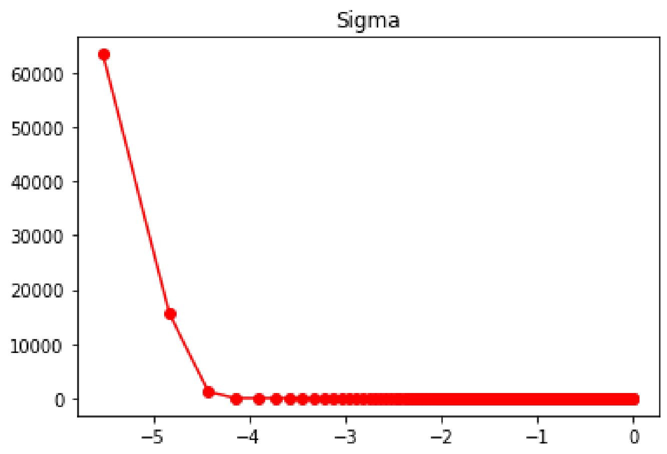


Figure 4.8: Singular values Coarse Square model

The Frobenius norm (Eq. 2.23) indicates that 6 modes are to be used.

The same problem was solved using a refined model. Table 4.10 shows the data of the refined Kratos model.

Refined Square with Dirichlet, Neumann and Radiation BCs		
	Number	Type
Nodes	1316	-
Elements	2498	LaplacianElement2D3N
Conditions	132	ThermalFace2D2N

Table 4.10: Characteristics of the refined finite element model: Refined Square Radiation

The models are shown in figure 4.9.

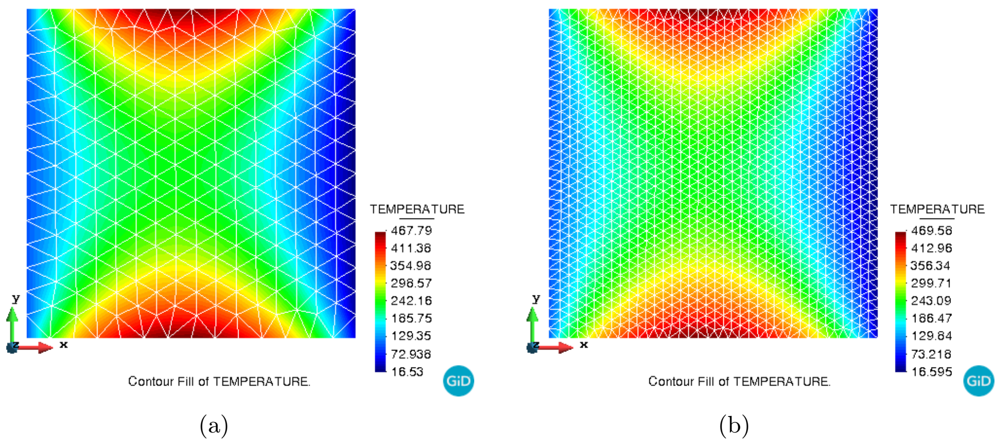


Figure 4.9: Different meshes for the square with radiation

Consequently, the snapshot matrix of the refined model for the 784 training cases was obtained and is shown in Figure 4.10.

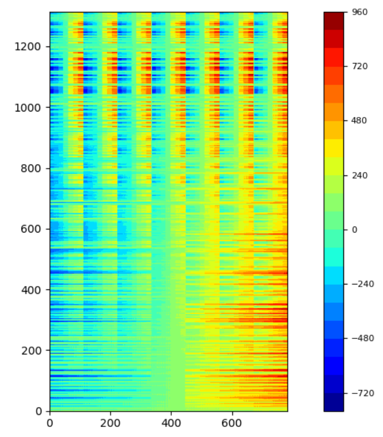


Figure 4.10: Snapshots matrix Refined Square model

The SVD of this matrix was taken. Figure 4.11 shows the plot of the singular values σ_i, of the diagonal matrix $\mathbf{\Sigma}$.

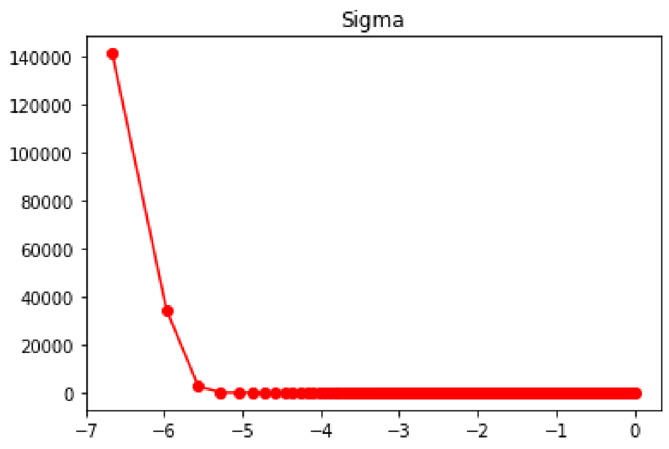


Figure 4.11: Singular values Refined Square model

The Frobenius norm (Eq. 2.23) with a tolerance of $1e^{-6}$ indicates that 4 modes are to be used.

As test cases, one can use values that have not been trained, but lay in between trained values (as already done for the simplified radiator example). For this case, the training cases are:

Test Cases				
Case	$\boldsymbol{q_1}$	$\boldsymbol{q_2}$	**Ambient Temperature**	**Imposed Temperature**
1	500	500	12	60
2	-1300	1300	7	110
3	700	-700	7	110
4	0	0	7	110
5	700	700	7	110

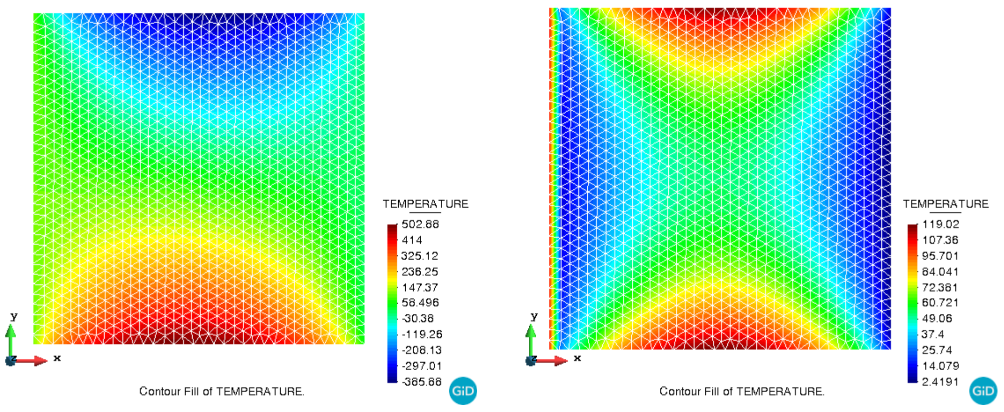
Table 4.11: Cases to test the example Square with Radiation

The L2 error is calculated using Equation 4.3 and is shown in table 4.12.

