Chapter 5
PGD for Dynamical Problems

Simulations are like Michelin star restaurants but should be like McDonalds: ubiquitous and standardised.

—Craig McIlhenny

Abstract This chapter develops the application of PGD methods initial and boundary value problems, with an eye towards the (non-linear) solid dynamics equations.

Model order reduction of initial and boundary value problems (IBVP) is a particularly challenging task. In this chapter we explain some interesting concepts related mostly with solid dynamics, taken as model problem to this end.

In [41] a method was developed that takes the field of initial conditions as a parameter to develop a very efficient dynamical integrator. However, the field of initial conditions (displacement, velocity) is in fact a parameter of infinite dimension, and hence hard to parameterize adequately. In this chapter we analyze how to do it in a proper way so as to render a very fast method, amenable for real-time simulation, even under very astringent conditions. Other approaches to the problem, such as a space-time one, can be found at [18], for instance.

5.1 Taking Initial Conditions as Parameters

As mentioned before, in [41] a method is developed based on PGD that acts as a sort of *black box* integrator in time. Given the converged displacement and velocity field of the solid at time step t, $\boldsymbol{u}^t$ and $\dot{\boldsymbol{u}}^t$, respectively, as parameters, the method returns the displacement and velocity fields at time $t + \triangle t$, see Fig. 5.1.

Once semi-discretized in space, the displacement and velocity fields are no longer of infinite dimension, but usual engineering finite element meshes involve tens of

E. Cueto et al., *Proper Generalized Decompositions*, SpringerBriefs
in Applied Sciences and Technology, DOI 10.1007/978-3-319-29994-5_5

Fig. 5.1 Sketch of the proposed method for the integration of solid dynamics in the PGD framework. Converged displacement and velocity fields at time step t are taken as parameters, so as to provide, without the need of any matrix inversion, the displacement and velocity fields at time step $t + \triangle t$

thousands to millions of degrees of freedom. This would imply to have into account millions of parameters, something out of reach even for PGD methods.

In order to avoid this enormous number of parameters, in [41] the use of Proper Orthogonal Decomposition [43, 48, 49] methods so as to employ a minimal number of parameters is proposed. In this way, initial displacement and velocity field can be optimally parameterized with a minimal number of degrees of freedom. The price to pay is to project the results of the integration at time $t + \triangle t$ onto the POD basis so as to be taken as parameters (initial conditions) for a subsequent integration to obtain $\boldsymbol{u}^{t+2\triangle t}$ and $\dot{\boldsymbol{u}}^{t+2\triangle t}$.

We provide details of the variational formulation in the subsequent sections.

5.2 Developing the Weak Form of the Problem

We consider the general problem of solid dynamics, in which we look for the displacement field,

$$\boldsymbol{u} : \bar{\Omega} \times]0, T] \times \mathcal{I} \times \mathcal{J} \rightarrow \mathbb{R}^3,$$

where $\mathcal{I} = [\boldsymbol{u}_0^-, \boldsymbol{u}_0^+]$ and $\mathcal{J} = [\dot{\boldsymbol{u}}_0^-, \dot{\boldsymbol{u}}_0^+]$ represent the considered intervals of variation of initial boundary conditions, $\boldsymbol{u}_0$ and $\dot{\boldsymbol{u}}_0$, taken as parameters. To obtain a parametric solution for any initial condition (within these intervals), it is therefore necessary to define a new (triply-) weak form:

given $\boldsymbol{f}, \boldsymbol{g}, \boldsymbol{h}, \boldsymbol{u}_0$ *and* $\dot{\boldsymbol{u}}_0$ *find* $\boldsymbol{u}(t) \in \mathcal{S}_t = \{\boldsymbol{u} | \boldsymbol{u}(\boldsymbol{x}, t) = \boldsymbol{g}(\boldsymbol{x}, t),\ \boldsymbol{x} \in \Gamma_u,\ \boldsymbol{u} \in \mathcal{H}^1(\Omega)\}, t \in [0, T]$, *such that for all* $\boldsymbol{u}^* in \mathcal{V}\{\boldsymbol{u} | \boldsymbol{u}(\boldsymbol{x}, t) = \boldsymbol{0},\ \boldsymbol{x} \in \Gamma_u,\ \boldsymbol{u} \in \mathcal{H}^1(\Omega)\}$,

$$(\boldsymbol{u}^*, \rho \ddot{\boldsymbol{u}}) + a(\boldsymbol{u}^*, \boldsymbol{u}) = (\boldsymbol{u}^*, \boldsymbol{f}) + (\boldsymbol{u}^*, \boldsymbol{h})_\Gamma \tag{5.1a}$$

$$(\boldsymbol{u}^*, \rho \boldsymbol{u}(0)) = (\boldsymbol{u}^*, \rho \boldsymbol{u}_0) \tag{5.1b}$$

$$(\boldsymbol{u}^*, \rho \dot{\boldsymbol{u}}(0)) = (\boldsymbol{u}^*, \rho \dot{\boldsymbol{u}}_0), \tag{5.1c}$$

where:

$$a(\boldsymbol{u}^*, \boldsymbol{u}) = \int_{\mathcal{I}} \int_{\mathcal{J}} \int_{\Omega} \nabla^s \boldsymbol{u}^* : \mathbf{C} : \nabla^s \boldsymbol{u} \; d\Omega d\dot{\boldsymbol{u}}_0 d\boldsymbol{u}_0,$$
$$(\boldsymbol{u}^*, \boldsymbol{f}) = \int_{\mathcal{I}} \int_{\mathcal{J}} \int_{\Omega} \boldsymbol{u}^* \boldsymbol{f} \; d\Omega d\dot{\boldsymbol{u}}_0 d\boldsymbol{u}_0,$$
$$(\boldsymbol{u}^*, \boldsymbol{h})_{\Gamma} = \int_{\mathcal{I}} \int_{\mathcal{J}} \int_{\Gamma_t} \boldsymbol{u}^* \boldsymbol{h} \; d\Gamma d\dot{\boldsymbol{u}}_0 d\boldsymbol{u}_0.$$

The next step is, by means of appropriate finite-dimensional approximations to $\mathcal{S}_t$ and $\mathcal{V}$, $\mathcal{S}_t^h$ and $\mathcal{V}^h$, respectively, to semi-discretize the weak form so as to obtain the following problem:

given $\boldsymbol{f}, \boldsymbol{g}, \boldsymbol{h}, \boldsymbol{u}_0$ *and* $\dot{\boldsymbol{u}}_0$ *find* $\boldsymbol{u}^h(t) = \boldsymbol{v}^h + \boldsymbol{g}^h \in \mathcal{S}_t^h$ *(note that* $\boldsymbol{g}(\boldsymbol{x}, t) = \boldsymbol{u}(\boldsymbol{x}, t)$ *on* Γ_u*) such that for every* $\boldsymbol{u}^{*h} \in \mathcal{V}^h$,

$$(\boldsymbol{u}^{*h}, \rho \ddot{\boldsymbol{v}}^h) + a(\boldsymbol{u}^{*h}, \boldsymbol{v}) = (\boldsymbol{u}^{*h}, \boldsymbol{f}) + (\boldsymbol{u}^{*h}, \boldsymbol{h})_{\Gamma} - (\boldsymbol{u}^{*h}, \rho \ddot{\boldsymbol{g}}^h) - a(\boldsymbol{u}^{*h}, \boldsymbol{g}^h), \tag{5.2a}$$
$$(\boldsymbol{u}^{*h}, \rho \boldsymbol{v}^h(0)) = (\boldsymbol{u}^{*h}, \rho \boldsymbol{u}_0) - (\boldsymbol{u}^{*h}, \rho \boldsymbol{g}^h(0)), \tag{5.2b}$$
$$(\boldsymbol{u}^{*h}, \rho \dot{\boldsymbol{v}}^h(0)) = (\boldsymbol{u}^{*h}, \rho \dot{\boldsymbol{u}}_0) - (\boldsymbol{u}^{*h}, \rho \dot{\boldsymbol{g}}^h(0)). \tag{5.2c}$$

This provides a sort of response surface or parametric solution (thus the name *computational vademecum* coined in [26]) to the problem (5.1) for *any* initial conditions.

If we consider direct integration in time (remember that we look for an interactive method, so that this prevents us from using a space-time approach) the sought displacement field will be approximated in a PGD framework as a finite series of separable functions,

$$\boldsymbol{v}^h(\boldsymbol{x}, t, \boldsymbol{u}_0, \dot{\boldsymbol{u}}_0) = \left[\sum_{i=1}^{n} \boldsymbol{F}^i(\boldsymbol{x}) \circ \boldsymbol{G}^i(\boldsymbol{u}_0) \circ \boldsymbol{H}^i(\dot{\boldsymbol{u}}_0) \right] \circ \boldsymbol{d}(t), \tag{5.3}$$

where the nodal coefficients $\boldsymbol{d}(t)$ carry out all the time-dependency of the solution and the symbol "$\circ$" stands for the entry-wise Hadamard or Schur multiplication of vectors.

Functions $\boldsymbol{F}$, $\boldsymbol{G}$ and $\boldsymbol{H}$ will be expressed in terms of low (here, three-) dimensional finite element basis functions. As usual, these are computed by means of a greedy algorithm in which one sum is computed at a time, while one product is computed in a fixed point, alternated directions algorithm. Thus, having an approximation to $\boldsymbol{v}^h$ converged at iteration n, the $(n+1)$-th term is obtained as

$$\boldsymbol{v}^{n+1}(\boldsymbol{x}, t, \boldsymbol{u}_0, \dot{\boldsymbol{u}}_0) = \left[\sum_{i=1}^{n} \boldsymbol{F}^i(\boldsymbol{x}) \circ \boldsymbol{G}^i(\boldsymbol{u}_0) \circ \boldsymbol{H}^i(\dot{\boldsymbol{u}}_0) + \boldsymbol{R}(\boldsymbol{x}) \circ \boldsymbol{S}(\boldsymbol{u}_0) \circ \boldsymbol{T}(\dot{\boldsymbol{u}}_0) \right] \circ \boldsymbol{d}(t).$$

By substituting the approximations to $\boldsymbol{v}^h$ and $\boldsymbol{w}^h$ into the weak form of the problem, Eq. (5.2), we arrive at a semi-discrete problem. One of the most salient features of this method relies in its ability of advancing in time using any time integrator existing in the literature (particularly, energy and momentum conserving schemes.) Of course, any other parametric dependence, such as the one on the position of the applied load, see Chap. 3, can be considered at the same time.

Still one ingredient of the implementation is missing. Instead of considering the whole time interval $]0, T]$ we look for the solution within a generic time increment $]0, \triangle t]$:

$$\boldsymbol{v} : \bar{\Omega} \times]0, \triangle t] \times \mathcal{I} \times \mathcal{J} \times [h^-, h^+] \rightarrow \mathbb{R}^3,$$

where $\triangle t$ represents the necessary time to response prescribed by the particular envisaged application. For instance, for haptic feedback applications, a physically realistic sensation of touch needs for some 500 Hz to 1 kHz feedback rate. This implies to take $\triangle t$ on the order of 0.001 seconds. This value $\triangle t$ could be smaller or greater than the necessary time step amplitude needed for stability of the chosen time integrator.

