Description and implementation of an algebraic multigrid preconditioner for $H^1$-conforming finite element schemes

Descripción e implementación de un precondicionador multinivel algebraico para esquemas de elementos finitos $H^1$-conformes

Descrição e implementação de um pré-condicionador multinível algébrico para esquemas de elementos finitos $H^1$-conformes

Abstract

This paper presents detailed aspects regarding the implementation of the Finite Element Method (FEM) to solve a Poisson's equation with homogeneous boundary conditions. The aim of this paper is to clarify details of this implementation, such as the construction of algorithms, implementation of numerical experiments, and their results. For such purpose, the continuous problem is described, and a classical FEM approach is used to solve it. In addition, a multilevel technique is implemented for an efficient resolution of the corresponding linear system, describing and including some diagrams to explain the process and presenting the implementation codes in MATLAB®. Finally, codes are validated using several numerical experiments. Results show an adequate behavior of the preconditioner since the number of iterations of the PCG method does not increase, even when the mesh size is reduced.
Keywords: Finite element methods; $H^1$-conforming schemes; low-order approximations; multilevel techniques; computational implementation; MATLAB®

Resumen
En este artículo se presenta, en forma detallada, aspectos sobre la implementación del Método de Elementos Finitos (FEM, por sus siglas en inglés), para resolver una ecuación de Poisson con condiciones de frontera homogéneas. El objetivo de este trabajo es clarificar los detalles de esta implementación, tales como la construcción de los algoritmos, creación de experimentos numéricos y los resultados acerca de estos. Por ello, se describe el problema continuo y se muestra un enfoque FEM clásico para resolverlo. Después, se establece una técnica multiniveles para la resolución eficiente del sistema lineal correspondiente, que describe e incluye algunos diagramas para explicar el proceso y presenta los códigos de la implementación en MATLAB®. Finalmente, se realiza una validación de los códigos con varios experimentos numéricos. Los resultados muestran un comportamiento adecuado del precondicionador debido a que el número de iteraciones del método PCG no se incrementa, incluso cuando el tamaño de la malla se reduce.

Palabras clave: Métodos de elementos finitos; esquemas $H^1$ conformes; aproximaciones de bajo orden; técnicas multiniveles; implementación computacional; MATLAB®

Resumo
Este artigo apresenta, em detalhes, aspectos sobre a implementação do Método dos Elementos Finitos (MEF) para resolver uma equação de Poisson com condições de contorno homogêneas. O objetivo deste trabalho é esclarecer os detalhes dessa implementação, tais como a construção dos algoritmos, a criação de experimentos numéricos e os resultados sobre eles. Descreve-se, portanto, o problema contínuo e mostra-se uma abordagem clássica do MEF para resolvê-lo. Em seguida, estabelece-se uma técnica multinível para a resolução eficiente do sistema linear correspondiente, que descreve e inclui alguns diagramas para explicar o processo e apresenta os códigos de implementação no MATLAB®. Finalmente, realiza-se uma validação dos códigos com várias experiências numéricas. Os resultados mostram um comportamento adequado do pré-condicionador devido ao número de iterações do método PCG não aumentar, mesmo quando o tamanho da malha é reduzido.

Palavras-chave: Métodos de elementos finitos; esquemas $H^1$ compatíveis; aproximações de baixa ordem; técnicas multiníveis; implementação computacional; MATLAB®

Introduction

The main goal of this paper is to spread some computational aspects about the implementation of numerical methods for engineering, computer science and mathematics. The original contribution is not related to the mathematical analysis of the numerical methods for partial differential equations (PDE), but rather to explain the aspects about an efficient and clear computational implementation of a numerical method to approximate the solution of a PDE. We hope that the present work will be used for researchers and people interested in learning this kind of topics. It is important to note that there are papers and books which present in detail the methods that we discuss on this work. Nevertheless, there is not literature concerning to the
explanation about the implementation of preconditioning techniques for linear systems associated to PDE.

In the first part of this article we discuss some general aspects related to the $H^1$-conforming Finite Element Method (FEM). The FEM approach is considered as one of the well-established and useful technique for the numerical solution of problems in different fields, described with the use of partial differential equations with different types of boundary conditions. One of the most important aspects related to the success of FEM corresponds to the fact that it is based largely on the basic finite element procedures used: the formulation of the problem in variational form, the discretization of this formulation and the effective solution of the resulting finite element equations. These features do not change from different types of problems (see, e.g., Ciarlet (2002) and Johnson (2009), for details).

In general, the discrete problem for FEM requires to solve a linear system of the form $Ax = b$. This kind of systems are usually solved by iterative methods, which approximate the exact solution of the system with a certain number of accurate digits. The corresponding selection of the iterative solver depends on the properties of matrix $A$. For example, for symmetric and positive definite matrices the most popular method is the Conjugate Gradient (CG) method. However, in general, iterative solver depends on the condition number of the matrix of the system, which is related to the performance of the method. According to this, preconditioning techniques can be considered to accelerate convergence of the iterative methods. There are several types of preconditioning techniques, for example, one simple preconditioning technique of the system is to consider the equivalent system $M^{-1}Ax = M^{-1}b$ for some invertible matrix $M$, which improve the conditioning of the system (see, e.g., Chen (2005)).