L2 Error	
Case 1	
Fine	0.461
Coarse	2.675
Case 2	
Fine	$1.132e^{-4}$
Coarse	$9.264e^{-6}$
Case 3	
Fine	$1.141e^{-4}$
Coarse	$7.137e^{-6}$
Case 4	
Fine	$0.128e^{-4}$
Coarse	$5.26e^{-6}$
Case 5	
Fine	0.645
Coarse	3.7451

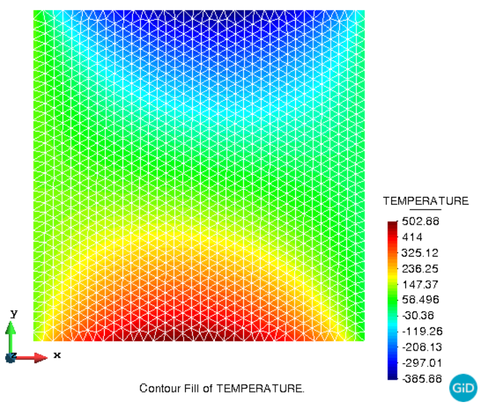
Table 4.12: Error for different cases studied. ROM vs full-order model

In order to study the error in the ROM when taking a different amount of modes, the refined model was used. The case studied is the **Case 2** in table 4.11. The number of modes is varied from 1 to 8, and the relative L2 using 4.3 error is observed. Further discussion is found in section 5.1.