5.3 Matrix Form of the Problem

5.3.1 Time Integration of the Equations of Motion

As usual, we start from the weak form of the solid dynamic equations, Eq. (5.1), i.e., finding the displacement $\boldsymbol{u} \in \mathcal{H}^1$ such that for all $\boldsymbol{u}^* \in \mathcal{H}^1_0$:

$$\int_\Omega \boldsymbol{u}^* \rho \ddot{\boldsymbol{u}} d\Omega + \int_\Omega \nabla_s \boldsymbol{u}^* : \mathsf{C} : \nabla_s \boldsymbol{u} d\Omega = \int_{\Gamma_{t2}} \boldsymbol{u}^* \cdot \boldsymbol{t} d\Gamma. \tag{5.4}$$

Once semi-discretized in space, we can identify a term where a mass matrix appears,

$$\boldsymbol{M_m} = \int_\Omega \boldsymbol{u}^{*h} \rho \ddot{\boldsymbol{u}}^h d\Omega,$$

and a term usually identified as the stiffness matrix

$$\boldsymbol{K_m} = \int_\Omega \nabla_s \boldsymbol{u}^{*h} : \mathsf{C} : \nabla_s \boldsymbol{u}^h d\Omega.$$

In the sequel we omit, if no risk of confusion exists, the superscript h, so that all vectors represent the set of nodal unknowns for the problem. For the integration in time of these equations we have several options. In general you can employ your

favorite integration scheme. Here we are considering, both for its simplicity and good results, an energy-momentum conserving scheme developed in [15]. This scheme has two sub-steps which compute a predictor of the displacement vector at time step $u_{t+(\Delta t/2)}$ in the first one and subsequently a correction $u_{t+\Delta t}$ in the second sub-step.

The first sub-step has the following form:

$$\boldsymbol{M_m}\ddot{\boldsymbol{u}}_{t+(\Delta t/2)} + \boldsymbol{K_m}\boldsymbol{u}_{t+(\Delta t/2)} = \boldsymbol{F}_{t+(\Delta t/2)}.$$

Employing classical finite difference approaches for the time derivatives,

$$\ddot{\boldsymbol{u}}_{t+(\Delta t/2)} = \frac{\dot{\boldsymbol{u}}_{t+(\Delta t/2)} - \dot{\boldsymbol{u}}_t}{\Delta t/4} - \ddot{\boldsymbol{u}}_t,$$

$$\dot{\boldsymbol{u}}_{t+(\Delta t/2)} = \frac{\boldsymbol{u}_{t+(\Delta t/2)} - \boldsymbol{u}_t}{\Delta t/4} - \dot{\boldsymbol{u}}_t.$$

Applying these expressions to the first sub-step, after some simple algebra, we obtain the final expression for the sub-step 1:

$$\left[\left[\frac{16}{\Delta t^2}\right]\boldsymbol{M_m} + \boldsymbol{K_m}\right]\cdot\boldsymbol{u}_{t+(\Delta t/2)} = \boldsymbol{F}_{t+(\Delta t/2)} + \left[\frac{16}{\Delta t^2}\right]\boldsymbol{M_m}\cdot\boldsymbol{u}_t + \left[\frac{8}{\Delta t}\right]\boldsymbol{M_m}\cdot\dot{\boldsymbol{u}}_t + \boldsymbol{M_m}\cdot\ddot{\boldsymbol{u}}_t. \tag{5.5}$$

The second sub-step has the following form:

$$\boldsymbol{M_m}\ddot{\boldsymbol{u}}_{t+\Delta t} + \boldsymbol{K_m}\boldsymbol{u}_{t+\Delta t} = \boldsymbol{F}_{t+\Delta t}.$$

Again, by employing classical finite differences for the time derivatives,

$$\ddot{\boldsymbol{u}}_{t+\Delta t} = \frac{\dot{\boldsymbol{u}}_t}{\Delta t} - \left[\frac{4}{\Delta t}\right]\dot{\boldsymbol{u}}_{t+(\Delta t/2)} + \left[\frac{3}{\Delta t}\right]\dot{\boldsymbol{u}}_{t+\Delta t},$$

$$\dot{\boldsymbol{u}}_{t+\Delta t} = \frac{\boldsymbol{u}_t}{\Delta t} - \left[\frac{4}{\Delta t}\right]\boldsymbol{u}_{t+(\Delta t/2)} + \left[\frac{3}{\Delta t}\right]\boldsymbol{u}_{t+\Delta t}.$$

By substituting these expressions in the second sub-step, the final formula for the sub-step 2 that the reader can find in the code included in Sect. 5.4 is:

$$\left[\left[\frac{9}{\Delta t^2}\right]\boldsymbol{M_m} + \boldsymbol{K_m}\right]\cdot\boldsymbol{u}_{t+\Delta t} = \boldsymbol{F}_{t+\Delta t} - \left[\frac{19}{\Delta t^2}\right]\boldsymbol{M_m}\cdot\boldsymbol{u}_t - \left[\frac{5}{\Delta t}\right]\boldsymbol{M_m}\cdot\dot{\boldsymbol{u}}_t + \left[\frac{28}{\Delta t^2}\right]\boldsymbol{M_m}\cdot u_{t+(\Delta t/2)}. \tag{5.6}$$

The strategy depicted in the previous section, when applied to the just explained time integration scheme, implies the construction of a PGD time integrator able to provide the value of $\boldsymbol{u}_{t+\triangle t}$ for any value of $\boldsymbol{u}_t$. In that framework, $\boldsymbol{u}_t$ acts in fact as a parameter. But recall that $\boldsymbol{u}_t$ represents the vector of nodal displacements at time step t. Therefore, it can consist of several millions of degrees of freedom, something our of reach even for PGD strategies!

In order to develop a suitable strategy, it is therefore of utmost importance to adequately parameterize the field of initial displacements at the beginning of the time step. In [41] this is done by employing a reduced-order basis instead of the traditional finite element one. And to do it by means of Proper Orthogonal Decompositions. This is explained in detail in what follows.

5.3.2 *Computing a Reduced-Order Basis for the Field of Initial Conditions*

For the sake of completeness, we briefly review here the basics of the POD technique for the computation of a reduced-order basis for the initial displacement field of the problem. Let us first assume that we have a collection of *snapshots*, i.e., finite element results for problems similar to the one at hand. By *similar* we mean results probably for the same solid, but possibly under different conditions, applied loads, boundary conditions, … We then store these snapshots column-wise in a matrix $\boldsymbol{Q}$ (more details can be found, for instance, in [54]). The next step is the computation of the so-called *auto-correlation matrix*,

$$\boldsymbol{c} = \boldsymbol{Q}\ \boldsymbol{Q}^T. \tag{5.7}$$

It can then be demonstrated that the best possible basis (that capturing the most of the energy of the system with the minimal number of degrees of freedom) is formed by the eigenvectors $\boldsymbol{\phi}$ of the problem

$$\boldsymbol{c}\ \boldsymbol{\phi} = \alpha \boldsymbol{\phi}.$$

By storing the nodal values (we assume to have N nodes in the mesh of the model) of the eigenvectors with the m biggest eigenvalues in a matrix

$$\mathbf{B} = \begin{pmatrix} \phi^1(\boldsymbol{x}_1) & \phi^2(\boldsymbol{x}_1) & \cdots & \phi^m(\boldsymbol{x}_1) \\ \phi^1(\boldsymbol{x}_2) & \phi^2(\boldsymbol{x}_2) & \cdots & \phi^m(\boldsymbol{x}_2) \\ \vdots & \vdots & \ddots & \vdots \\ \phi^1(\boldsymbol{x}_N) & \phi^2(\boldsymbol{x}_N) & \cdots & \phi^m(\boldsymbol{x}_N) \end{pmatrix}$$

we can therefore project the initial system of equations onto a reduced-order one by simply doing the change of variable

$$\boldsymbol{u}_t \approx \sum_{i=1}^{i=\mathtt{nrb}} \zeta_t^i \, \boldsymbol{\phi}^i \; = \; \boldsymbol{B} \, \boldsymbol{\zeta}_t,$$

so that we will finally face a system of $3m$ equations for $\boldsymbol{\zeta}_t$ instead of the original $3N$ for $\boldsymbol{u}_t$. The advantage of this strategy is that usually the number of reduced basis, $\mathtt{rnb} \ll N$ and therefore the resulting system of equations is generally much smaller.

5.3.3 Projection of the Equations onto a Reduced, Parametric Basis

For each sub-step within the time integration scheme we compute the PGD approximation to the solution $\boldsymbol{u}_{t+(\triangle t/2)}$ and $\boldsymbol{u}_{t+\triangle t}$ such that,

$$\begin{aligned}\boldsymbol{u}_{t+(\triangle t/2)}&(\boldsymbol{x}, \boldsymbol{\zeta}_t, \dot{\boldsymbol{\zeta}}_t, \ddot{\boldsymbol{\zeta}}_t, p_a, \boldsymbol{s}) \\ &= \sum_{i=1}^{n} \boldsymbol{N}^T(\boldsymbol{x}) \boldsymbol{F}_{\boldsymbol{x}}^i \cdot \boldsymbol{N}^T(\boldsymbol{\zeta}_t) \boldsymbol{F}_{\boldsymbol{\zeta}_t}^i \cdot \boldsymbol{N}^T(\dot{\boldsymbol{\zeta}}_t) \boldsymbol{F}_{\dot{\boldsymbol{\zeta}}_t}^i \cdot \boldsymbol{N}^T(\ddot{\boldsymbol{\zeta}}_t) \boldsymbol{F}_{\ddot{\boldsymbol{\zeta}}_t}^i \cdot \boldsymbol{N}^T(p_a) \boldsymbol{F}_{p_a}^i \cdot \boldsymbol{N}^T(\boldsymbol{s}) \boldsymbol{F}_{\boldsymbol{s}}^i\end{aligned} \tag{5.8}$$

$$\begin{aligned}\boldsymbol{u}_{t+\triangle t}&(\boldsymbol{x}, \boldsymbol{\zeta}_t, \dot{\boldsymbol{\zeta}}_t, \boldsymbol{\zeta}_{t+(\triangle t/2)}, p_a, \boldsymbol{s}) \\ &= \sum_{i=1}^{n} \boldsymbol{N}^T(\boldsymbol{x}) \boldsymbol{F}_{\boldsymbol{x}}^i \cdot \boldsymbol{N}^T(\boldsymbol{\zeta}_t) \boldsymbol{F}_{\boldsymbol{\zeta}_t}^i \cdot \boldsymbol{N}^T(\dot{\boldsymbol{\zeta}}_t) \boldsymbol{F}_{\dot{\boldsymbol{\zeta}}_t}^i \cdot \boldsymbol{N}^T(\boldsymbol{\zeta}_{t+(\triangle t/2)}) \boldsymbol{F}_{\boldsymbol{\zeta}_{t+(\triangle t/2)}}^i \cdot \boldsymbol{N}^T(p_a) \boldsymbol{F}_{p_a}^i \cdot \boldsymbol{N}^T(\boldsymbol{s}) \boldsymbol{F}_{\boldsymbol{s}}^i\end{aligned} \tag{5.9}$$

where $\boldsymbol{x}$ represents the physical space, $\boldsymbol{u}_t$ is the vector of displacement degrees of freedom at time step t, $\dot{\boldsymbol{u}}_n$ is the vector of velocity degrees of freedom at time step t, $\ddot{\boldsymbol{u}}_n$ is the vector of nodal accelerations at time step t, $\boldsymbol{u}_{t+(\triangle t/2)}$ is the vector of nodal displacements at time step $t + (\triangle t/2)$, p_a is the classical loading parameter, its value varying continuously in the interval [0, 1]. It allows us to apply or not a load at a particular time step or to apply a ramp load, for instance. Finally, $\boldsymbol{s}$ represents the position of the load, as in Chap. 3.

