Chapter 2
To Begin With: PGD for Poisson Problems

It is one of the first duties of a professor, for example, in any subject, to exaggerate a litlle both the importance of his subject and his own importance in it

—G.H. Hardy, 1940

Abstract In this chapter we cover the detailed implementation of PGD methods for the simplest problem, the Poisson equation. Detailed code is provided and its results compared with data available in the bibliography.

2.1 Introduction

To begin with, let us consider one of the simplest problems, the Poisson equation. This problem was first analyzed in the original publication by Ammar et al. [6], still when the term PGD had not been coined.

Even if it is of little practical interest (very rarely we are interested in separating the space coordinates, unless very special cases such as plate and shell geometries [16] for instance), let us briefly recall the PGD method applied to a Poisson problem, still in two dimensions, for the ease of representation. In it, coordinates x and y have been separated with an eye towards the full comprehension of the method. In general, for non-separable (non parallelepipedic) domains, to separate the physical space is not possible nor even desirable. If you are, nevertheless forced to do it, maybe you could be interested in reading the reference [38].

E. Cueto et al., *Proper Generalized Decompositions*, SpringerBriefs
in Applied Sciences and Technology, DOI 10.1007/978-3-319-29994-5_2

2.2 The Poisson Problem

The D-dimensional Poisson equation writes

$$\triangle u = -f(x_1, x_2, \ldots, x_D), \tag{2.1}$$

where u is a scalar function of $(x_1, x_2, \ldots, x_D)$. Although for representation purposes we will restrict ourselves to the two-dimensional case, we consider here Eq. (2.1), defined in the domain $(x_1, x_2, \ldots, x_D) \in \Omega = (-L, +L)^D$ with vanishing essential boundary conditions. The general treatment of non-vanishing boundary conditions under the PGD framework needs for an special (although straightforward) treatment, deeply discussed from a practical point of view in [38]. We refer the interested reader to that reference.

Under the basic PGD assumption given by Eq. (1.1), we express the unknown solution field as a sum of separable functions, i.e.,

$$u(x_1, x_2, \ldots, x_D) = \sum_{j=1}^{\infty} \alpha_j \prod_{k=1}^{D} F_k^j(x_k),$$

where F_k^j is the jth basis function, with unit norm, which only depends on the kth coordinate.

This a priori infinite sum is then truncated (usually with the help of some error indicator, see [4, 19, 46, 52]) at a number (n) of approximation functions, i.e.:

$$u(x_1, x_2, \ldots, x_D) \approx \sum_{j=1}^{n} \alpha_j \prod_{k=1}^{D} F_k^j(x_k). \tag{2.2}$$

> Note that originally, in [6], the separate functions F_k^j (which we later refer to as *modes* very often) were of unity norm. This is the origin of the α_j weight that accompanies each term j of the sum. While it is not strictly necessary to employ such unitary functions, the relative decay of the α_j weights with j gives a very intuitive notion on the convergence of the series.

Note also that, in Eq. (2.2) only one-dimensional functions F_k^j have been considered. The method is of course much more general than that, and a combination of functions defined in arbitrary dimensional spaces can be employed. Also the (in principle 1D) mesh employed for each function need not be uniform. h-refinements can be made along each dimension as needed.

The modes F_k^j at a given iteration of the method, j, and the α_j value need now to be computed. In the original paper [6] an algorithm was proposed that proceeded by

Step 1: *Projection of the solution in a discrete basis*

If we assume functions $F_k^j(\forall j \in [1,\dots,n]; \forall k \in [1,\dots,D])$ already known, coefficients α_j can be computed by introducing the approximation of u into the Galerkin variational formulation associated with Eq. (2.1):

$$\int_\Omega \nabla u^* \cdot \nabla u d\Omega = \int_\Omega u^* f \, d\Omega. \tag{2.3}$$

PGD methods assume a separated representation of both u and u^*:

$$u(x_1, x_2, \dots, x_D) = \sum_{j=1}^{n} \alpha_j \prod_{k=1}^{D} F_k^j(x_k), \tag{2.4}$$

and

$$u^*(x_1, x_2, \dots, x_D) = \sum_{j=1}^{n} \alpha_j^* \prod_{k=1}^{D} F_k^j(x_k).$$