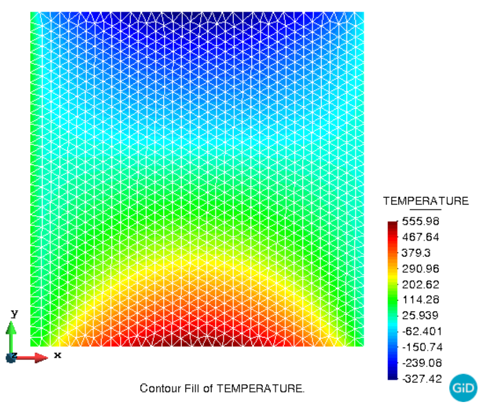


(a) full-order model (b) ROM 1 mode

Figure 4.12: ROM 1 mode. Error 178.66

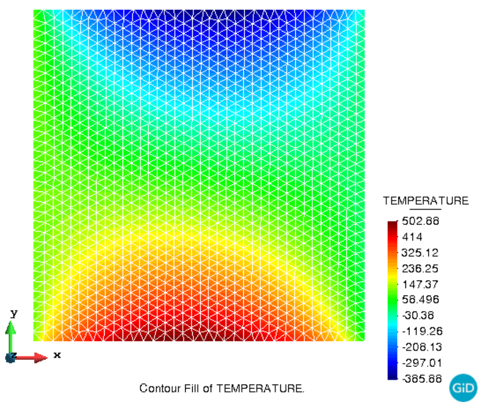


(a) full-order model

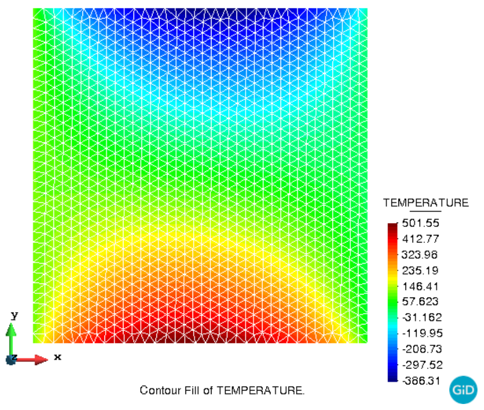


(b) ROM 2 modes

Figure 4.13: ROM 2 modes. Error 36.873

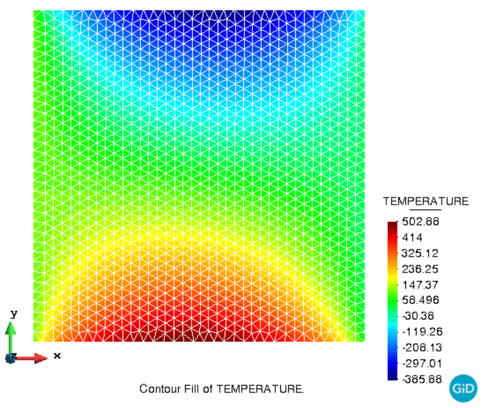


(a) full-order model

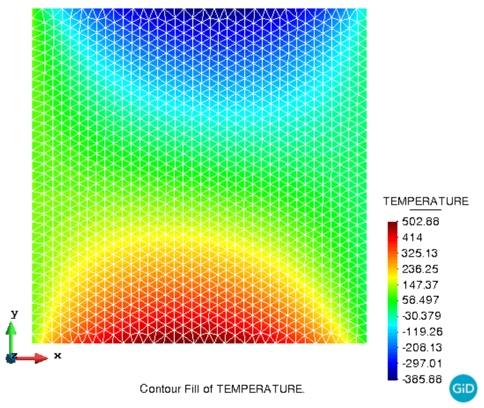


(b) ROM 3 modes

Figure 4.14: ROM 3 modes. Error 2.964

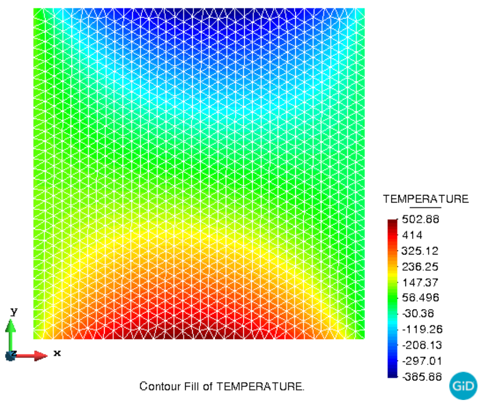


(a) full-order model

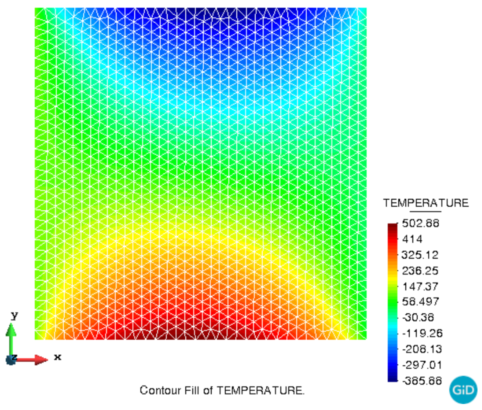


(b) ROM 4 modes

Figure 4.15: ROM 4 modes. Error $1.132e^{-4}$

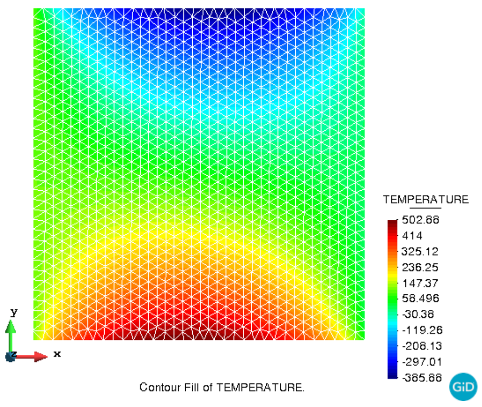


(a) full-order model

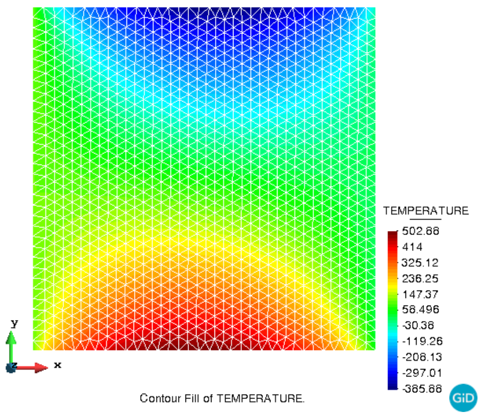


(b) ROM 5 modes

Figure 4.16: ROM 5 modes. Error $6.582e^{-6}$

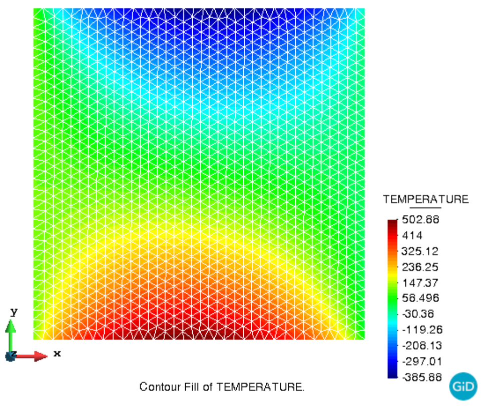


(a) full-order model

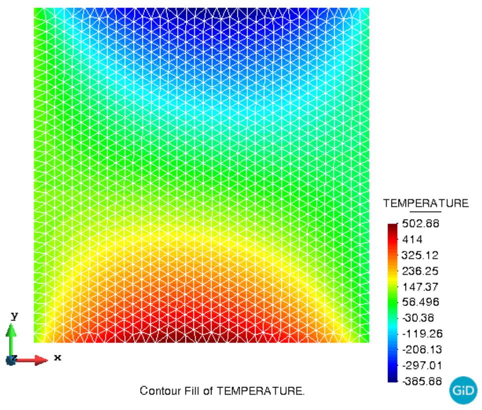


(b) ROM 6 modes

Figure 4.17: ROM 6 modes. Error $8.676e^{-6}$

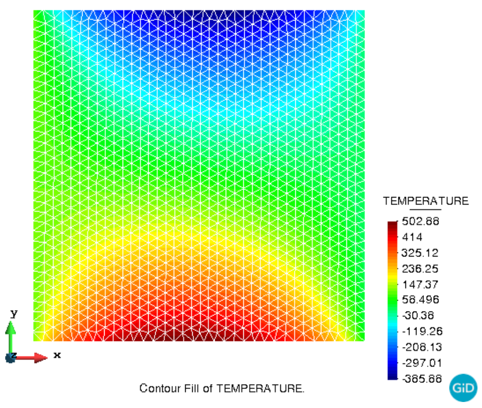


(a) full-order model

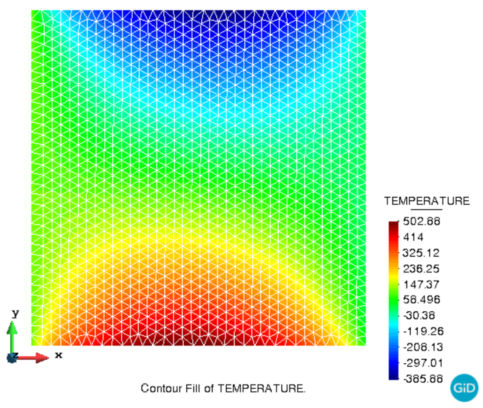


(b) ROM 7 modes

Figure 4.18: ROM 7 modes. Error $3.610e^{-5}$

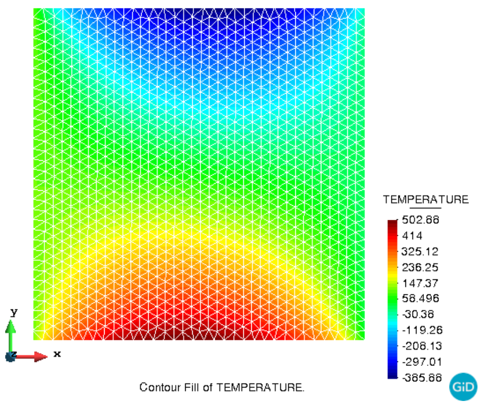


(a) full-order model

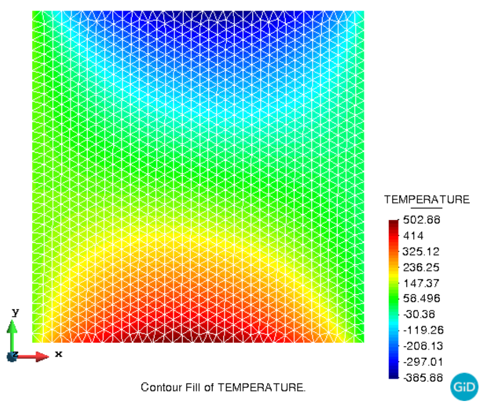


(b) ROM 8 modes

Figure 4.19: ROM 8 modes. Error $4.150e^{-7}$

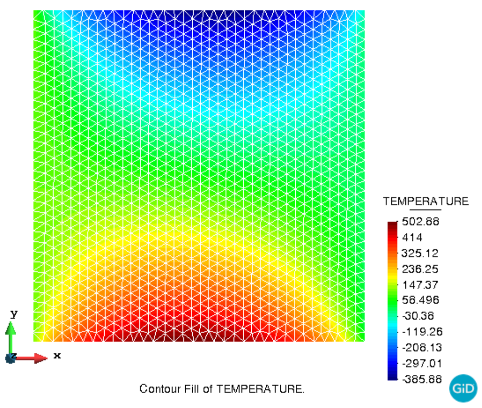


(a) full-order model

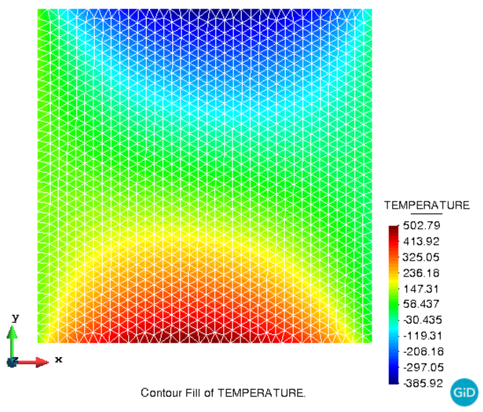


(b) ROM 16 modes

Figure 4.20: ROM 16 modes. Error $1.76e^{-5}$

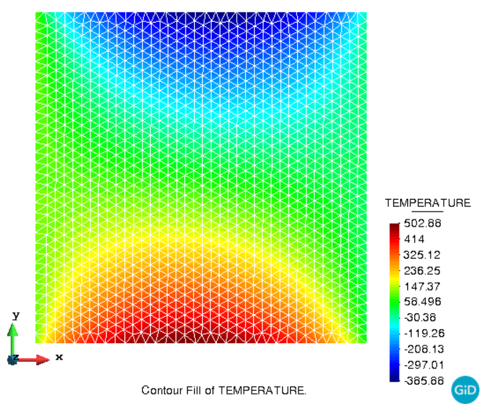


(a) full-order model

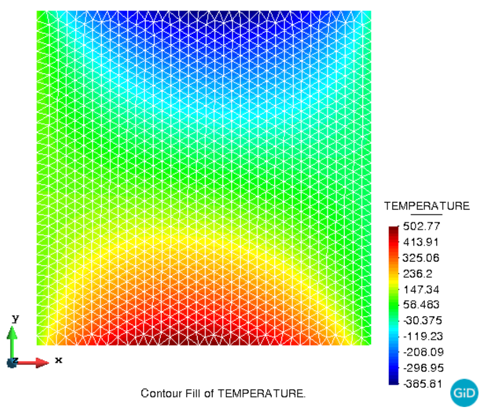


(b) ROM 24 modes

Figure 4.21: ROM 24 modes. Error 0.056

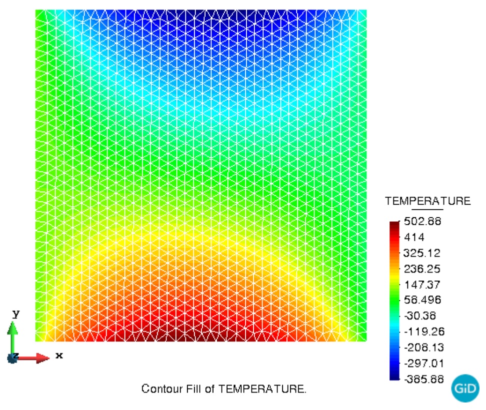


(a) full-order model

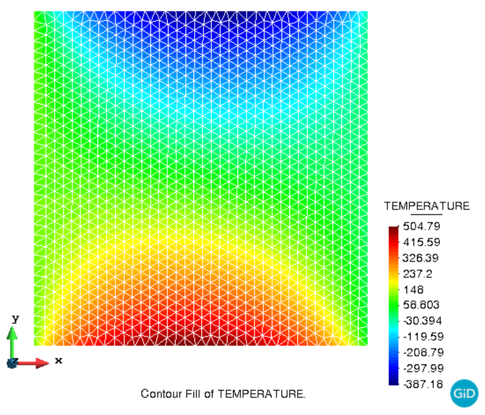


(b) ROM 30 modes

Figure 4.22: ROM 30 modes. Error 0.320

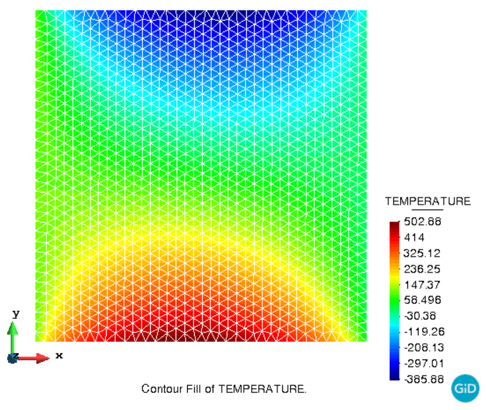