We denote, as in the rest of the book, by $\boldsymbol{N}(\cdot)$ the vector of finite element shape functions employed to discretize the different dimensions of the problem. Note that we are considering a solution depending on the physical space $\boldsymbol{x}$ plus a number of parameters $\boldsymbol{\zeta} = [\zeta_1, \zeta_2, \ldots, \zeta_m]$, which in this case coincide with the chosen POD degrees of freedom parameterizing the fields of initial displacements and velocities. Taking also into account that the density parameter ρ and the symmetric gradients $\boldsymbol{\nabla}_s$ in Eq. (5.1) depend solely on space coordinates, we can write the mass matrix, and the stiffness matrix of the problem in separated form as

$$\boldsymbol{M_m} = \left[\int_{\Omega_{\boldsymbol{x}}} \boldsymbol{N}^T(\boldsymbol{x})\rho\boldsymbol{N}(\boldsymbol{x})d\Omega_{\boldsymbol{x}}\right]\cdot\left[\int_{\Omega_{\zeta_1}} \boldsymbol{N}^T(\zeta_1)\boldsymbol{N}(\zeta_1)d\Omega_{\zeta_1}\right]\cdot\ldots$$
$$\cdot\left[\int_{\Omega_{\zeta_m}} \boldsymbol{N}^T(\zeta_m)\boldsymbol{N}(\zeta_m)d\Omega_{\zeta_m}\right]\cdot\left[\int_{\Omega_{p_a}} \boldsymbol{N}^T(p_a)\boldsymbol{N}(p_a)d\Omega_{p_a}\right]\cdot\left[\int_{\Omega_{\boldsymbol{s}}} \boldsymbol{N}^T(\boldsymbol{s})\boldsymbol{N}(\boldsymbol{s})d\Omega_{\boldsymbol{s}}\right],$$

$$\boldsymbol{K_m} = \left[\int_{\Omega_{\boldsymbol{x}}} \nabla_s\boldsymbol{N}^T(\boldsymbol{x})\mathbf{C}\nabla_s\boldsymbol{N}(\boldsymbol{x})d\Omega_{\boldsymbol{x}}\right]\cdot\left[\int_{\Omega_{\zeta_1}} \boldsymbol{N}^T(\zeta_1)\boldsymbol{N}(\zeta_1)d\Omega_{\zeta_1}\right]\cdot\ldots$$
$$\cdot\left[\int_{\Omega_{\zeta_m}} \boldsymbol{N}^T(\zeta_m)\boldsymbol{N}(\zeta_m)d\Omega_{\zeta_m}\right]\left[\int_{\Omega_{p_a}} \boldsymbol{N}^T(p_a)\boldsymbol{N}(p_a)d\Omega_{p_a}\right]\cdot\left[\int_{\Omega_{\boldsymbol{s}}} \boldsymbol{N}^T(\boldsymbol{s})\boldsymbol{N}(\boldsymbol{s})d\Omega_{\boldsymbol{s}}\right].$$

The influence on the solution of the number of parameters m (terms in the POD basis of the initial conditions) chosen to parameterize the fields of initial conditions was deeply analyzed in [41].

The PGD final solution uses the PGD solutions for each sub-step, Eqs. (5.5) and (5.6). Starting from $\boldsymbol{u}_0$, $\dot{\boldsymbol{u}}_0$ and $\ddot{\boldsymbol{u}}_0$ (in fact, their projection onto the POD basis!), by using the first sub-step vademecum we obtain $\boldsymbol{u}_{0+1/2}$. These values are then introduced in second sub-step vademecum which in turn uses as input parameters $\boldsymbol{u}_0$, $\dot{\boldsymbol{u}}_0$ and $\boldsymbol{u}_{0+1/2}$ (computed in the first sub-step). The value returned by the second vademecum is $\boldsymbol{u}_1$. We then change the time step, and apply the new input parameters $\boldsymbol{u}_1$, $\dot{\boldsymbol{u}}_1$ and $\ddot{\boldsymbol{u}}_1$ in the first vademecum to obtain $\boldsymbol{u}_{1+1/2}$. Again, by using $\boldsymbol{u}_1$, $\dot{\boldsymbol{u}}_1$ and $\boldsymbol{u}_{1+1/2}$ we obtain $\boldsymbol{u}_2$ by using the second vademecum. This procedure is repeated for each time step of the simulation. This loop indeed runs under very astringent real time constraints, such as those of a realistic rendering.

As in previous examples, we assume for simplicity of the exposition, that the load is of unity module and acts along the vertical axis: $\boldsymbol{t} = \boldsymbol{e}_k \cdot \delta(\boldsymbol{x} - \boldsymbol{s})$, where δ represents the Dirac-delta function and $\boldsymbol{e}_k$ the unit vector along the z-coordinate axis on the top of the domain. A more general setting would need for new parameters, i.e., the components of the load vector, for instance, but it is perfectly possible in the same framework here explained.

Regarding the matrix structure of the problems given by Eqs. (5.5) and (5.6), in both of them the force vector applied at each time step, $\boldsymbol{F}_{n+1/2}$ and $\boldsymbol{F}_{n+1}$ appears. Like in the parametric problem in Chap. 3, we must consider the load in a separated form, i.e. a separated sum of products of separated functions,

$$f_{n+1/2}(\boldsymbol{x}, \boldsymbol{\zeta}_n, \dot{\boldsymbol{\zeta}}_n, \ddot{\boldsymbol{\zeta}}_n, p_a, \boldsymbol{s}) = \sum_{j=1}^{m} \boldsymbol{f}_{\boldsymbol{x}}^j \cdot \boldsymbol{f}_{\boldsymbol{\zeta}_n}^j \cdot \boldsymbol{f}_{\dot{\boldsymbol{\zeta}}_n}^j \cdot \boldsymbol{f}_{\ddot{\boldsymbol{\zeta}}_n}^j \cdot \boldsymbol{f}_{p_c}^j \cdot \boldsymbol{f}_{\boldsymbol{s}}^j. \tag{5.10}$$

A simple way to obtain such a decomposition is to consider as many terms j as possible nodal load locations, and to set $\boldsymbol{f}_{\boldsymbol{x}}^j$, as the force modulus (here, unity). In turn, $\boldsymbol{f}_{p_c}^j = [0\ 1]^T$ for each j, $\boldsymbol{f}_{\boldsymbol{s}}^j = \boldsymbol{I}$ (the identity matrix) and the rest of vectors $\boldsymbol{f}_{\boldsymbol{\zeta}_n}^j$, $\boldsymbol{f}_{\dot{\boldsymbol{\zeta}}_n}^j$, $\boldsymbol{f}_{\ddot{\boldsymbol{\zeta}}_n}^j = \mathbf{1}$, that is, the `ones` instruction in Matlab, a vector composed by

ones in every entry. We proceed analogously for the force vector $\boldsymbol{f}_{n+1}$ in the second substep of the time integrator scheme.

In order to completely define the right-hand side vector, let us see how to compute the others terms in Eqs. (5.5) and (5.6). In these formulae, we find terms defined as a constant value multiplying the mass matrix $\boldsymbol{M_m}$ and multiplying $\boldsymbol{\zeta}_n$, $\dot{\boldsymbol{\zeta}}_n$, $\ddot{\boldsymbol{\zeta}}_n$ and $\boldsymbol{u}_{n+1}$. These vectorial parameters should be considered in separated form, so as to have the following form,

$$\begin{aligned}
\boldsymbol{\zeta}_n &= \bar{\mathbf{1}}_{\boldsymbol{x}} \cdot \left[\zeta_n^{min} \ \ldots \ \zeta_n^{max}\right]^T \cdot \mathbf{1}_{\dot{\zeta}_n} \cdot \mathbf{1}_{\ddot{\zeta}_n} \cdot \mathbf{1}_{p_c} \cdot \mathbf{1}_{\boldsymbol{s}}, \\
\dot{\boldsymbol{\zeta}}_n &= \bar{\mathbf{1}}_{\boldsymbol{x}} \cdot \mathbf{1}_{\zeta_n} \cdot \left[\dot{\zeta}_n^{min} \ \ldots \ \dot{\zeta}_n^{max}\right]^T \cdot \mathbf{1}_{\ddot{\zeta}_n} \cdot \mathbf{1}_{p_c} \cdot \mathbf{1}_{\boldsymbol{s}}, \\
\ddot{\boldsymbol{\zeta}}_n &= \bar{\mathbf{1}}_{\boldsymbol{x}} \cdot \mathbf{1}_{\zeta_n} \cdot \mathbf{1}_{\dot{\zeta}_n} \cdot \left[\ddot{\zeta}_n^{min} \ \ldots \ \ddot{\zeta}_n^{max}\right]^T \cdot \mathbf{1}_{p_c} \cdot \mathbf{1}_{\boldsymbol{s}}, \\
\boldsymbol{\zeta}_{n+1/2} &= \bar{\mathbf{1}}_{\boldsymbol{x}} \cdot \mathbf{1}_{\zeta_n} \cdot \mathbf{1}_{\dot{\zeta}_n} \cdot \left[\zeta_{n+1/2}^{min} \ \ldots \ \zeta_{n+1/2}^{max}\right]^T \cdot \mathbf{1}_{p_c} \cdot \mathbf{1}_{\boldsymbol{s}},
\end{aligned}$$

where vector $\bar{\mathbf{1}}_{\boldsymbol{x}}$ refers to the ones vector in space direction that satisfies essential boundary conditions or, equivalently, the ones vector in which entries related to nodes pertaining to the essential boundary have been replaced by zeros.