By introducing both in the weak form of the problem, Eq. (2.3), we arrive at

$$\begin{aligned} &\int_\Omega \nabla \left(\sum_{j=1}^{n} \alpha_j^* \prod_{k=1}^{D} F_k^j(x_k) \right) \cdot \nabla \left(\sum_{j=1}^{n} \alpha_j \prod_{k=1}^{D} F_k^j(x_k) \right) d\Omega \\ &\quad = \int_\Omega \left(\sum_{j=1}^{n} \alpha_j^* \prod_{k=1}^{D} F_k^j(x_k) \right) f \, d\Omega \end{aligned} \tag{2.5}$$

We assume also that the source term $f(x_1, \cdots, x_D)$ admits a separated representation

$$f(x_1, \cdots, x_D) \approx \sum_{h=1}^{m} \prod_{k=1}^{D} f_k^h(x_k),$$

for a sufficiently low number of terms m. If it is not the case, a simple singular value decomposition would in general suffice, maybe in high dimensions (HOSVD) [38]. PGD could equally be employed to this end, by applying it on the identity operator, see [28].

Equation (2.5) involves integrals of products involving D different functions, each one defined in a different coordinate. Let $\prod_{k=1}^{D} g_k(x_k)$ be one of these functions to be integrated. One of the key ingredients of PGD is that the integral over Ω can be performed by integrating each function along its definition interval and then multiplying the D computed integrals according to:

$$\int_{\Omega} \prod_{k=1}^{D} g_k(x_k)\, d\Omega = \prod_{k=1}^{D} \int_{-L}^{L} g_k(x_k) dx_k.$$

This constitutes an essential feature of PGD that makes it possible to solve problems defined in high dimensional spaces.

Since u^* represents an admissible variation of the solution u, the weights α_j^* are arbitrary, too (very much like nodal coefficients of admissible variations in FEM). Thus, Eq. (2.5) allows to compute the n approximation coefficients α_j, solving the resulting linear system of size $n \times n$. This problem is linear and moreover rarely exceeds the order of some tens of degrees of freedom. Thus, even if the resulting coefficient matrix is densely populated, the time required for its solution is negligible with respect to the one required for performing the approximation basis enrichment (step 3).

Step 2: *Checking convergence*

From the solution of u at iteration n given by Eq. (2.4) the residual Re related to Eq. (2.1) can be computed:

$$Re = \frac{\sqrt{\int_{\Omega} (\triangle u + f(x_1, \ldots, x_D))^2}}{u}. \tag{2.6}$$

By fixing a tolerance $Re < \epsilon$, the iteration process can be stopped, thus providing the solution $u(x_1, \ldots, x_D)$.

As per the weak form of the problem, the integral in Eq. (2.6) can be written as the product of one-dimensional integrals by performing a separated representation of the square of the residual.

Step 3: *Enrichment of the approximation basis*

If the stopping criterion has not yet be accomplished, the PGD approximation can be enriched by adding a new functional product $\prod_{k=1}^{D} F_k^{(n+1)}(x_k)$. To this end, the non-linear Galerkin variational formulation related to Eq. (2.1) is then solved:

$$\int_{\Omega} \nabla u^* \cdot \nabla u d\Omega = \int_{\Omega} u^* f\, d\Omega,$$

using the approximation of u given by:

$$u(x_1, x_2, \ldots, x_D) = \sum_{j=1}^{n} \alpha_j \prod_{k=1}^{D} F_k^j(x_k) + \prod_{k=1}^{D} R_k(x_k).$$

Identically, the test function has the form

$$u^*(x_1, x_2, \ldots, x_D) = R_1^*(x_1)\cdot R_2(x_2)\cdot\ldots\cdot R_D(x_D)+\ldots+R_1(x_1)\cdot R_2(x_2)\cdot\ldots\cdot R_D^*(x_D),$$

by simply applying the rules of variational calculus.

This leads finally to a non-linear variational problem (note that we seek a product of functions expressed in a one dimensional finite element basis), whose solution allows to compute the D sought functions $R_k(x_k)$. Functions $F_k^{(n+1)}(x_k)$ need finally to be normalized.

To solve this problem we introduce a discretization of those functions $R_k(x_k)$. Each one of these functions is approximated using a 1D finite element description. If we assume than p_k nodes are used to construct the interpolation of function $R_k(x_k)$ in the interval $[-L, L]$, then the size of the resulting discrete non-linear problem is $\sum_{k=1}^{k=D} p_k$. The price to pay for avoiding a whole mesh in the multidimensional domain is the solution of a non-linear problem. However, even in high dimensions the size of the non-linear problems remains moderate and no particular difficulties have been found in its solution up to hundreds dimensions. Concerning the computation time, even when the non-linear solver converges quickly, this step consumes the main part of the global computing time.