(a) full-order model

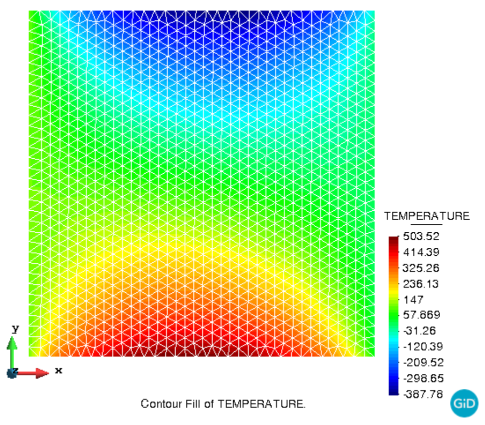


(b) ROM 50 modes

Figure 4.23: ROM 50 modes. Error 0.927



(a) full-order model



(b) ROM 100 modes

Figure 4.24: ROM 100 modes. Error 3.626

Chapter 5

Discussion

This chapter analyses the results obtained in section 4. First, the difference or the error obtained by comparing the full-order model against the ROM using 4.3, is covered. The time improvement with respect to the full-order model, is also treated.

5.1 Accuracy of the ROM

For linear problems, the results using the ROM solver implemented in Kratos were expected to match those obtained using full-order model. Tables 4.2 and 4.6, do confirm that, when using the amounts of modes dictated by the Frobenius norm (Eq. 2.23), the ROM solver is capable of reproducing the solution of the full-order model.

The case is not the same for nonlinear problems solved using ROM, for which the solution, even using the recommended amounts of modes, can differ (within a given tolerance) from the full-order model. The difference can be observed in Table 4.12.

On the other hand, by changing the amount of modes used, one can observe that the accuracy of the solution improves up to a certain point. After this point, the inclusion of more modes is polluting the solution, as was observed in the case of the linear cube (Table 4.2). In this example, 1 mode is producing a not-so-accurate solution. By incorporating 1 more mode, the solution is improved, but after including more, the solutions worsens. Since the cube example is linear on temperature; this behavior is not so evident. In the nonlinear example studied, this same characteristic is observed. Figures 4.12 to 4.24 show the comparison of the temperature fields of the ROM and the full-order model using from 1 to 100 modes. Figure 5.1 evidently shows the mentioned characteristic.