In the code reproduced below, these values of the RHS vector are computed for each reduced basis `nrb` and at each substep, and are saved in the $\boldsymbol{FR1}$ and $\boldsymbol{FR2}$ vectors. So for substep 1,

$$\begin{aligned}
\boldsymbol{FR1}_1^j &= \bar{\mathbf{1}}_{\boldsymbol{x}} \cdot \left[\zeta_n^{min} \ \ldots \ \zeta_n^{max}\right]^T \cdot \mathbf{1}_{\dot{\zeta}_n} \cdot \mathbf{1}_{\ddot{\zeta}_n} \cdot \mathbf{1}_{p_c} \cdot \mathbf{1}_{\boldsymbol{s}}, \\
\boldsymbol{FR1}_2^j &= \bar{\mathbf{1}}_{\boldsymbol{x}} \cdot \mathbf{1}_{\zeta_n} \cdot \left[\dot{\zeta}_n^{min} \ \ldots \ \dot{\zeta}_n^{max}\right]^T \cdot \mathbf{1}_{\ddot{\zeta}_n} \cdot \mathbf{1}_{p_c} \cdot \mathbf{1}_{\boldsymbol{s}}, \\
\boldsymbol{FR1}_3^j &= \bar{\mathbf{1}}_{\boldsymbol{x}} \cdot \mathbf{1}_{\zeta_n} \cdot \mathbf{1}_{\dot{\zeta}_n} \cdot \left[\ddot{\zeta}_n^{min} \ \ldots \ \ddot{\zeta}_n^{max}\right]^T \cdot \mathbf{1}_{p_c} \cdot \mathbf{1}_{\boldsymbol{s}},
\end{aligned}$$

where $j = 1, \ldots,$ `nrb` refers to the number in the set of reduced basis.

Equivalently, in substep 2,

$$\begin{aligned}
\boldsymbol{FR2}_1^j &= \bar{\mathbf{1}}_{\boldsymbol{x}} \cdot \left[\zeta_n^{min} \ \ldots \ \zeta_n^{max}\right]^T \cdot \mathbf{1}_{\dot{\zeta}_n} \cdot \mathbf{1}_{\ddot{\zeta}_n} \cdot \mathbf{1}_{p_c} \cdot \mathbf{1}_{\boldsymbol{s}}, \\
\boldsymbol{FR2}_2^j &= \bar{\mathbf{1}}_{\boldsymbol{x}} \cdot \mathbf{1}_{\zeta_n} \cdot \left[\dot{\zeta}_n^{min} \ \ldots \ \dot{\zeta}_n^{max}\right]^T \cdot \mathbf{1}_{\ddot{\zeta}_n} \cdot \mathbf{1}_{p_c} \cdot \mathbf{1}_{\boldsymbol{s}}, \\
\boldsymbol{FR2}_3^j &= \bar{\mathbf{1}}_{\boldsymbol{x}} \cdot \mathbf{1}_{\zeta_n} \cdot \mathbf{1}_{\dot{\zeta}_n} \cdot \left[\zeta_n^{min} \ \ldots \ \zeta_n^{max}\right]^T \cdot \mathbf{1}_{p_c} \cdot \mathbf{1}_{\boldsymbol{s}},
\end{aligned}$$

where j refers to the number of reduced basis. We use the same discretization rationale for $\boldsymbol{u}_n$ and $\boldsymbol{u}_{n+1/2}$.

To solve both substeps so as to generate the multi-parametric solution, we employ a greedy algorithm, using a fixed-point strategy, so as to compute the new terms in the sum, represented by Eqs. (5.8) and (5.9). If, within the enrichment loop, the solution is not accurate enough, the already computed approximation is improved by adding a new separated term

$$\boldsymbol{u}_{n+1/2} = \sum_{i=1}^{N} \prod_{j=1}^{3+3*\mathrm{nrb}} \boldsymbol{N}^T(\mathrm{var\ j})\boldsymbol{F1}^i_{\mathrm{var\ j}} + \prod_{j=1}^{3+3*\mathrm{nrb}} \boldsymbol{N}^T(\mathrm{var\ j})\boldsymbol{R}^i_{\mathrm{var\ j}}, \tag{5.11}$$

where N is the number of terms already for the PGD solution. This Eq. (5.11) is analogous for substep 2, with just changing $\boldsymbol{u}_{n+1/2}$ by $\boldsymbol{u}_{n+1}$, and $\boldsymbol{F1}^i_{\mathrm{var\ j}}$ by $\boldsymbol{F2}^i_{\mathrm{var\ j}}$.

Finally, Eq. (5.5) is solved in the code written below using the following notation:

$$\begin{aligned}
&\prod_{j=1}^{3+3*\mathrm{nrb}} \boldsymbol{R}^*_j \left[\left[\frac{16}{\Delta t^2}\right]\boldsymbol{M_m} + \boldsymbol{K_m}\right] \prod_{j=1}^{3+3*\mathrm{nrb}} \boldsymbol{R}_j \\
&\quad = \prod_{j=1}^{3+3*\mathrm{nrb}} \boldsymbol{R}^*_j \cdot \left[\boldsymbol{F}_{n+1/2} + \left[\frac{16}{\Delta t^2}\right]\boldsymbol{M_m}\cdot\boldsymbol{FR1_1} + \left[\frac{8}{\Delta t}\right]\boldsymbol{M_m}\cdot\boldsymbol{FR1_2} + \boldsymbol{M_m}\cdot\boldsymbol{FR1_3}\right. \\
&\qquad\qquad \left. - \sum_{i=1}^{N}\left[\left[\frac{16}{\Delta t^2}\right]\boldsymbol{M_m} + \boldsymbol{K_m}\right] \prod_{j=1}^{3+3*\mathrm{nrb}} \boldsymbol{F1}^i_j\right]
\end{aligned} \tag{5.12}$$

where $\prod_{j=1}^{3+3*\mathrm{nrb}} \boldsymbol{R}^*_j$ represents the weight function, that within the fixed-point strategy takes a different form depending on the particular iteration. Thus if, for instance, we are computing along the k-th coordinate, assuming the others directions to be known, we have

$$\prod_{j=1}^{3+3*\mathrm{nrb}} \boldsymbol{R}^*_j = \prod_{j=1, j\neq k}^{3+3*\mathrm{nrb}} \boldsymbol{R}_j \cdot \boldsymbol{R}^*_k. \tag{5.13}$$

These variables can be readily identified in the routine `enrichment_substep1` by taking into account the following notation,

$$\begin{aligned}
&\texttt{matrix1} = \left[\frac{16}{\Delta t^2}\right]\boldsymbol{M_m},\\
&\texttt{matrix2} = \boldsymbol{K_m},\\
&\boldsymbol{V} = \boldsymbol{F}_{n+1/2},\\
&\texttt{value1} = \left[\frac{16}{\Delta t^2}\right]\boldsymbol{M_m}\cdot\boldsymbol{FR1_1},\\
&\texttt{value2} = \left[\frac{8}{\Delta t}\right]\boldsymbol{M_m}\cdot\boldsymbol{FR1_2},\\
&\texttt{value3} = \boldsymbol{M_m}\cdot\boldsymbol{FR1_3},\\
&\boldsymbol{FV} = \boldsymbol{F1}.
\end{aligned}$$

For Eq. (5.6), the implemented routine `enrichment_substep2` computes

$$\prod_{j=1}^{3+3*\mathtt{nrb}} \boldsymbol{R}_j^* \left[\left[\frac{9}{\Delta t^2}\right]\boldsymbol{M_m} + \boldsymbol{K_m}\right] \prod_{j=1}^{3+3*\mathtt{nrb}} \boldsymbol{R}_j$$
$$= \prod_{j=1}^{3+3*\mathtt{nrb}} \boldsymbol{R}_j^* \cdot \left[\boldsymbol{F}_{n+1} - \left[\frac{19}{\Delta t^2}\right]\boldsymbol{M_m}\cdot\boldsymbol{FR2_1} - \left[\frac{5}{\Delta t}\right]\boldsymbol{M_m}\cdot\boldsymbol{FR2_2} + \left[\frac{28}{\Delta t^2}\right]\boldsymbol{M_m}\cdot\boldsymbol{FR2_3}\right.$$
$$\left. - \sum_{i=1}^{N}\left[\left[\frac{16}{\Delta t^2}\right]\boldsymbol{M_m} + \boldsymbol{K_m}\right]\prod_{j=1}^{3+3*\mathtt{nrb}} \boldsymbol{F1}_j^i\right].$$

In turn, $\prod_{j=1}^{3+3*nrb} \boldsymbol{R}_j^*$ is defined in Eq. (5.13). The variables can be identified as,

$$\mathtt{matrix1} = \left[\frac{9}{\Delta t^2}\right]\boldsymbol{M_m},$$
$$\mathtt{matrix2} = \boldsymbol{K_m},$$
$$\boldsymbol{V} = \boldsymbol{F}_{n+1},$$
$$\mathtt{value1} = \left[\frac{19}{\Delta t^2}\right]\boldsymbol{M_m}\cdot\boldsymbol{FR2_1},$$
$$\mathtt{value2} = \left[\frac{5}{\Delta t}\right]\boldsymbol{M_m}\cdot\boldsymbol{FR2_2},$$
$$\mathtt{value3} = \left[\frac{28}{\Delta t^2}\right]\boldsymbol{M_m}\cdot\boldsymbol{FR2_3},$$
$$\boldsymbol{FV} = \boldsymbol{F2}.$$

In next section the detailed code implmenting this strategy is provided.

5.4 Matlab Code

As always, the code begins by the `main.m` file, which is reproduced below. In this case, a series of previous simulations are needed so as to construct the POD basis referred to in Eq. (5.7). These simulations were carried out by us with the help of the commercial software Abaqus, although the reader can use his or her preferred code to do it. Once these simulations are done, and the POD modes computed, they are stored in memory by means of the instruction `WS = load('WorkSpaceBeam_REF.mat','Vreal');`, see below.