Different non-linear solvers have been analyzed: Newton or one based on an alternating directions scheme. In this work the last strategy was retained. Thus, in the enrichment step, function $R_1^{s+1}(x_1)$ is updated by assuming known all the others functions (given at the previous iteration of the non-linear solver $R_2^s(x_2), \cdots, R_D^s(x_D)$). Then, functions $R_1^{s+1}(x_1)$, $R_3^s(x_3), \cdots, R_D^s(x_D)$ are assumed known for updating function $R_2^{s+1}(x_2)$, and so on until updating the last function $R_D^{s+1}(x_D)$. Now the convergence is checked by calculating $\sum_{i=1}^{i=D} R_i^{s+1}(x_i) - R_i^s(x_i)^2$. If this norm is small enough we can define the functions $F_k^{(n+1)}(x_k)$ by normalizing the functions $R_1, R_2, \ldots, R_D$ and come back to step 1. On the contrary, if this norm is not small enough, a new iteration of the non-linear solver should be performed by updating functions $R_i^{s+2}(x_i)$, $i = 1, \cdots, D$ and then checking again the convergence. Despite its simplicity, our experience proves that this strategy is in fact very robust.

We must recall that the technique that we proposed in the papers just referred, is not a universal strategy able to solve any kind of multidimensional partial differential equation (PDE). Thus, the efficient application of the technique that we just described requires the separability of all the fields involved in the model. Obviously, this separability is not always possible because some functions need a tremendous number of sums. On the other hand, even when the field is separable (one could perform this separation by invoking for example the SVD or the multidimensional SVD) the finite sums decomposition of general multidimensional functions is not realistic because the amount of memory needed for storing the discrete form of such functions before applying the multidimensional SVD.

In many physical models (see for example [6, 25]) a fully separation (consisting of a sum of products of one-dimensional functions) could not be envisaged from a practical point of view. Thus, a better approximation lies in writing

$$u(\boldsymbol{x}_1, \cdots, \boldsymbol{x}_d) \approx \sum_{i=1}^{i=N} F_1^i(\boldsymbol{x}_1) \times \cdots \times F_D^i(\boldsymbol{x}_d)$$

where the different functions taking part in the finite sums decomposition are defined in spaces of moderate dimensions, that is $\boldsymbol{x}_i \in \Omega_i \subset \mathbb{R}^{q_i}$, where usually $q_i = 1, 2$ or 3.

2.3 Matrix Structure of the Problem

The approximation to u given by Eq. (2.4) can indeed be further simplified by assuming

$$u(x_1, x_2, \ldots, x_D) = \sum_{j=1}^{n} \prod_{k=1}^{D} F_k^j(x_k), \tag{2.7}$$

i.e., there is no need to assume unit-normed functions and a weighting parameter α_j in the approximations of u. This was the initial approach followed in [6, 7], but we soon realized that it is equally possible to compute directly functions F_k without the need to enforce its unity norm, nor the projection stage of the algorithm presented before.

Consider, for simplicity, a two-dimensional code, although its extension to an arbitrary number of dimensions is strightforward. In it, functions F^i (we are going to rename them now by their two-dimensional counterparts $F^i(x)$ and $G^i(y)$) are approximated by employing (linear in this case) finite elements, so that, at iteration n, the i-th sum of the approximation will be given by

$$u^i(x, y) = \left[\boldsymbol{N}^T \boldsymbol{F}_i \boldsymbol{M}^T \boldsymbol{G}_i\right], \tag{2.8}$$

with $\boldsymbol{N}$ and $\boldsymbol{M}$ the vectors containing the values of finite element shape functions at integration points and $\boldsymbol{F}_i$ and $\boldsymbol{G}_i$ the vectors of nodal values at the FE mesh for the functions $F^i(x)$ and $G^i(y)$, respectively. In the code included below we assume identical approximation along x and y directions so that only a matrix $\boldsymbol{N} = \boldsymbol{M}$ will be necessary.