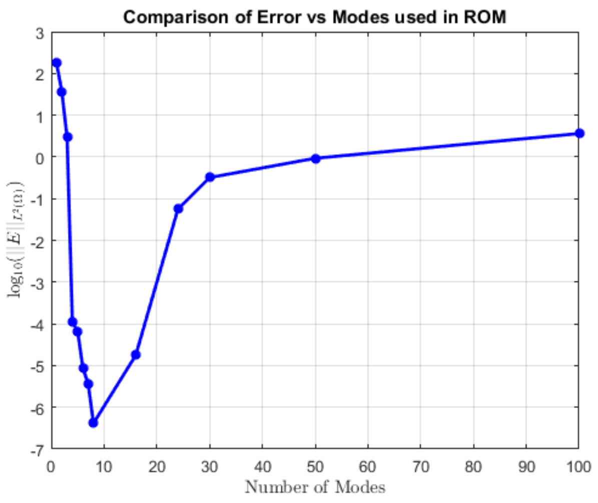


Figure 5.1: L2 error calculated using 4.3 for the ROM using different amounts of modes for example 4.2.1

5.2 Improvement in Time

While Table 4.8 does demonstrate the significant improvement in time of specific operations inside Kratos when comparing the full-order model to the ROM, the whole story is not that shocking.

A Kratos simulation is composed by many other processes for which time is not varying that substantially. In general, for the problems considered, the total time that the ROM simulation required to be completed is about half of the time of the full-order model.

It was observed that the gaining in time was not amazing. It could be said that one of the limitations of POD methods is that they reduce the dimension of the problem, but not necessarily its complexity [10]. The largest reduction in time is obtained when using a technique called hyper-reduction. In order to implement a ROM application in Kratos, this second reduction stage is necessary.

Chapter 6
Conclusion

The main objective of the present work was to carry out the implementation in the software Kratos Multiphysics of a Reduced Order Model application based on the Proper Orthogonal Decomposition technique. This task proved to be challenging, specially due to the fact that the implementation was done from scratch, creating all the infrastructure necessary. At the end of the time allotted, this main objective was not completely fulfilled (The final sketch of the implemented infrastructure can be seen in figure 3.11). However, firm steps have been taken in order to add the mentioned application to Kratos.

In this work, chapter 2 presented the main theory related to the Singular Value Decomposition. Moreover, a 1D example in Python, which is included in the appendix, served to highlight the properties of the Proper Orthogonal Decomposition. Chapter 3 presented the tools that have been employed, that is GiD and COMPSs. This chapter also introduced all of the necessary steps to have a ROM solver working in Kratos.

Chapter 4 served to check the correctness and efficiency of the implementation performed. Simple geometries were studied for this purpose. Chapter 5 presented a discussion of the results obtained.

Based on the results and discussion presented, it can be concluded that the created infrastructure is capable of training, and running a correct ROM simulation. Moreover, the implemented Reduced Order Model solver (section 3.5) is capable of obtaining fast and accurate solutions for linear and nonlinear problems for static thermal applications.

6.1 Future Work

There are a number of tasks required to take the implemented infrastructure presented in this work, from its current state, to a robust and fully functional ROM application to be incorporated in future releases of Kratos Multiphysics.

6.1.1 Training the ROM

As it was observed in the case of linear problems (Table 4.4), the amount of training cases necessary to capture the dominant modes in linear problems is small. In the case of nonlinear problems, orders of magnitude more cases and computational power is needed.

The program COMPSs (see section 3.5.1) was used to train the nonlinear 2D model presented (section 4.2.1). However, the hardware limitations did not allow the training of 3D nonlinear models.

Therefore, in order to implement the ROM application in Kratos, it is imperative to gain access to more computational power. This will allow also to study more complex geometries, which are of interest for industrial applications.

6.1.2 Efficient SVD

In this thesis, the SVD has been treated as a black-box, the implementation used is that of numpy. In the beginning of the work done for this project, this approach was sufficient. But it is important to highlight that the SVD is an expensive operation and that in order to obtain an efficient ROM application, other alternatives must be considered.

The classical algorithm performs the SVD in two steps. The first one consists on the reduction of the original matrix into a bidiagonal one,and the second step is to compute the SVD of the bidiagonal matrix, this is done iteratively. The overall cost is of $O(mn^2)$[11].

An important step to deal with is the implementation of a more efficient way to calculate the SVD for large matrices. A very feasible option is to implement the Ramdomized SVD.

6.1.3 Hyper-reduction

A very important step is the implementation of the hyper-reduced model (HROM), as it is known that the largest saving in computational burden when running the model is obtained from this step. Implementing the HROM is crucial because, unlike the case of the randomized SVD mentioned in last section, whose greater advantage is perceived during the training stage, the HROM is immediately translated into orders of magnitude less time for calculating a given simulation[12].

Bibliography

[1] Zienkiewicz OC, Taylor RL, Nithiarasu P, Zhu J. The finite element method. vol. 3. McGraw-hill London; 1977.

[2] Hernández Ortega JA, Oliver Olivella X, Huespe AE, Caicedo MA. High-performance model reduction procedures in multiscale simulations. Centre Internacional de Mètodes Numèrics en Enginyeria (CIMNE); 2012.

[3] Reduced Order Modeling Course;. Accessed: 2019-06-06. `https://faculty.washington.edu/kutz/rom/rom.html`.

[4] Rossi R. Light weight structures: structural analysis and coupling issues. Doktorarbeit, University of Bologna. 2005;.

[5] Falkiewicz NJ, S Cesnik CE. Proper orthogonal decomposition for reduced-order thermal solution in hypersonic aerothermoelastic simulations. AIAA journal. 2011;49(5):994–1009.

[6] Polansky J, Wang M, Faraj Y. Proper Orthogonal Decomposition as a technique for identifying multiphase flow regime based on Electrical Impedance Tomography. In: 7th International Symposium on Process Tomography. Leeds; 2015. .

[7] Dadvand P. A framework for developing finite element codes for multi-disciplinary applications. Barcelona, Spain: Universitat Politècnica de Catalunya; 2007.

[8] P Dadvand RR, Oñate E. An object-oriented environment for developing finite element codes for multi-disciplinary applications. Archives of computational methods in engineering. 2010;17(3):253–297.

[9] Pinnau R. Model reduction via proper orthogonal decomposition. In: Model order reduction: theory, research aspects and applications. Springer; 2008. p. 95–109.

[10] Chaturantabut S, Sorensen DC. Nonlinear model reduction via discrete empirical interpolation. SIAM Journal on Scientific Computing. 2010;32(5):2737–2764.

[11] Trefethen LN, Bau III D. Numerical linear algebra. vol. 50. Siam; 1997.

[12] Hernandez JA, Caicedo MA, Ferrer A. Dimensional hyper-reduction of nonlinear finite element models via empirical cubature. Computer methods in applied mechanics and engineering. 2017;313:687–722.