The quality of the final results will obviously depend on the similarity of this POD basis to the problem being simulated. In general, our experience indicates that with good basis, the number of POD modes necessary for a good energy conservation (i.e., avoidance of numerical dissipation) tends to be on the order of 6–8 modes.

```
%
%                       PGD Code for Dynamic Problem
%                    D. González, I. Alfaro, E. Cueto
%                        Universidad de Zaragoza
%                          AMB-I3A Dec 2015
%
clear all; close all; clc;
%
% VARIABLES
%
global E nu coords tet
Modulus = 10000.0; % Force modulus.
cooru = linspace(-5E1,5E1,300); % Discretization for displacement field.
coorv = linspace(-5E+2,5E+2,300);% Discretization for velocity field.
coora = linspace(-1E+3,1E+3,300); % Discretization for acceleration field.
deltat = 0.00125; coort = 0:deltat:2; % Time discretization.
TOL = 1.0E-04; % Tolerance.
num_max_iter = 15; % # of summands of the approach.
E = 2E11; nu = 0.3; Rho = 2.5E+04; % Material.
deltatA = 0.00125; % Time step for Reference problem - Abaqus' result.
NodeR = 6; NodeC = 104; % Reference nodes to compare PGD solution.
nrb = 1; % Number of directions on reduced basis.
% The PGD solution depends on: Space, Load, load parameter and
% displacement-velocity-acceleration for each reduced POD basis
nv = 3 + 3*nrb; % # of parameters (or variables) for the PGD solution.
%
% GEOMETRY
%
coords = load('gcoordBeam.dat'); % Nodal coordinates.
tet = load('conecBeam.dat'); % Connectivity list.
Ind = 1:size(coords,1); % List of nodes.
bcnode = Ind(coords(:,1)==min(coords(:,1))); % Boundary: Fix left side.
IndBcnode = sort([3*(bcnode-1)+1 3*(bcnode-1)+2 3*bcnode]); % D.o.f. BCs.
dof = setxor(IndBcnode,1:numel(coords))'; % D.o.f. Free nodes.
% Make use of triangulation MatLab function to obtain boundary surface.
TR = triangulation(tet,coords);
[tf] = freeBoundary(TR); % Dependent of 3D geometry of the boundary.
[tri,coors] = freeBoundary(TR); % Independent triangulation of boundary.
IndS = 1:size(coors,1); ncoors = numel(IndS);
%
% STIFNESS AND MASS MATRICES COMPUTATION
%
[r1,r2] = fem3D; r2 = Rho.*r2; % Space: Stiffness: r1, Mass: r2.
[z1,vu] = elemstiff(cooru); % Displacements.
[w1,vv] = elemstiff(coorv); % Velocities.
[s1,va] = elemstiff(coora); % Accelerations.
vu = repmat(vu,1,ncoors); % Reshape to construct source in separated form.
vv = repmat(vv,1,ncoors);
va = repmat(va,1,ncoors);
%
% SOURCE
%
coorp = 1:size(coors,1); % We consider each possible load position
                                    % as a different load case.
p1 = elemstiff(coorp); % Mass matrix for load parameter: 1st vargout.
% Identifying local nodes of the loaded surface on the global connectivity.
% To obtain that: coords(IndL,:)-coors = zeros(nn2,1).
[trash,trash2,xj] = intersect(IndS,tri(:)); % TRI Local connectivity.
IndL = tf(xj); % TF Global connectivity of the loaded surface (Top).
DOFLoaded = 3*IndL; % Consider vertical load on the top of the beam.
vx = zeros(numel(coords),ncoors);
vx(DOFLoaded,:) = eye(ncoors); % Space terms for the source.
vp = eye(ncoors); vp = p1*vp; % Load terms for the source.
%
% ACTIVATION OF LOAD PARAMETER
%
```

```
coorac = [1 2]; % Value to activate the load. Two possibilities 1-No 2-Yes.
pa1 = elemstiff(coorac); % Mass matrix for load activation: 1st vargout.
vpa = pa1*repmat([0; -Modulus],1,ncoors); % Activation terms for the source
%
% LOADING P.O.D. DATA TO CONSTRUCT REDUCED BASIS
%
WS = load('WorkSpaceBeam_REF.mat','Vreal');
Vreal = WS.Vreal; % Loading Displacement field of the reference solution.
%
% APPLY P.O.D. TECHNIQUE TO OBTAIN REDUCED BASIS
%
Q = Vreal(dof,:)*Vreal(dof,:)'; [A,lam] = eigs(Q,[],nrb);
%
% ALLOCATION OF MATRICES AND VECTORS FOR EACH TIME INTEGRATION STEP (1,2)
%
K1 = cell(nv,1); M1 = K1; V1 = K1; Fv1 = K1; FR1 = K1; coor1 = cell(nv,1);
K2 = cell(nv,1); M2 = K2; V2 = K2; Fv2 = K2; FR2 = K2; coor2 = cell(nv,1);
%
% SPACE MATRICES
%
K1{1} = r1; % Stiffness matrix for SubStep 1.
M1{1} = r2; % Mass matrix for SubStep 1.
V1{1} = vx; % Space term for the source at the SubStep 1.
K2{1} = r1; M2{1} = r2; V2{1} = vx; % SubStep 2.
%
% LOAD MATRICES
%
K1{nv} = p1; % Stiffness matrix contribution of load for SubStep 1.
M1{nv} = p1; % Mass matrix for SubStep 1.
V1{nv} = vp; % Load term for the source at the SubStep 1.
coor1{nv} = coorp; % Load Discretization for SubStep 1.
K2{nv} = p1; M2{nv} = p1; V2{nv} = vp; coor2{nv} = coorp; % SubStep 2.
%
% MATRICES RELATED TO ACTIVATION PARAMETER
%
K1{nv-1} = pa1; % Stiffness contribution of act.param. for SubStep 1.
M1{nv-1} = pa1; % Mass matrix for SubStep 1.
V1{nv-1} = vpa; % Activation parameter term for the source, SubStep 1.
coor1{nv-1} = coorac; % Activation parameter discretization for SubStep 1.
K2{nv-1} = pa1; M2{nv-1} = pa1; V2{nv-1} = vpa; coor2{nv-1} = coorac; % S2.
%
% REDUCED BASIS MATRICES
%
for i1=2:3:nv-2
    %
    % SUBSTEP 1
    %
    K1{i1} = z1; M1{i1} = z1; V1{i1} = vu; coor1{i1} = cooru; % U
    K1{i1+1} = w1; M1{i1+1} = w1; V1{i1+1} = vv; coor1{i1+1} = coorv; % V
    K1{i1+2} = s1; M1{i1+2} = s1; V1{i1+2} = va; coor1{i1+2} = coora; % A
    %
    % SUBSTEP 2
    %
    K2{i1} = z1; M2{i1} = z1; V2{i1} = vu; coor2{i1} = cooru; % U
    K2{i1+1} = w1; M2{i1+1} = w1; V2{i1+1} = vv; coor2{i1+1} = coorv; % V
    K2{i1+2} = z1; M2{i1+2} = z1; V2{i1+2} = vu; coor2{i1+2} = cooru; % U/2
end
%
% INICIALIZATING PGD SOLUTION
%
for i1=1:nv
    Fv1{i1} = 0.0.*V1{i1}(:,1); % PGD vectors for SubStep 1.
    Fv2{i1} = 0.0.*V2{i1}(:,1); % PGD vectors for SubStep 2.
end
%
% BOUNDARY CONDITIONS
%
```

```
Free1 = cell(nv,1); Free2 = cell(nv,1);
Free1{1} = dof; Free2{1} = dof; % Free DOF for Space.
for i1=2:nv % No BCs for rest of parameters (variables).
    Free1{i1} = 1:numel(coor1{i1});
    Free2{i1} = 1:numel(coor2{i1});
end
%
% Un, Vn, An ... IN SEPARATED FORM FOR INTEGRATION SCHEME
%
% We have 3*nen terms in the source related to Un, Vn and An in SubStep 1
% and Vn, Un+1/2 and Un for SubStep 2. Following the sort of the variables
% in the PGD solution for SubStep 1, U1_{n+1/2}(Var_1, Var_2,...,
% Var_{nv-1}, Var_{nv}), where Var_1=Spatial Coordinates, Var_{nv-1} =
% Activation Parameter, Var_{nv} = Loads, and Var_(2:3:nv_1) = U_n
% (Displacement in time n), Var_(3:3:nv_1) = V_n (Velocity in time n) and
% Var_(4:3:nv_1) = A_n (Acceleration in time n)
% For the PGD solution for SubStep 2, U2_{n+1/2}(Var_1, Var_2,...,
% Var_{nv-1}, Var_{nv}), where Var_1=Spatial Coordinates, Var_{nv-1} =
% Activation Parameter, Var_{nv} = Loads, and Var_(2:3:nv_1) = U_n
% (Displacement in time n), Var_(3:3:nv_1) = V_n (Velocity in time n) and
% Var_(4:3:nv_1) = U_{n+1/2} (Displacement in time n + 1/2).
%
% IMPORTANT: To obtain U_n and V_n variables in SubStep 1 (for instance) in
% separated form we consider that:
% U_n = 1_{space} U_n 1_{velocity} 1_{acceleration} 1_{activation} 1_{load}
% V_n = 1_{space} 1_{displac.} V_n 1_{acceleration} 1_{activation} 1_{load}
FR1{1} = zeros(size(Fv1{1},1),nv-3); FR2{1} = zeros(size(Fv2{1},1),nv-3);
for i1=2:nv
    FR1{i1} = ones(size(Fv1{i1},1),nv-3);
    FR2{i1} = ones(size(Fv2{i1},1),nv-3);
end
for i1=1:3 % 3 terms per # reduced basis.
    FR1{1}(dof,i1:3:end) = A; % Projection space onto Reduced basis.
    FR2{1}(dof,i1:3:end) = A;
end
for i1=2:nv-2
    FR1{i1}(:,i1-1) = coor1{i1}'; % U_n, V_n, A_n.
    FR2{i1}(:,i1-1) = coor2{i1}'; % U_n, V_n, U_{n+1/2}.
end
%
% ENRICHMENT OF THE APPROXIMATION: SUBSTEP 1
%
num_iter1 = 0; Error_iter = 1.0; iter = zeros(1); Aprt = 0;
while Error_iter>TOL && num_iter1<num_max_iter
    num_iter1 = num_iter1 + 1; R0 = cell(nv,1);
    for i1=1:nv
        R0{i1} = ones(size(Fv1{i1},1),1); % Initial guess for R, S, ...
    end;
    R0{1}(IndBcnode) = 0; % We impose initial guess for spacial coordinates
    %
    % ENRICHMENT STEP
    %
    [R,iter(num_iter1)] = enrichment_substep1(K1,M1,V1,num_iter1,Fv1,R0,...
        FR1,Free1,deltat);
    for i1=1:nv, Fv1{i1}(:,num_iter1) = R{i1}; end % R is valid, add it.
    %
    % STOPPING CRITERION
    %
    Error_iter = 1.0;
    for i1=1:nv
        Error_iter = Error_iter.*norm(Fv1{i1}(:,num_iter1));
    end
    Aprt = max(Aprt,sqrt(Error_iter));
    if num_iter1>nrb, Error_iter = sqrt(Error_iter)/Aprt; end
    fprintf(1,'SubStep_1:_%dst_summand_in_%d_',num_iter1,iter(num_iter1));
    fprintf(1,'iterations_with_a_weight_of_%f\n',sqrt(Error_iter));
end
```

```
fprintf(1,'\n');
%
% ENRICHMENT OF THE APPROXIMATION: SUBSTEP 2
%
num_iter2 = 0; Error_iter = 1.0; iter = zeros(1); Aprt = 0;
while Error_iter>TOL && num_iter2<num_max_iter
    num_iter2 = num_iter2 + 1; R0 = cell(nv,1);
    for i1=1:nv
        R0{i1} = rand(size(Fv2{i1},1),1); % Initial guess for R, S, ...
    end
    R0{1}(IndBcnode) = 0; % We impose initial guess for spacial coordinates
    %
    % ENRICHMENT STEP
    %
    [R,iter(num_iter2)] = enrichment_substep2(K2,M2,V2,num_iter2,Fv2,R0,...
        FR2,Free2,deltat);
    for i1=1:nv, Fv2{i1}(:,num_iter2) = R{i1}; end % R is valid, add it.
    %
    % STOPPING CRITERION
    %
    Error_iter = 1.0;
    for i1=1:nv
        Error_iter = Error_iter.*norm(Fv2{i1}(:,num_iter2));
    end
    Aprt = max(Aprt,sqrt(Error_iter));
    if num_iter2>nrb, Error_iter = sqrt(Error_iter)/Aprt; end
    fprintf(1,'SubStep_2:_%dst_summand_in_%d_',num_iter2,iter(num_iter2));
    fprintf(1,'iterations_with_a_weight_of_%f\n',sqrt(Error_iter));
end
fprintf(1,'PGD_Process_exited_normally\n\n');
save('WorkSpacePGD_Dynamic.mat');
%
% POST-PROCESSING
%
figure; % Plotting reference solution for the node NodeR
plot(0:deltatA:deltatA*(size(Vreal,2)-1),Vreal(3*(NodeR-1)+3,:),...
    'b-','LineWidth',2.5); % Reference solution for vertical displacement
%
% ALLOCATE MEMORY FOR SUBSTEP SOLUTION VECTORS
%
Mv = cell(nv,1); % Nodal values for each parameter
% We need to compute SUBSTEP 1: MΰA_{n+1/2} + KΰU_{n+1/2} = F_{n+1/2}
% and A_{n+1/2} = 4*(V_{n+1/2}-V_n)/Deltat - A_n
% and V_{n+1/2} = 4*(U_{n+1/2}-U_n)/Deltat - V_n
Disp2 = zeros(nrb,numel(coort)); % Allocate memory for U_{n+1/2}
Vel2 = zeros(nrb,numel(coort)); % Allocate memory for V_{n+1/2}
Acel2 = zeros(nrb,numel(coort)); % Allocate memory for A_{n+1/2}
% We need to compute SUBSTEP 2: MΰA_{n+1} + KΰU_{n+1} = F_{n+1}
% and A_{n+1} = V_n/Deltat - 4*V_{n+1/2}/Deltat + 3*V_{n+1}/Deltat
% and V_{n+1} = U_n/Deltat - 4*U_{n+1/2}/Deltat + 3*U_{n+1}/Deltat
Disp = zeros(nrb,numel(coort)); % Allocate memory for U_{n+1}
Vel = zeros(nrb,numel(coort)); % Allocate memory for V_{n+1}
Acel = zeros(nrb,numel(coort));% Allocate memory for A_{n+1}
% We compute U_n,V_n,... onto reduced basis. We back to real Space
RealDisp2 = zeros(numel(coords),numel(coort));
RealDisp = zeros(numel(coords),numel(coort));
%
% INITIAL VALUES FOR DISPLACEMENT FIELD
%
Ai = pinv(A); % Pseudo-inverse P.O.D. matrix.
% 1st and 2nd displacement field for t = 0 and t = deltat from reference
% solution to start the simulation.
RealDisp(dof,1:2) = Vreal(dof,1:2).*deltat./deltatA;
% We project these first two displacements onto reduced basis.
Disp(:,1:2) = Ai*RealDisp(dof,1:2).*deltat./deltatA;
% We interpolate the reference solution for [1:2]+1/2 steps.
Disp2(:,1) = Ai*((Vreal(dof,1) + Vreal(dof,2))/2).*deltat./deltatA;
```

```
Disp2(:,2) = Ai*((Vreal(dof,2) + Vreal(dof,3))/2).*deltat./deltatA;
%
% REAL-TIME LOOP FOR TIME INTEGRATION
%
for k2=3:numel(coort)
    %
    % APPLY SUBSTEP 1
    %
    for i1=2:3:nv-2
        Mv{i1} = evaluate_shpfunc(coor1{i1},Disp((i1+1)/3,k2-1));
        Mv{i1+1} = evaluate_shpfunc(coor1{i1+1},Vel((i1+1)/3,k2-1));
        Mv{i1+2} = evaluate_shpfunc(coor1{i1+2},Acel((i1+1)/3,k2-1));
    end
    Mv{nv} = evaluate_shpfunc(coor1{nv},NodeC); % Load case NodeC.
    if coort(k2)-deltat/2<0.25
        Mv{nv-1} = [0; 1]; % Value of the shape function for act. parameter
    elseif coort(k2)-deltat/2>=0.5 % Time when the load vanishes
        Mv{nv-1} = [1; 0]; % Value of the shape function for act. parameter
    else % Ramp case
        Mv{nv-1}(2) = (0.5-(coort(k2)-deltat/2))/0.25;
        Mv{nv-1}(1) = 1-Mv{nv-1}(2); % Value of the shape function.
    end
    %
    % COMPUTE PGD SUBSTEP 1 SOLUTION
    %
    for k1=1:num_iter1
        value1 = Fv1{1}(dof,k1);
        for j1=2:nv, value1 = value1.*(Mv{j1}'*Fv1{j1}(:,k1)); end
        RealDisp2(dof,k2) = RealDisp2(dof,k2) + value1;
    end
    %
    % TIME SCHEME SUBSTEP 1
    %
    % MůA_{n+1/2} + KůU_{n+1/2} = F_{n+1/2}
    % and A_{n+1/2} = 4*(V_{n+1/2}-V_n)/Deltat - A_n
    % and V_{n+1/2} = 4*(U_{n+1/2}-U_n)/Deltat - V_n
    Disp2(:,k2) = Ai*RealDisp2(dof,k2); % Project onto reduced basis
    Vel2(:,k2) = 4*(Disp2(:,k2)-Disp(:,k2-1))/deltat - Vel(:,k2-1);
    Acel2(:,k2) = 4*(Vel2(:,k2)-Vel(:,k2-1))/deltat - Acel(:,k2-1);
    %
    % APPLY SUBSTEP 2
    %
    for i1=2:3:nv-2
        Mv{i1+2} = evaluate_shpfunc(coor2{i1+2},Disp2((i1+1)/3,k2));
    end
    Mv{nv} = evaluate_shpfunc(coor1{nv},NodeC); % Load case NodeC.
    if coort(k2)<0.25
        Mv{nv-1} = [0; 1]; % Value of the shape function for act. parameter
    elseif coort(k2)>=0.5 % Time when the load vanishes
        Mv{nv-1} = [1; 0]; % Value of the shape function for act. parameter
    else % Ramp case
        Mv{nv-1}(2) = (0.5-coort(k2))/0.25;
        Mv{nv-1}(1) = 1-Mv{nv-1}(2); % Value of the shape function.
    end
    %
    % COMPUTE PGD SUBSTEP 2 SOLUTION
    %
    for k1=1:num_iter2
        value2 = Fv2{1}(dof,k1);
        for j1=2:nv, value2 = value2.*(Mv{j1}'*Fv2{j1}(:,k1)); end
        RealDisp(dof,k2) = RealDisp(dof,k2) + value2;
    end
    %
    % TIME SCHEME SUBSTEP 2
    %
    % MůA_{n+1} + KůU_{n+1} = F_{n+1}
    % and A_{n+1} = V_n/Deltat - 4*V_{n+1/2}/Deltat + 3*V_{n+1}/Deltat
```

```
    % and V_{n+1} = U_n/Deltat - 4*U_{n+1/2}/Deltat + 3*U_{n+1}/Deltat
    Disp(:,k2) = Ai*RealDisp(dof,k2); % Project onto reduced basis
    Vel(:,k2) = Disp(:,k2-1)/deltat - 4*Disp2(:,k2)/deltat + ...
        3*Disp(:,k2)/deltat;
    Acel(:,k2) = Vel(:,k2-1)/deltat - 4*Vel2(:,k2)/deltat + ...
        3*Vel(:,k2)/deltat;
end
%
% PLOT PGD FINAL SOLUTION AND COMPARE WITH THE REFERENCE SOLUTION
%
hold on; plot(coort,RealDisp(3*(NodeR-1)+3,:),'m--','LineWidth',2.5);
save('WorkSpacePGD_Dynamic.mat');
fprintf(1,'\n#####ăEnd_of_simulation_#####\n\n');
```