The same is necessary for the computation of the gradient terms arising in Eq. (2.3),

$$\left[\frac{\partial u}{\partial x} \; \frac{\partial u}{\partial y}\right]^T = \begin{bmatrix} d\boldsymbol{N}^T \boldsymbol{F}_1 \boldsymbol{M}^T \boldsymbol{G}_1 & d\boldsymbol{N}^T \boldsymbol{F}_2 \boldsymbol{M}^T \boldsymbol{G}_2 & \ldots & d\boldsymbol{N}^T \boldsymbol{F}_n \boldsymbol{M}^T \boldsymbol{G}_n \\ \boldsymbol{N}^T \boldsymbol{F}_1 d\boldsymbol{M}^T \boldsymbol{G}_1 & \boldsymbol{N}^T \boldsymbol{F}_2 d\boldsymbol{M}^T \boldsymbol{G}_2 & \ldots & \boldsymbol{N}^T \boldsymbol{F}_n d\boldsymbol{M}^T \boldsymbol{G}_n \end{bmatrix},$$

where $d\boldsymbol{N}$ and $d\boldsymbol{M}$ represent the vector containing the value of shape function's derivatives at Gauss points. A similar expression can be envisaged both for u^* and ∇u^*, while in this case the nodal values of functions $\boldsymbol{F}_i^*$ and $\boldsymbol{G}_i^*$ are arbitrary, as it is well known from standard finite element theories.

When we look for a new term in the approximation, we assume that

$$u(x, y) = \sum_{i=1}^{n} F^i(x)G^i(y) + R(x)S(y), \tag{2.9}$$

while

$$u^*(x, y) = R^*(x)S(y) + R(x)S^*(y).$$

The new, enhanced, expression of the gradients will be

$$\begin{aligned}\begin{bmatrix}\frac{\partial u}{\partial x} & \frac{\partial u}{\partial y}\end{bmatrix}^T &= \sum_i \begin{bmatrix}d\boldsymbol{N}^T\boldsymbol{F}_i\boldsymbol{M}^T\boldsymbol{G}_i \\ \boldsymbol{N}^T\boldsymbol{F}_i d\boldsymbol{M}^T\boldsymbol{G}_i\end{bmatrix} + \begin{bmatrix}\boldsymbol{M}^T\boldsymbol{S}d\boldsymbol{N}^T & \boldsymbol{0} \\ \boldsymbol{0} & \boldsymbol{N}^T\boldsymbol{R}d\boldsymbol{M}^T\end{bmatrix}\begin{bmatrix}\boldsymbol{R} \\ \boldsymbol{S}\end{bmatrix} \\ &= \sum_i \boldsymbol{D}_i + \boldsymbol{E}\begin{bmatrix}\boldsymbol{R} \\ \boldsymbol{S}\end{bmatrix},\end{aligned}$$

and, similarly,

$$u^*(x, y) = \begin{bmatrix}\boldsymbol{R}^T & \boldsymbol{S}^T\end{bmatrix}\begin{bmatrix}\boldsymbol{S}^T\boldsymbol{M}\boldsymbol{N} \\ \boldsymbol{R}^T\boldsymbol{N}\boldsymbol{M}\end{bmatrix}.$$

The same must be done for

$$\begin{bmatrix}\frac{\partial u^*}{\partial x} & \frac{\partial u^*}{\partial y}\end{bmatrix} = \begin{bmatrix}\boldsymbol{R}^{*T} & \boldsymbol{S}^{*T}\end{bmatrix}\begin{bmatrix}\boldsymbol{S}^T\boldsymbol{M}d\boldsymbol{N} & \boldsymbol{S}^T d\boldsymbol{M}\boldsymbol{N} \\ \boldsymbol{R}^T d\boldsymbol{N}\boldsymbol{M} & \boldsymbol{R}^T\boldsymbol{N}d\boldsymbol{M}\end{bmatrix} = \begin{bmatrix}\boldsymbol{R}^{*T} & \boldsymbol{S}^{*T}\end{bmatrix}\boldsymbol{F}^T,$$

and for the source term, by assuming that

$$f(x, y) \approx \sum_h a^h(x)b^h(y).$$

Once all this matrices have been substituted into the weak form of the problem, Eq. (2.3) ,we arrive at

$$\begin{aligned}\int_\Omega \begin{bmatrix}\boldsymbol{R}^{*T} & \boldsymbol{S}^{*T}\end{bmatrix}\sum_i \boldsymbol{F}^T\boldsymbol{D}_i d\Omega + \int_\Omega \begin{bmatrix}\boldsymbol{R}^{*T} & \boldsymbol{S}^{*T}\end{bmatrix}\boldsymbol{F}^T\boldsymbol{E}\begin{bmatrix}\boldsymbol{R}^T \\ \boldsymbol{S}^T\end{bmatrix}d\Omega \\ = \sum_h \int_\Omega \begin{bmatrix}\boldsymbol{R}^{*T} & \boldsymbol{S}^{*T}\end{bmatrix}\begin{bmatrix}\boldsymbol{S}^T\boldsymbol{M}b^h(y)\boldsymbol{N}a^h(x) \\ \boldsymbol{R}^T\boldsymbol{N}a^h(x)\boldsymbol{M}b^h(y)\end{bmatrix}d\Omega.\end{aligned} \tag{2.10}$$