Appendix A

Python Code. Modal Analysis 1D Bar

```
#Jose Raul Bravo Martinez
#MSc Computational Mechanics

##########################################################################
# This code calculates the vibration of a 1D bar using
# the full model and modal analysis

import numpy as np
from scipy.linalg import eigh
from scipy import linalg
from matplotlib import pyplot as plt
from numpy import zeros,dot


#Function to assemble the global Mass and Stiffness Matrices. Also obtains
    the
#Eigenvalues and Eigenvectors for Modal Analysis
def bar(num_elems):
  restrained_dofs = [0,]

  # element mass and stiffness matrices for a bar
  m = np.array([[2,1],[1,2]]) / (6. * num_elems)
  k = np.array([[1,-1],[-1,1]]) * float(num_elems)

  # construct global mass and stiffness matrices
  M = np.zeros((num_elems+1,num_elems+1))
  K = np.zeros((num_elems+1,num_elems+1))

  # assembly of elements
  for i in range(num_elems):
    M_temp = np.zeros((num_elems+1,num_elems+1))
    K_temp = np.zeros((num_elems+1,num_elems+1))
    M_temp[i:i+2,i:i+2] = m
    K_temp[i:i+2,i:i+2] = k
    M += M_temp
    K += K_temp

  # remove the fixed degrees of freedom
  for dof in restrained_dofs:
    for i in [0,1]:
      M = np.delete(M, dof, axis=i)
      K = np.delete(K, dof, axis=i)

  # eigenvalue problem
  evals, evecs = eigh(K,M)
  frequencies = np.sqrt(evals)
  return M, K, frequencies, evecs


#Function to calculate the newmark coefficients for time intergration
def Newmark_coefficients(dt):
    alpha=0.25
    beta=0.5

    a0=1/(alpha*(dt**2))
    a1=beta/(alpha*dt)
    a2=1/(alpha*dt)
    a3=(1/(2*alpha))-1
    a4=(beta/alpha)-1
    a5=(dt/2)*((beta/alpha)-2)
    a6=dt*(1-beta)
    a7=beta*dt

    return a0,a1,a2,a3,a4,a5,a6,a7

#Setting conditions
tt=10
dt=0.01    #Time step
Ntp=int(tt/dt) #Number of time steps
#Ntp=1000
NT=20 #Number of elements
print ('Number of elements:', NT )
M, K, frequencies, evecs = bar(NT)
C=zeros((M.shape[0], M.shape[1]))
F= zeros((M.shape[0], 1))
F[NT-1]=.2 #Force
a0,a1,a2,a3,a4,a5,a6,a7= Newmark_coefficients(dt)

######################################
#Full Model
######################################

KH=K+a0*M+a1*C

U=np.array(zeros((M.shape[0], 1)))
Ud=np.array(zeros((M.shape[0], 1)))
Udd=np.array(linalg.solve(M, F))

results = []

for j in range (0,Ntp):
    V1=(a1*U + a4*Ud +a5*Udd)
    V2=(a0*U + a2*Ud +a3*Udd)
    CV=dot(C,V1)
    MA=dot(M,V2)
    FH=F+MA+CV
    #Solve for displacements
    Un=linalg.solve(KH, FH)
    #Solve for acceleration
    Uddn=a0*(Un-U)-a2*Ud-a3*Udd
    #Solve for velocity
    Udn=Ud+a6*Udd+a7*Uddn

    #Update_Acc,Vel,Disp
    U=Un
    Ud=Udn
    Udd=Uddn
    results.append( (j, U[NT-1]) )

```

```

######################################
#Modal Analysis
######################################

#Setting conditions
Phi=evecs
Phi=Phi[:,0:1] #Select the number of modes to use
#Reduced Matrices
F_star=np.matmul(Phi.transpose(),F)
K_star=np.matmul(Phi.transpose(),(np.matmul(K, Phi)))
M_star=np.matmul(Phi.transpose(),(np.matmul(M, Phi)))
C=zeros((M_star.shape[0], M_star.shape[1]))

KH=K_star+a0*M_star+a1*C

U=np.array(zeros((M_star.shape[0], 1)))
Ud=np.array(zeros((M_star.shape[0], 1)))
Udd=np.array(linalg.solve(M_star, F_star))

results1 = []

for j in range (0,Ntp):
    V1=(a1*U + a4*Ud +a5*Udd)
    V2=(a0*U + a2*Ud +a3*Udd)
    CV=dot(C,V1)
    MA=dot(M_star,V2)
    FH=F_star+MA+CV
    #Solve for displacements
    Un=linalg.solve(KH, FH)
    #Solve for acceleration
    Uddn=a0*(Un-U)-a2*Ud-a3*Udd
    #Solve for velocity
    Udn=Ud+a6*Udd+a7*Uddn

    #Update_Acc,Vel,Disp
    U=Un
    Ud=Udn
    Udd=Uddn
    Converted=dot(Phi,U)
    results1.append( (j, Converted[NT-1]) )

# plot the results
Displacement  = np.array([x[1] for x in results])
Time_Steps = np.array([x[0] for x in results])
Time_Steps=dt*Time_Steps

plt.title('Displacement at End Node')
line_1, =plt.plot(Time_Steps, Displacement, 'rs', label='Full Model')
Displacement1  = np.array([x[1] for x in results1])
line_2, =plt.plot(Time_Steps, Displacement1, 'b', label='Modal Analysis')
plt.legend(handles=[line_1,line_2])
```