This just presented code reads nodal coordinates and connectivity list from files `gcoordBeam.dat` and `conecBeam.dat`, respectively. If, after the execution of the program, one types

» `trisurf(tri,coors(:,1),coors(:,2),coors(:,3));`

» `axis equal`

in Matlab's command line, a plot of the finite element model of the beam is obtained, see Fig. 5.2.

As in previous examples, the code calls to the subroutine `elemstiff.m`, which is reproduced below.

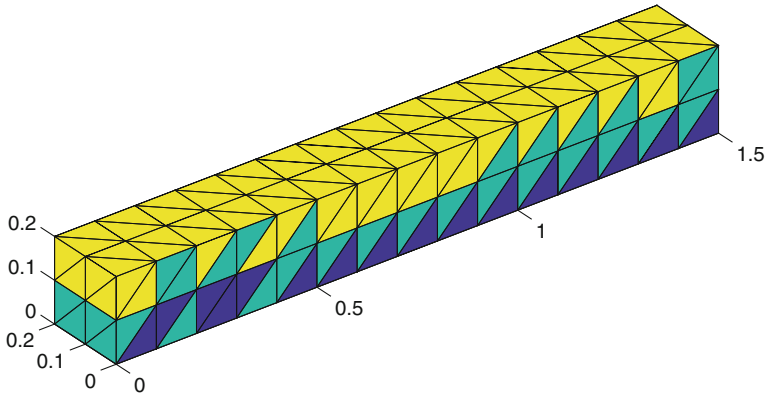

Fig. 5.2 Finite element mesh for the beam dynamics problem. The beam is assumed to be encastred at $y = 0$

```
function [p1,p2] = elemstiff(coor)
% function [p1,p2] = elemstiff(coor)
% For variable X compute p1=\int N N, p2=\int N
% Universidad de Zaragoza - 2015
nen = numel(coor);
%
% ALLOCATE MEMORY
%
p1 = zeros(nen); p2 = zeros(nen,1);
X = coor(1:nen-1)'; Y = coor(2:nen)'; % Coordinates of elements.
L = Y - X; % Longitude of each  element for parametrized variable.
sg = [-1.0/sqrt(3.0) 1.0/sqrt(3.0)]; wg = ones(2,1); % Gauss points.
npg = numel(sg);
for i1=1:nen-1
    c = zeros(1,npg); c(1,:) = 0.5.*(1.0-sg).*X(i1) + 0.5.*(1.0+sg).*Y(i1);
    N = zeros(nen,npg);
    N(i1+1,:) = (c(1,:)-X(i1))./L(i1); N(i1,:) = (Y(i1)-c(1,:))./L(i1);
    for j1=1:npg
        p1 = p1 + N(:,j1)*N(:,j1)'*0.5.*wg(j1).*L(i1); % \int N N.
        p2 = p2 + N(:,j1).*0.5.*wg(j1).*L(i1); % \int N.
    end
end
return
```