These integrals in Ω can in fact be separated (since every term is) into a sequence of integrals along x and y coordinates. The resulting terms for matrices $\boldsymbol{F}^T\boldsymbol{D}_i$ and $\boldsymbol{F}^T\boldsymbol{E}$ will involve a repeated evaluation of four terms, namely,

$$\int_{x=-L}^{x=+L} d\boldsymbol{N}d\boldsymbol{N}^T dx \text{ and } \int_{y=-L}^{y=+L} d\boldsymbol{M}d\boldsymbol{M}^T dy \tag{2.11}$$

and

$$\int_{x=-L}^{x=+L} \boldsymbol{N}\boldsymbol{N}^T dx \text{ and } \int_{y=-L}^{y=+L} \boldsymbol{M}\boldsymbol{M}^T dy, \tag{2.12}$$

which are referred to as `p1` and `p2`, respectively, in routine `elemstiff` (see the call `[Km{i1},Mm{i1}] = elemstiff(coor{i1})` in the main file of the code). The code below in fact assumes that $\boldsymbol{N} = \boldsymbol{M}$, since equal partitions are made along x and y directions. For instance, the term 11 of the integration of matrix $\boldsymbol{F}^T\boldsymbol{E}$ has the form,

$$\int_{\Omega} (\boldsymbol{F}^T\boldsymbol{E})_{11} d\Omega = \left(\int_{x=-L}^{x=+L} d\boldsymbol{N}d\boldsymbol{N}^T dx\right) \boldsymbol{S}^T \left(\int_{y=-L}^{y=+L} \boldsymbol{M}\boldsymbol{M}^T dy\right) \boldsymbol{S}. \tag{2.13}$$

Similarly, the right-hand-side term in Eq. (2.10) has the form,

$$\sum_{h=1}^{m} \begin{bmatrix} \boldsymbol{S}^T \left(\int_{y=-L}^{y=+L} \boldsymbol{M}b^h(y)dy\right) \left(\int_{x=-L}^{x=+L} \boldsymbol{N}a^h(x)dx\right) \\ \boldsymbol{R}^T \left(\int_{x=-L}^{x=+L} \boldsymbol{N}a^h(x)dx\right) \left(\int_{y=-L}^{y=+L} \boldsymbol{M}b^h(y)dy\right) \end{bmatrix}. \tag{2.14}$$

The problem, finally, renders Eq. (2.3) in a matrix form that can be simplified, after invoking the arbitrariness of $\boldsymbol{R}^{*T}$ and $\boldsymbol{S}^{*T}$, to

$$\boldsymbol{V}_1(\boldsymbol{R}, \boldsymbol{S}) + \boldsymbol{K}(\boldsymbol{R}, \boldsymbol{S}) \begin{bmatrix} \boldsymbol{R} \\ \boldsymbol{S} \end{bmatrix} = \boldsymbol{V}_2(\boldsymbol{R}, \boldsymbol{S}),$$

which is more easily recognized if we write it in the form

$$\boldsymbol{K}(\boldsymbol{R}, \boldsymbol{S}) \begin{bmatrix} \boldsymbol{R} \\ \boldsymbol{S} \end{bmatrix} = \boldsymbol{V}_2(\boldsymbol{R}, \boldsymbol{S}) - \boldsymbol{V}_1(\boldsymbol{R}, \boldsymbol{S}) = \boldsymbol{V}(\boldsymbol{R}, \boldsymbol{S}). \tag{2.15}$$

It is important to note that the problem in Eq. (2.15) is non-linear, since we look for functions R and S, but both appear multiplied to each other in Eq. (2.9). You can choose your favorite linearization procedure (Newton methods, for instance). In our experience, fixed-point, alternated directions algorithms render excellent results and, despite their general lack of convergence, this is rarely found in practice.