Appendix B

Python Code. ROM linear 1D Bar

```
#Jose Raul Bravo Martinez
#MSc Computational Mechanics

###########################################################################
# This code calculates the vibration of a 1D bar using
# the full model and proper orthogonal decomposition

import numpy as np
from scipy.linalg import eigh
from scipy import linalg
from matplotlib import pyplot as plt
from numpy import zeros,dot
from scipy.linalg import svd

#Function to assemble the global Mass and Stiffness Matrices. Also obtains
    the
#Eigenvalues and Eigenvectors for Modal Analysis
def bar(num_elems):
  restrained_dofs = [0,]

  # element mass and stiffness matrices for a bar
  m = np.array([[2,1],[1,2]]) / (6. * num_elems)
  k = np.array([[1,-1],[-1,1]]) * float(num_elems)

  # construct global mass and stiffness matrices
  M = np.zeros((num_elems+1,num_elems+1))
  K = np.zeros((num_elems+1,num_elems+1))

  # assembly of elements
  for i in range(num_elems):
    M_temp = np.zeros((num_elems+1,num_elems+1))
    K_temp = np.zeros((num_elems+1,num_elems+1))
    M_temp[i:i+2,i:i+2] = m
    K_temp[i:i+2,i:i+2] = k
    M += M_temp
    K += K_temp

  # remove the fixed degrees of freedom
  for dof in restrained_dofs:
    for i in [0,1]:
      M = np.delete(M, dof, axis=i)
      K = np.delete(K, dof, axis=i)

  # eigenvalue problem
  evals, evecs = eigh(K,M)
  frequencies = np.sqrt(evals)
  return M, K, frequencies, evecs


#Function to calculate the newmark coefficients for time intergration
def Newmark_coefficients(dt):
    alpha=0.25
    beta=0.5

    a0=1/(alpha*(dt**2))
    a1=beta/(alpha*dt)
    a2=1/(alpha*dt)
    a3=(1/(2*alpha))-1
    a4=(beta/alpha)-1
    a5=(dt/2)*((beta/alpha)-2)
    a6=dt*(1-beta)
    a7=beta*dt

    return a0,a1,a2,a3,a4,a5,a6,a7

#Setting conditions
tt=10
dt=0.01    #Time step
Ntp=int(tt/dt) #Number of time steps
#Ntp=1000
NT=20 #Number of elements
print ('Number of elements:', NT )
M, K, frequencies, evecs = bar(NT)
C=zeros((M.shape[0], M.shape[1]))
F= zeros((M.shape[0], 1))
F[NT-1]=.2 #Force
a0,a1,a2,a3,a4,a5,a6,a7= Newmark_coefficients(dt)

#####################################
#Full Model
#####################################

KH=K+a0*M+a1*C

results_SVD= zeros((M.shape[0], Ntp))
U=np.array(zeros((M.shape[0], 1)))
Ud=np.array(zeros((M.shape[0], 1)))
Udd=np.array(linalg.solve(M, F))

results = []

for j in range (0,Ntp):
    V1=(a1*U + a4*Ud +a5*Udd)
    V2=(a0*U + a2*Ud +a3*Udd)
    CV=dot(C,V1)
    MA=dot(M,V2)
    FH=F+MA+CV
    #Solve for displacements
    Un=linalg.solve(KH, FH)
    #Solve for acceleration
    Uddn=a0*(Un-U)-a2*Ud-a3*Udd
    #Solve for velocity
    Udn=Ud+a6*Udd+a7*Uddn

    #Update_Acc,Vel,Disp
    U=Un
    Ud=Udn
    Udd=Uddn
    results.append( (j, U[NT-1]) )
    results_SVD[:,j]=U.transpose()

Displacement   = np.array([x[1] for x in results])

####################################
# Proper Orthogonal Decomposition
####################################
# Taking the SVD
U, s, VT = svd(results_SVD)
Phi=U
Phi=Phi[:,0:1] #Select the number of modes to use
#Reduced Matrices
F_star=np.matmul(Phi.transpose(),F)
K_star=np.matmul(Phi.transpose(),(np.matmul(K, Phi)))
M_star=np.matmul(Phi.transpose(),(np.matmul(M, Phi)))
C=zeros((M_star.shape[0], M_star.shape[1]))

KH=K_star+a0*M_star+a1*C

U=np.array(zeros((M_star.shape[0], 1)))
Ud=np.array(zeros((M_star.shape[0], 1)))
Udd=np.array(linalg.solve(M_star, F_star))

results1 = []

for j in range (0,Ntp):
    V1=(a1*U + a4*Ud +a5*Udd)
    V2=(a0*U + a2*Ud +a3*Udd)
    CV=dot(C,V1)
    MA=dot(M_star,V2)
    FH=F_star+MA+CV
    #Solve for displacements
    Un=linalg.solve(KH, FH)
    #Solve for acceleration
    Uddn=a0*(Un-U)-a2*Ud-a3*Udd
    #Solve for velocity
    Udn=Ud+a6*Udd+a7*Uddn

    #Update_Acc,Vel,Disp
    U=Un
    Ud=Udn
    Udd=Uddn
    Converted=dot(Phi,U)
    results1.append( (j, Converted[NT-1]) )

# plot the results
Displacement   = np.array([x[1] for x in results])
Time_Steps = np.array([x[0] for x in results])
Time_Steps=dt*Time_Steps

plt.title('Displacement at End Node')
line_1, =plt.plot(Time_Steps, Displacement, 'rs', label='Full Model')
Displacement1   = np.array([x[1] for x in results1])
line_2, =plt.plot(Time_Steps, Displacement1, 'b', label='POD')
plt.legend(handles=[line_1,line_2])
```