Our particular implementation of the method makes use of the energy and momentum conserving algorithm by Bathe [15]. This algorithm makes use of a predictor-corrector algorithm, whose first sub-step is included in routine `enrichment_substep1`, reproduced here:

```
function [R,iter] = enrichment_substep1(K,M,V,num_iter,FV,R,FR,CC,deltat)
% [R,iter] = enrichment_substep1(K,M,V,num_iter,FV,R,FR,CC,deltat)
% Compute the new R, S, functions to enrich the PGD solution for SubStep 1
% in Dynamic problems.
% Universidad de Zaragoza - 2015
iter = 1; TOL = 1.0E-4; error = 1.0; % Initializating values.
nv = size(FV,1); % # Number of variables for the PGD.
mxit = 11; % # of possible iterations for the fixed point algorithm.
%
% FIXED POINT ALGORITHM
%
while abs(error)>TOL
    Raux = R; % Updating R(S) last values.
    for i1=1:nv
        %
        % MATRIX COMPUTATION
        %
        matrix1 = 16.0/deltat/deltat; % Constant of mass contribution.
        matrix2 = 1.0; % Constant of stiffness contribution.
        for j1=1:nv
            if j1~=i1
                matrix1 = matrix1.*(R{j1}'*M{j1}*R{j1});
                matrix2 = matrix2.*(R{j1}'*K{j1}*R{j1});
            end
        end
        matrix = M{i1}.*matrix1 + K{i1}.*matrix2;
        %
        % SOURCE COMPUTATION
        %
        source = 0.0;
        for k1=1:size(V{1},2) % Loop over number of functions of the source
            sourceval = 1.0;
            for j1=1:nv
                if j1~=i1
                    sourceval = sourceval.*(R{j1}'*V{j1}(:,k1));
```

```
                end
            end
            source = source + sourceval.*V{i1}(:,k1);
        end
        for k1=1:3:(nv-3)
            value1 = 16/deltat/deltat;% Constant value for U_n contribution
            value2 = 8/deltat; % Constant value for V_n contribution.
            value3 = 1.0; % Constant value for A_n contribution.
            for j1=1:nv
                if j1~=i1
                    value1 = value1.*(R{j1}'*M{j1}*FR{j1}(:,k1));
                    value2 = value2.*(R{j1}'*M{j1}*FR{j1}(:,k1+1));
                    value3 = value3.*(R{j1}'*M{j1}*FR{j1}(:,k1+2));
                end
            end
            source = source + value1.*(M{i1}*FR{i1}(:,k1)) + ...
                value2.*(M{i1}*FR{i1}(:,k1+1)) + ...
                value3.*(M{i1}*FR{i1}(:,k1+2));
        end
        %
        % CONTRIBUTION TO SOURCE OF KNOWNING SOLUTION
        %
        for i2=1:num_iter-1
            value1 = 16/deltat/deltat; % Constant of mass contribution.
            value2 = 1.0; % Constant of stiffness contribution.
            for j1=1:nv
                if j1~=i1
                    value1 = value1.*(R{j1}'*M{j1}*FV{j1}(:,i2));
                    value2 = value2.*(R{j1}'*K{j1}*FV{j1}(:,i2));
                end
            end
            source = source - (M{i1}*FV{i1}(:,i2)).*value1 - ...
                (K{i1}*FV{i1}(:,i2)).*value2;
        end
        %
        % SOLVE THE R{i1} VARIABLE
        %
        R{i1}(CC{i1}) = matrix(CC{i1},CC{i1})\source(CC{i1});
    end
    %
    % COMPUTING STOP CRITERIA
    %
    error = 0;
    for j1=1:nv
        error = error + norm(Raux{j1}-R{j1});
    end
    error = sqrt(error);
    iter = iter + 1;
    if iter==mxit, error = 0.0; end
end
return
```

In turn, we reproduce here the second sub-step of the time integration algorithm proposed by Bathe [15]. Remember that you can use in this framework your favorite time integration scheme (Newmark, HHT, …)

```
function [R,iter] = enrichment_substep2(K,M,V,num_iter,FV,R,FR,CC,deltat)
% [R,iter] = enrichment_substep2(K,M,V,num_iter,FV,R,FR,CC,deltat)
% Compute the new R, S, functions to enrich the PGD solution for SubStep 2
% in Dynamic problems.
% Universidad de Zaragoza - 2015
iter = 1; TOL = 1.0E-4; error = 1.0; % Inicializating values.
nv = size(FV,1); % # Number of variables for the PGD.
mxit = 11; % # of possible iterations for the fixed point algorithm.
%
```

```
% FIXED POINT ALGORITHM
%
while abs(error)>TOL
    Raux = R; % Updating R(S) last values.
    for i1=1:nv
        %
        % MATRIX COMPUTATION
        %
        matrix1 = 9.0/deltat/deltat; % Constant of mass contribution.
        matrix2 = 1.0; % Constant of stiffness contribution.
        for j1=1:nv
            if j1~=i1
                matrix1 = matrix1.*(R{j1}'*M{j1}*R{j1});
                matrix2 = matrix2.*(R{j1}'*K{j1}*R{j1});
            end
        end
        matrix = M{i1}.*matrix1 + K{i1}.*matrix2;
        %
        % SOURCE COMPUTATION
        %
        source = 0.0;
        for k1=1:size(V{1},2) % Loop over number of functions of the source
            sourceval = 1.0;
            for j1=1:nv
                if j1~=i1
                    sourceval = sourceval.*(R{j1}'*V{j1}(:,k1));
                end
            end
            source = source + sourceval.*V{i1}(:,k1);
        end
        for k1=1:3:(nv-3)
            value1 = 19/deltat/deltat;% Constant value for U_n contribution
            value2 = 5/deltat; % Constant value for V_n contribution.
            value3 = 28/deltat/deltat; % Constant value for U_{n+1/2}.
            for j1=1:nv
                if j1~=i1
                    value1 = value1.*(R{j1}'*M{j1}*FR{j1}(:,k1));
                    value2 = value2.*(R{j1}'*M{j1}*FR{j1}(:,k1+1));
                    value3 = value3.*(R{j1}'*M{j1}*FR{j1}(:,k1+2));
                end
            end
            source = source - value1.*(M{i1}*FR{i1}(:,k1)) - ...
                value2.*(M{i1}*FR{i1}(:,k1+1)) +...
                value3.*(M{i1}*FR{i1}(:,k1+2));
        end
        %
        % CONTRIBUTION TO SOURCE OF KNOWNING SOLUTION
        %
        for i2=1:num_iter-1
            value1 = 9/deltat/deltat; % Constant of mass contribution.
            value2 = 1.0; % Constant of stiffness contribution.
            for j1=1:nv
                if j1~=i1
                    value1 = value1.*(R{j1}'*M{j1}*FV{j1}(:,i2));
                    value2 = value2.*(R{j1}'*K{j1}*FV{j1}(:,i2));
                end
            end
            source = source - (M{i1}*FV{i1}(:,i2)).*value1 - ...
                (K{i1}*FV{i1}(:,i2)).*value2;
        end
        %
        % SOLVE THE R{i1} VARIABLE
        %
        R{i1}(CC{i1}) = matrix(CC{i1},CC{i1})\source(CC{i1});
    end
    %
    % COMPUTING STOP CRITERIA
```

```
    %
    error = 0;
    for j1=1:nv
        error = error + norm(Raux{j1}-R{j1});
    end
    error = sqrt(error);
    iter = iter + 1;
    if iter==mxit, error = 0.0; end
end
return
```