2.4 Matlab Code for the Poisson Problem

The code main file is called `main.m`, of course! Its content is reproduced below.

```
%
%                  PGD Code for Poisson problems
%               D. Gonzalez, I. Alfaro, E. Cueto
%                   Universidad de Zaragoza
%                      AMB-I3A Dec 2015
%
clear all; close all; clc;
%
% VARIABLES
%
ndim = 2; nn = 40.*ones(ndim,1); % # of Dimensions, # of Elements
num_max_iter = 15; % Max. # of summands for the approach
TOL = 1.0E-4; npg = 2; % Tolerance, Gauss Points
coor = cell(ndim,1); % Coordinates in each direction
L0 = -1.*ones(ndim,1);
L1 = ones(ndim,1); % Geometry (min,max coordinates)
for i1=1:ndim,
    coor{i1} = linspace(L0(i1),L1(i1),nn(i1));
end
%
% ALLOCATION OF IMPORTANT MATRICES
%
Km = cell(ndim,1); % "Stiffness" matrix \int dN dN dx, Eq. (2.10)
Fv = cell(ndim,1); % R and S sought enrichment functions
Mm = cell(ndim,1); % "Mass" matrix \int N N dx, Eq. (2.11)
V = cell(ndim,1);  % Source term in separated form
%
% COMPUTING STIFFNESS AND MASS MATRICES ALONG EACH DIRECTION
%
for i1=1:ndim,
    [Km{i1},Mm{i1}] = elemstiff(coor{i1});
end
%
% SOURCE TERM IN SEPARATED FORM
%
% Let us begin by a separable expression. Evaluation of Eq. (2.13)
Ch{1,1} = @(x) cos(2*pi*x); Ch{2,1} = @(y) sin(2*pi*y);
% Try this new source term by yourself by simply uncommenting next 2 lines!
% Ch{1,1} = @(x) x.*x; Ch{1,2} = @(x) -1.0+0.0*x;
% Ch{2,1} = @(y) 1.0+0.0*y; Ch{2,2} = @(y) y.*y;
for j1=1:ndim
    for k1=1:size(Ch,2)
        V{j1}(:,k1) = Ch{j1,k1}(coor{j1});
    end
    % Although in this case we have a closed-form expression for the source
    % term, in general we know its nodal values.
    V{j1} = Mm{j1}*V{j1};
end
%
% BOUNDARY CONDITIONS
%
CC = cell(ndim,1);
for i1=1:ndim,
    IndBcnode{i1} = [1 numel(coor{i1})];
end
for i1=1:ndim,
    CC{i1} = setxor(IndBcnode{i1},[1:numel(coor{i1})])';
end
%
% ENRICHMENT OF THE APPROXIMATION, LOOKING FOR R AND S
%
```

```
num_iter = 0; iter = zeros(1); Aprt = 0; Error_iter = 1.0;
while Error_iter>TOL && num_iter<num_max_iter
    num_iter = num_iter + 1; R0 = cell(ndim,1);
    for i1=1:ndim
      % Initial guess for R and S.
      % It works equally well by choosing something random.
      R0{i1} = ones(numel(coor{i1}),1);
     % We impose that initial guess for functions R and S verify
     % homogeneous essential boundary conditions.
      R0{i1}(IndBcnode{i1}) = 0;
    end
    %
    % ENRICHMENT STEP
    %
    [R,iter(num_iter)] = enrichment(Km,Mm,V,num_iter,Fv,R0,CC,TOL);
    for i1=1:ndim, Fv{i1}(:,num_iter) = R{i1}; end % R (S) is valid, add it
    %
    % STOPPING CRITERION
    %
    Error_iter = 1.0;
    % One possible criterion is to stop when the norm of the new sum is
    % negligible wrt the pair of functions with the maximum norm
    for i1=1:ndim, Error_iter = Error_iter.*norm(Fv{i1}(:,num_iter)); end
    Aprt = max(Aprt,sqrt(Error_iter));
    Error_iter = sqrt(Error_iter)/Aprt;
    fprintf(1,'%dst summand in %d iterations with a weight of %f\n',...
        num_iter,iter(num_iter),sqrt(Error_iter));
end
num_iter = num_iter - 1;% the last sum was negligible, we discard it.
fprintf(1,'PGD Process exited normally\n\n');
save('WorkSpacePGD_Basic.mat');
%
% POST-PROCESSING
%
for i1=1:ndim
    figure;
    plot(coor{i1},Fv{i1}(:,1:num_iter));
end
figure;
if ndim==2
    surf(coor{1},coor{2},Fv{2}*Fv{1}');
end
```