Appendix C

Python Code. ROM Nonlinear 1D Bar

```
#Jose Raul Bravo Martinez
#MSc Computational Mechanics

####################################################################
# This code calculates the vibration of a 1D bar with a nonlinear
# response using the full model and proper orthogonal decomposition

import numpy as np
from scipy import linalg
from matplotlib import pyplot as plt
from numpy import zeros,dot
from scipy.linalg import svd

#Function to calculate the newmark coefficients for time intergration
def Newmark_coefficients(dt):
    alpha=0.25
    beta=0.5

    a0=1/(alpha*(dt**2))
    a1=beta/(alpha*dt)
    a2=1/(alpha*dt)
    a3=(1/(2*alpha))-1
    a4=(beta/alpha)-1
    a5=(dt/2)*((beta/alpha)-2)
    a6=dt*(1-beta)
    a7=beta*dt

    return a0,a1,a2,a3,a4,a5,a6,a7

#Setting conditions
NT=20#Number of elements
tt=10#total time
dt=0.01 #Time step size
Ntp=180 #Number of time steps

a0,a1,a2,a3,a4,a5,a6,a7= Newmark_coefficients(dt) #Getting Newmark
    Coefficients
restrained_dofs=[0,] #Restrained degreees of freedom
deltaX= np.zeros((NT+1,1))
tol=1e-6 #Setting tolerance


#####################################
#Full Model
#####################################

U=np.zeros((NT+1,1)) #predicted displacement 1
Un=U
```

```
Ud=np.zeros((NT+1,1))#initializing Velocity
Udn=Ud
Udd=np.zeros((NT+1,1)) #initializing acceleration
Uddn=Udd
F= np.zeros((len(U), 1)) #creating the force of zeros
F[NT]=0.2 #Applying a force at the end of the bar
K_static_el=(np.array([[2,0],[0,2]]) * float(NT))  #For nonlinear case f=u
    **2 -1
M_el= np.array([[2,1],[1,2]]) / (6. * NT )
results_U= zeros(((len(U)), Ntp))
results_Ud= zeros(((len(U)), Ntp))
results_Udd= zeros(((len(U)), Ntp))

#Time step loop
for CurrentTime in range (Ntp):
    print ("time step", CurrentTime)

    Ud=Udn
    Udd=Uddn
    Un=U+(Ud*dt)

    #Newmark for time step
    #Solve for acceleration
    Uddn=a0*(Un-U)-a2*Ud-a3*Udd
    #Solve for velocity
    Udn=Ud+a6*Udd+a7*Uddn

    deltaX[NT-1]=50 #Setting a high value of dx to enter loop
    iter=0

    while abs(sum(deltaX))>tol:
        iter+=1
        print ("iteration", iter)

        #Creating the global K and R for NR
        Residual_global = np.zeros((NT+1,1))
        K_dynamic_global = np.zeros((NT+1,NT+1))

        #Assemble contributions for every element
        for i in range(NT):
            F_ext_el=F[i:i+2,0]
            Un_el=Un[i:i+2,0]
            Uddn_el=Uddn[i:i+2,0]

            F_int_el=np.power(Un_el,2)-1 #This is the nonlinear term f(x)
            Residual_element=F_ext_el-F_int_el-(dot(M_el,Uddn_el))
            K_dynam_el=K_static_el+(a0*M_el)
            K_dyn_temp=np.zeros((NT+1,NT+1))
            Residual_temp=np.zeros((NT+1,1))
            K_dyn_temp[i:i+2,i:i+2]=K_dynam_el
            Residual_temp[i:i+2,0]=Residual_element
            K_dynamic_global+=K_dyn_temp
            Residual_global+=Residual_temp
        #Remove fixed degrees of freedom
        for dof in restrained_dofs:
            K_dynamic_global = np.delete(K_dynamic_global, dof, axis=0)
            K_dynamic_global = np.delete(K_dynamic_global, dof, axis=1)
            Residual_global= np.delete(Residual_global, dof, axis=0)

        #Solve global system
        dx=linalg.solve(K_dynamic_global, Residual_global)

        #Correct displacement
        deltaX[1:NT+1,0]=dx.transpose()
        Un=Un + deltaX

        #Newmark for iteration
        #Solve for acceleration
        Uddn=a0*(Un-U)-a2*Ud-a3*Udd
        #Solve for velocity
        Udn=Ud+a6*Udd+a7*Uddn
        #Check convergence
        #Go to next timestep
    print ("")
    results_U[:,CurrentTime]=Un.transpose()
    results_Ud[:,CurrentTime]=Udn.transpose()
    results_Udd[:,CurrentTime]=Uddn.transpose()
    U=Un

####################################
# Proper Orthogonal Decomposition
####################################
Usvd, Sigma, VTsvd =svd(results_U)

Phi=Usvd
Phi=Phi[:,0:4]    #Select the number of modes to use
NumberOfDOF=int (Phi.shape[1])
deltaQ= np.zeros((NumberOfDOF,1))
q=np.zeros((NumberOfDOF,1)) #predicted displacement 1
qn=q
qd=np.zeros((NumberOfDOF,1))#initializing Velocity
qdn=qd
qdd=np.zeros((NumberOfDOF,1)) #initializing acceleration
qddn=qdd
F= np.zeros((len(U), 1)) #creating the force of zeros
F[NT]=0.2 #Applying a force at the end of the bar
#K_static_el=(np.array([[1,-1],[-1,1]]) * float(NT))      #K static is the
    same as before
M_el= np.array([[2,1],[1,2]]) / (6. * NT )
results_U_svd= zeros(((len(U)), Ntp))
results_Ud_svd= zeros(((len(U)), Ntp))
results_Udd_svd= zeros(((len(U)), Ntp))

#Time step loop
for CurrentTime in range (Ntp):
    print ("time step", CurrentTime)

    q=qn
    qd=qdn
    qdd=qddn
    qn=q+(qd*dt)
    #Newmark for time step
    #Solve for acceleration
    qddn=a0*(qn-q)-a2*qd-a3*qdd
    #Solve for velocity
    qdn=qd+a6*qdd+a7*qddn

    deltaQ[(NumberOfDOF-1),0]=50 #Setting a high value of dx to enter loop
    iter=0

    while abs(sum(deltaQ))>tol:
        iter+=1
        print ("iteration", iter)

        #Creating the global K and R for NR
        Residual_global = np.zeros((NumberOfDOF,1))
        K_dynamic_global = np.zeros((NumberOfDOF,NumberOfDOF))

        #Assemble contributions for every element
        for i in range(NT):
            Phi_elem = Phi[i:i+2,:]
            F_ext_el=(F[i:i+2,0:1])
            Un_el=dot(Phi_elem,qn)
            Uddn_el=dot(Phi_elem,qddn)

            #Calculate Residual
            F_int_el=np.power(Un_el,2)-1 #This is the nonlinear term f(x)
            Residual_element=F_ext_el-F_int_el-(dot(M_el,Uddn_el))
            Residual_element_svd=dot(Phi_elem.transpose(),Residual_element)

            #Calculate Tangent
            K_dynam_el=K_static_el+(a0*M_el)
            K_dynam_el_svd=np.matmul(Phi_elem.transpose(),(np.matmul(
   K_dynam_el, Phi_elem)))
            K_dynamic_global+=K_dynam_el_svd
            Residual_global+=Residual_element_svd

        #Solve global system
        dx=linalg.solve(K_dynamic_global, Residual_global)

        #Correct displacement
        deltaQ=dx
        qn=qn + deltaQ

        #Newmark for iteration
        #Solve for acceleration
        qddn=a0*(qn-q)-a2*qd-a3*qdd
        #Solve for velocity
        qdn=qd+a6*qdd+a7*qddn

        #Check convergence
        #Go to next timestep
    print ("")
    results_U_svd[:,CurrentTime]=(dot(Phi,qn)).transpose()
    results_Ud_svd[:,CurrentTime]=(dot(Phi,qdn)).transpose()
    results_Udd_svd[:,CurrentTime]=(dot(Phi,qddn)).transpose()


################################################################
# Plotting
################################################################
Xaxis=np.array(np.linspace(0,tt,Ntp))
nodeToPrint=NT-1

```

```
plt.figure()
line_up, = plt.plot(Xaxis,results_U[nodeToPrint,:], 'r-o', label='Disp Full
    Model')
line_down,= plt.plot(Xaxis,results_U_svd[nodeToPrint,:], 'go', label='Disp
    SVD 5 modes')
plt.title('Displacement U')
plt.legend(handles=[line_up, line_down])

plt.figure()
line_up, =plt.plot(Xaxis,results_Ud[nodeToPrint,:], 'r-o', label='Vel Full
    Model')
line_down, =plt.plot(Xaxis,results_Ud_svd[nodeToPrint,:], 'go',label='Vel
    SVD 5 Model')
plt.title('Velocity')
plt.legend(handles=[line_up, line_down])

plt.figure()
line_up, =plt.plot(Xaxis,results_Udd[nodeToPrint,:], 'r-o',label='Acc Full
    Model')
line_down, =plt.plot(Xaxis,results_Udd_svd[nodeToPrint,:], 'go', label='Acc
    SVD 5 Modes')
plt.title('Acceleration')
plt.legend(handles=[line_up, line_down])

plt.figure()
Sigma_Plot_X=np.linspace(0,1,len(Sigma))
line_up,=plt.plot(Sigma_Plot_X, Sigma, 'ro-', label='Singular Values')
plt.title('Sigma')
plt.legend(handles=[line_up])
```