In subroutine `fem3D.m` we accomplish traditional FE computations regarding stiffness matrix, etc., for linear tetrahedrons.

```
function [A,N] = fem3D
% function [A,N] = fem3D
% Finite Element Method for Tetrahedron. The vargout are the Stifness and
% Mass matrices for spacial variables.
% Universidad de Zaragoza - 2015
global E nu coords tet

dof = 3; % Degrees of Freedom per node.
numNodos = size(coords,1); % # of nodes.
numTet = size(tet,1); % # of 3D elements.
A = zeros(dof*numNodos); N = zeros(dof*numNodos); % Allocate memory.
%
% MATERIAL MATRIX
%
D = zeros(6); cte = E*(1-nu)/(1+nu)/(1-2*nu);
D(1) = cte; D(8) = D(1); D(15) = D(1);
D(2) = cte*nu/(1-nu); D(3) = D(2); D(7) = D(2); D(9) = D(2); D(13) = D(2);
D(14) = D(2); D(22) = cte*(1-2*nu)/2/(1-nu); D(29) = D(22); D(36) = D(22);
%
% INTEGRATION POINTS
%
sg = zeros(3,4); wg = 1./24.*ones(4,1); nph = numel(wg);
a = (5.0 - sqrt(5))/20.0; b= (5.0 + 3.0*sqrt(5))/20.0;
sg(:,1) = [a; a; a]; sg(:,2) = [a; a; b];
sg(:,3) = [a; b; a];  sg(:,4) = [b; a; a];
%
% FINITE ELEMENT LOOP
%
for i1=1:numTet
    elnodes = tet(i1,:);
    xcoord = coords(elnodes,:);
    K = zeros(dof*4); KK = zeros(dof*4);
    %
    % JACOBIAN
    %
    v1 = xcoord(1,:)-xcoord(2,:); v2 = xcoord(2,:)-xcoord(3,:);
    v3 = xcoord(3,:)-xcoord(4,:);
    jcob = abs(det([v1;v2;v3]));
    %
    % SHAPE FUNCTION CONSTANTS
    %
    a1 = det([xcoord(2,:); xcoord(3,:); xcoord(4,:)]);
    a2 = -det([xcoord(1,:); xcoord(3,:); xcoord(4,:)]);
    a3 = det([xcoord(1,:); xcoord(2,:); xcoord(4,:)]);
    a4 = -det([xcoord(1,:); xcoord(2,:); xcoord(3,:)]);
    b1 = -det([1 xcoord(2,2:end); 1 xcoord(3,2:end); 1 xcoord(4,2:end)]);
    b2 = det([1 xcoord(1,2:end); 1 xcoord(3,2:end); 1 xcoord(4,2:end)]);
    b3 = -det([1 xcoord(1,2:end); 1 xcoord(2,2:end); 1 xcoord(4,2:end)]);
    b4 = det([1 xcoord(1,2:end); 1 xcoord(2,2:end); 1 xcoord(3,2:end)]);
    c1 = det([1 xcoord(2,1) xcoord(2,end); 1 xcoord(3,1) xcoord(3,end);...
        1 xcoord(4,1) xcoord(4,end)]);
    c2 = -det([1 xcoord(1,1) xcoord(1,end); 1 xcoord(3,1) xcoord(3,end);...
```

```
    1 xcoord(4,1) xcoord(4,end)]);
c3 = det([1 xcoord(1,1) xcoord(1,end); 1 xcoord(2,1) xcoord(2,end);...
    1 xcoord(4,1) xcoord(4,end)]);
c4 = -det([1 xcoord(1,1) xcoord(1,end); 1 xcoord(2,1) xcoord(2,end);...
    1 xcoord(3,1) xcoord(3,end)]);
d1 = -det([1 xcoord(2,1:end-1); 1 xcoord(3,1:end-1);...
    1 xcoord(4,1:end-1)]);
d2 = det([1 xcoord(1,1:end-1); 1 xcoord(3,1:end-1);...
    1 xcoord(4,1:end-1)]);
d3 = -det([1 xcoord(1,1:end-1); 1 xcoord(2,1:end-1);...
    1 xcoord(4,1:end-1)]);
d4 = det([1 xcoord(1,1:end-1); 1 xcoord(2,1:end-1);...
    1 xcoord(3,1:end-1)]);
%
% INTEGRATION POINTS LOOP
%
for j1=1:nph
    chi = sg(3*(j1-1)+1);
    eta = sg(3*(j1-1)+2);
    tau = sg(3*j1);
    %
    % GEOMETRY APPROACH
    %
    SHPa(4) = tau;
    SHPa(3) = eta; SHPa(2) = chi; SHPa(1) = 1.-chi-eta-tau;
    chiG = 0.0; etaG = 0.0; tauG = 0.0;
    for k1=1:4
        chiG = chiG + SHPa(k1)*xcoord(k1,1);
        etaG = etaG + SHPa(k1)*xcoord(k1,2);
        tauG = tauG + SHPa(k1)*xcoord(k1,3);
    end
    %
    % SHAPE FUNCTION COMPUTATION
    %
    SHP(1) = (a1 + b1*chiG + c1*etaG + d1*tauG)/jcob;
    dSHPx(1) = b1/jcob; dSHPy(1) = c1/jcob; dSHPz(1) = d1/jcob;
    SHP(2) = (a2 + b2*chiG + c2*etaG + d2*tauG)/jcob;
    dSHPx(2) = b2/jcob; dSHPy(2) = c2/jcob; dSHPz(2) = d2/jcob;
    SHP(3) = (a3 + b3*chiG + c3*etaG + d3*tauG)/jcob;
    dSHPx(3) = b3/jcob; dSHPy(3) = c3/jcob; dSHPz(3) = d3/jcob;
    SHP(4) = (a4 + b4*chiG + c4*etaG + d4*tauG)/jcob;
    dSHPx(4) = b4/jcob; dSHPy(4) = c4/jcob; dSHPz(4) = d4/jcob;
    %
    % B MATRIX COMPUTATION
    %
    B = [dSHPx(1) 0 0 dSHPx(2) 0 0 dSHPx(3) 0 0 dSHPx(4) 0 0; ...
        0 dSHPy(1) 0 0 dSHPy(2) 0 0 dSHPy(3) 0 0 dSHPy(4) 0;...
        0 0 dSHPz(1) 0 0 dSHPz(2) 0 0 dSHPz(3) 0 0 dSHPz(4);...
        dSHPy(1) dSHPx(1) 0 dSHPy(2) dSHPx(2) 0 dSHPy(3) dSHPx(3)...
        0 dSHPy(4) dSHPx(4) 0;...
        dSHPz(1) 0 dSHPx(1) dSHPz(2) 0 dSHPx(2) dSHPz(3) 0 dSHPx(3)...
        dSHPz(4) 0 dSHPx(4);...
        0 dSHPz(1) dSHPy(1) 0 dSHPz(2) dSHPy(2) 0 dSHPz(3) dSHPy(3)...
        0 dSHPz(4) dSHPy(4)];
    %
    % MASS MATRIX
    %
    M = [SHP(1) 0 0; 0 SHP(1) 0; 0 0 SHP(1); SHP(2) 0 0; 0 SHP(2) 0;...
        0 0 SHP(2); SHP(3) 0 0; 0 SHP(3) 0; 0 0 SHP(3); SHP(4) 0 0;...
        0 SHP(4) 0; 0 0 SHP(4)]';
    K = K + B'*D*B*jcob*wg(j1);  % Element Stiffness matrix.
    KK = KK + M'*M*jcob*wg(j1); % Element Mass matrix.
end
% System degrees of freedom associated with each element.
index = [3*elnodes-2;3*elnodes-1;3*elnodes];
index = reshape(index,1,4*dof);
% Assembling of the system stiffness matrix.
```

```
    A(index,index) = A(index,index) + K;
    N(index,index) = N(index,index) + KK;
end
return
```

Function `evaluate_shpfunc.m` computes finite element one-dimensional shape functions.

```
function S = evaluate_shpfunc(coor,cx)
% function S = evaluate_shpfunc(coor,cx)
% Compute approach value for cx coordinate respect coor points using 1D
% Shape functions
% Universidad de Zaragoza - 2015
tam = numel(coor); % # number of nodes.
TOL = 1.0E-8; % Tolerance.
S = zeros(tam,1); % Allocating memory.
idx = 0; % Index of elemnt where cx is.
for i1=1:numel(coor)-1 % # Elements Loop to find with in
    if coor(i1)-TOL<=cx && coor(i1+1)>=cx
        idx = i1;
        break;
    end
end
% A = find(coor(coor>=cx+TOL));
% idx = A(end); % Idx is element containing cx.
if idx~=tam && idx~=0
    L = coor(idx+1)-coor(idx); % Linear approximation.
    S(idx+1) = (cx-coor(idx))/L;
    S(idx) = (coor(idx+1)-cx)/L;
elseif idx==0 % Not found any element
    S(1) = 1.0;
    disp('It is possible that discretization is not enough');
else % Last element
    S(tam) = 1.0;
end
return
```

This same code has been employed, for instance, to generate an interactive web page that makes use very efficiently of the PGD concepts. It can be found at http://amb.unizar.es/beamdyn.htm. It represents, see Fig. 5.4, a linear elastic beam that can be interactively manipulated with the aid of the mouse. It places a vertical, unitary load on the upper surface of the beam. It can be noticed how the very efficient time integration algorithms employed for its construction make it possible to remain vibrating for very long times with minimal numerical dissipation.

Execution of the program produces a window with the tip displacement, see Fig. 5.3. Minimal deviation with respecto to the full-order problem solution is obtained. In any case, higher accuracy can be obtained by employing more POD modes, for instance.

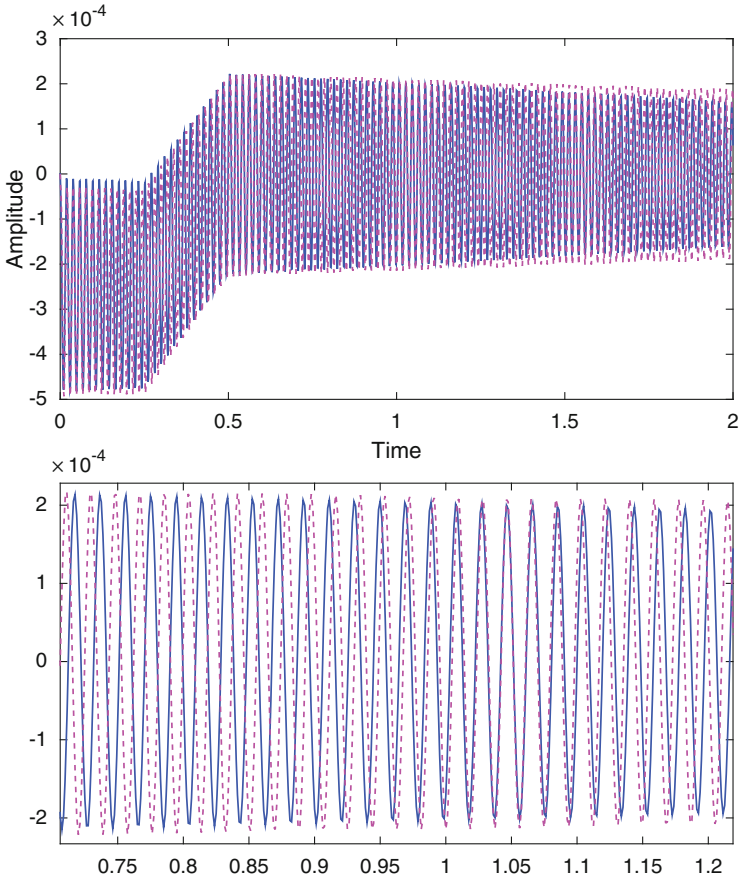

Fig. 5.3 Results of the vibration of a beam (*top*) and detail of the reference solution versus the approximated one (*bottom*)

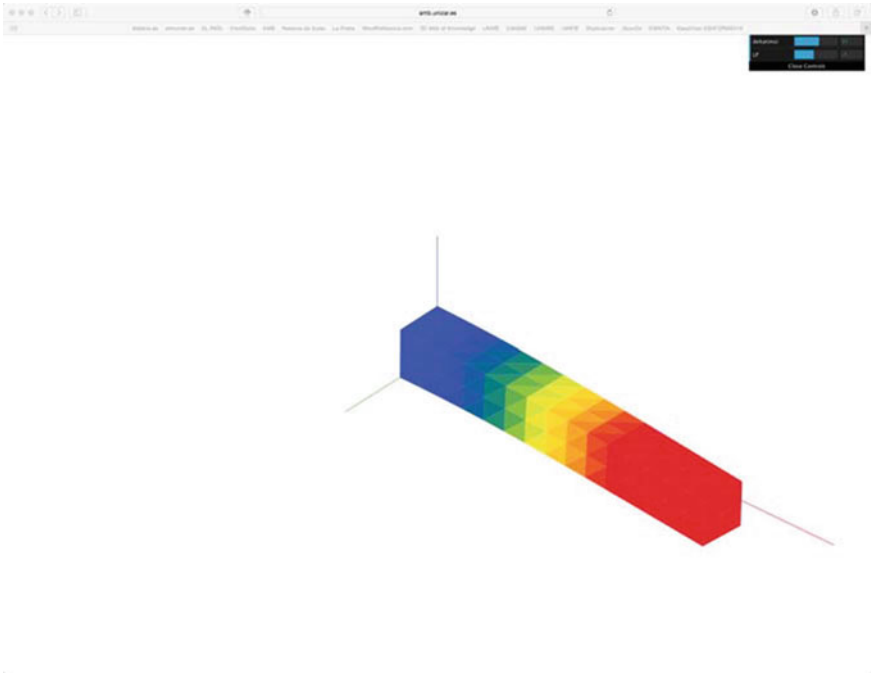

Fig. 5.4 Web implementation of the algorithm here described so as construct an interactive simulator. It represents a linear elastic beam. With the help of the mouse a vertical load is placed on the upper surface of the beam. It can be downloaded from http://amb.unizar.es/beamdyn.htm