The source code reproduce before includes a call to a function called `elemstiff` that, obviously, provides with the stiffness matrix for each 1D element in the problem. It is reproduced below:

```
function [p1,p2] = elemstiff(coor)
% function [p1,p2] = elemstiff(coor)
% For the coordinates coor, obtains p1 (Stiffness) and p2(Mass) matrices
% Universidad de Zaragoza - 2015

nen = numel(coor); p1 = zeros(nen); p2 = zeros(nen); %  p3 = zeros(nen,1);
X = coor(1:nen-1)'; % Left coordinate of the elements
Y = coor(2:nen)'; % Right coordinate of the elements
L = Y - X; % Length of the elements
sg = [-0.57735027, 0.57735027]; wg  = [1, 1]; % Gauss and weight points
npg = numel(sg);
for i1=1:nen-1
    c = zeros(1,npg);
    N = zeros(nen,npg);
    dN = zeros(nen,npg);
    c(1,:) = 0.5.*(1.0-sg).*X(i1) + 0.5.*(1.0+sg).*Y(i1);
    N(i1+1,:) = (c(1,:)-X(i1))./L(i1);
    N(i1,:) = (Y(i1)-c(1,:))./L(i1);
```

```
    dN(i1+1,:) = ones(1,npg)./L(i1);
    dN(i1,:) = -dN(i1+1,:);
    for j1=1:npg
        p1 = p1 + dN(:,j1)*dN(:,j1)'*0.5.*wg(j1).*L(i1); % dNůdN
        p2 = p2 + N(:,j1)*N(:,j1)'.*0.5.*wg(j1).*L(i1); % NůN
    end
end
return
```

Once the sequence of 1D problems has lead us to a new term in the PGD approximation, we check if it is enough for the prescribed accuracy. If not, we add a new couple of functions in the `enrichment` function:

```
function [R,iter] = enrichment(K,M,V,num_iter,FV,R,CC,TOL)
% function [R,iter] = enrichment(K,M,V,num_iter,FV,R,CC,TOL)
% Compute a new sumand by fixed-point algorithm using PGD
% Universidad de Zaragoza, 2015
iter = 1;
mxit = 25; % # of possible iterations for the fixed point algorithm
error = 1.0e8; % Initialization
ndim = size(FV,1); % Number of Variables
%
% FIXED POINT ALGORITHM
%
while abs(error)>TOL
    Raux = R; % Remember: R is a cell containing both R and S
    for i1=1:ndim % Alternating between R and S
        matrix = zeros(numel(R{i1})); % K matrix in Eq. (2.14)
        source = zeros(size(R{i1},1),1);% V2-V1 in Eq. (2.14)
        %
        % COMPUTING K MATRIX Eq (2.14)
        %
        for i2=1:ndim % Products in sum = ndim (2 in this case)
            FTE = 1.0; % Computing F^T E in Eq. (2.8)
            % Remember: K = \int dN dN dx and M = \int N N dx
            for i3=1:ndim
                if i3==i2
                    if i3==i1
                        FTE = FTE.*K{i3};
                    else
                        FTE = FTE.*(R{i3}'*K{i3}*R{i3});
                    end
                else
                    if i3==i1
                        FTE = FTE.*M{i3};
                    else
                        FTE = FTE.*(R{i3}'*M{i3}*R{i3});
                    end
                end
            end
            matrix = matrix + FTE;
        end
        %
        % COMPUTING V2 in Eq. (2.14)
        %
        for j1=1:size(V{i1},2) % Number functions of the source
            V2 = 1.0;
            for i2=1:ndim
                if i2==i1
                    V2 = V2.*V{i2}(:,j1);
                else
                    V2 = V2.*(R{i2}'*V{i2}(:,j1));
                end
            end
            source = source + V2;
```

```
        end
        %
        % COMPUTING V1 in Eq. (2.14)
        %
        for j1=1:num_iter-1
            for i2=1:ndim % Terms in sum
                FTD = 1.0; % COMPUTING F^T D in Eq. (2.9)
                for i3=1:ndim
                    if i3==i2
                        if i3==i1
                            FTD = FTD.*(K{i3}*FV{i3}(:,j1));
                        else
                            FTD = FTD.*(R{i3}'*K{i3}*FV{i3}(:,j1));
                        end
                    else
                        if i3==i1
                            FTD = FTD.*(M{i3}*FV{i3}(:,j1));
                        else
                            FTD = FTD.*(R{i3}'*M{i3}*FV{i3}(:,j1));
                        end
                    end
                end
                source = source - FTD; % Note that source = V2-V1
            end
        end
        %
        % SOLVE Eq. (2.14) FOR EACH DIRECTION
        %
        R{i1}(CC{i1}) = matrix(CC{i1},CC{i1})\source(CC{i1});
        % We normalize S. R takes care of the alpha constant in Eq. (2.2)
        if i1~=1, R{i1} = R{i1}./norm(R{i1}); end
    end
    % If two successive Rs are too similar, we stop
    error = 0;
    for j1=1:ndim
        error = error + norm(Raux{j1}-R{j1});
    end
    error = sqrt(error);
    iter = iter + 1;
    if iter == mxit % If we reach the max # of iterations, we exit
        return;
    end
end
return
```

After executing this code in your own Matlab client, it provides you with the following figures. In Fig. 2.1 the obtained solution for the Poisson problem is show. Since the source term $\cos(2\pi x)\sin(2\pi y)$ is separable, the code provides the solution with one only term in the PGD sum. Of course, this is not always the case (indeed, it is almost never the case!). The two functions obtained whose multiplication gives the bi-dimensional solution are plotted in Fig. 2.2. Actually, both resemble very much to (the finite element approximation of) the cos and sin functions, respectively.

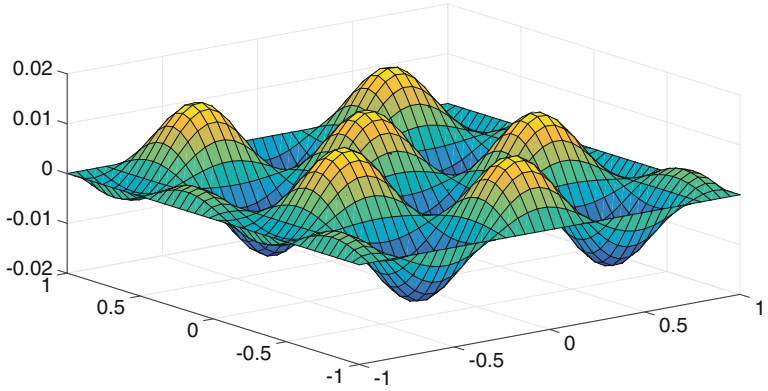

Fig. 2.1 Solution to the poisson problem, as given by the PGD code

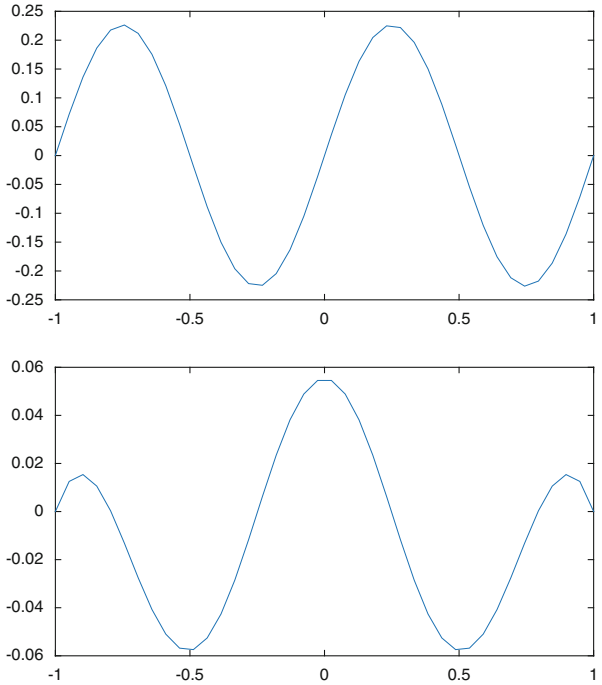

Fig. 2.2 Solution to the poisson problem. Single modes encountered in the x-direction (*top*) and y-direction (*bottom*)

http://www.springer.com/978-3-319-29993-8

Proper Generalized Decompositions
An Introduction to Computer Implementation with Matlab

Cueto, E.; González, D.; Alfaro, I.

2016, XII, 96 p. 20 illus., 1 illus. in color., Softcover
ISBN: 978-3-319-29993-8