An important advance about preconditioning of large linear systems is given by multigrid techniques. The main idea in multigrid approaches is that any iterative solver works on a series of nested meshes, usually defined by a coarsening strategy, which depends on each mesh. Geometric grids are preferred for structured meshes, but algebraic multigrid (AMG) methods are better for unstructured meshes (see, e.g, Stüben (2001)). The AMG requires a simple relaxation process, usually Jacobi or Gauss-Seidel relaxation, and a suitable construction of the coarsening and interpolation operators. The goal of this kind of methods is to define the coarse level variables as a subset of the fine level variables. In other words, to split the fine level variables in two disjoint sets, one of them contains the coarse level variables, and the second one the remaining variables. In practice, the algorithms are usually based on heuristics. For example, for Beck’s algorithm, the coarsening strategy are constructed with the information of the graph associated to the matrix of the system (see Beck (1999)).

Finally, this paper is organized as follows. First, we present a classical FEM scheme for a Poisson equation with homogeneous boundary conditions and describe some aspects about its computational implementation. Next, we present an explanation about preconditioning techniques for primal-FEM formulations. In particular, we focus in the algebraic multigrid preconditioner from Beck (1999), since it is an efficient and simple preconditioner for our variational formulation. However, we include more details related to the construction of the meshes and the coarsening operators. In this way,
we describe more aspects than Beck (1999) in order to allow more people to understand the main aspects of the computational implementation of multigrid approaches. In next section, we perform some numerical experiments and discuss the results, to describe the advantages and disadvantages of the preconditioner. Here, we present a complete MATLAB® code. Finally, we provide some conclusions and further work related to the preconditioning techniques for FEM schemes.

**H¹-conforming finite element method**

In this section we present an example of a classical boundary value problem in order to justify the construction of the algebraic multigrid preconditioner. More precisely, given a bounded polygonal domain Ω in IR² with boundary Γ, we seek a scalar field u such that

\[-Δu = f \text{ in } Ω \text{ and } u = 0 \text{ on } Γ \quad (1)\]

where \( f \in L^2(Ω) \) is a volume force, and as usual \( Δu = \frac{∂^2 u}{∂x^2} + \frac{∂^2 u}{∂y^2} \). Now, given \( ν \in H^1_0(Ω) \) from the Poisson equation note that

\[-∫_Ω Δuv = ∫_Ω f v \]

where, employing the Green’s formula in the left-hand side, we arrive at

\[∫_Ω νu · νv = ∫_Ω f v \]

for all \( ν ∈ H^1_0(Ω) \). In other words, we obtain a variational formulation of problem (1), which reads: Find \( u ∈ H^1_0(Ω) \) such that

\[∫_Ω νu · νv = ∫_Ω f v \forall ν ∈ H^1_0(Ω). \quad (2)\]

It is well known that this problem has a unique solution, which continuously depends of the source \( f \).

It is important to note from the new problem (2) that it only requires one order derivative and it establishes a function Hilbert space for the solution. On the other hand, the problem (1) does not allow us the explicit calculation of the solution \( u \), because \( H^1_0(Ω) \) is an infinite dimensional space. A classical way of circumventing this drawback is the introduction of a finite dimensional subspace \( H_h \) of the space \( H^1_0(Ω) \) defined above consisting of piecewise polynomial continuous functions. More precisely, given \( T_h \) be a shape-regular triangulation of \( Ω \) without the presence of hanging nodes, and an integer \( k ≥ 1 \), it follows that

\[H_h = \{ν ∈ C^0(Ω)/ ν |_T ∈ P_k(T) \forall T ∈ T_h\}\]

where, we let \( P_k(T) \) be the space of polynomials defined in \( T \) of total degree at most \( k \). Furthermore, we define a discrete variational formulation (i.e. Galerkin scheme) which reads: Find \( u_h ∈ H_h \) such that

\[∫_Ω νu_h · νv_h = ∫_Ω f v_h \forall ν_h ∈ H_h. \quad (3)\]

It is easy to check that the problem (3) has a unique solution \( u_h ∈ H_h \) (see, for example, Ciarlet (2002)), which in fact is an approximation of the continuous solution \( u ∈ H^1_0(Ω) \) of problem (2). Furthermore, there is a basis \( \{φ_i^h\}_{i=1,...,dim H_h} \) of \( H_h \), and then

\[u_h = ∑_{i=1}^{dim H_h} α_i φ_i^h\]
where, at least for \( k = 1 \) (low order), \( \{ \varphi^h_1 \}_{i=1}^{\dim H_h} \) is called a Lagrange-type basis, because it holds that \( \alpha_k = u_h(x_k) \), for each vertex \( x_k \) of \( T_h \). In other words, the unknowns of the problem (3) correspond to the values of the solution at the vertices of the triangulation of \( \Omega \).

In general, the finite element method (FEM) for solving partial differential equations, is one of the high-order discretization schemes that has become a very active research area during the last decade. The main advantage of FEM approaches includes numerical solving of big partial differential systems through the analysis of small versions of the problem (on each element of \( T_h \)). There are some variations of the FEM techniques, for example, the Galerkin scheme (3) is known as the classical finite element technique. On the other hand, there is a locally discontinuous version originally introduced in Cockburn (1998), and a virtual version presented in Beirão da Veiga (2013).

### Computational implementation

A fact of the development of FEM methods lies in the difficulty of its computational implementation. Actually, there is few literature detailing the computational implementation of FEM methods, for example, one of the most famous contribution was presented in Carstensen (2002). In the present paper, we do not present details of the implementation of FEM schemes, but in order to explain a preconditioning technique below, we consider some general aspects. Indeed, let \( T \) be an element of \( T_h \) and define \( \{ \varphi^T_1 \}_{i=1}^{\frac{(k+1)(k+2)}{2}} \) as the Lagrange basis on \( T \), which holds

\[
\text{dof}^T_1(\varphi^T_1) = \begin{cases} 1 & i = j \\ 0 & i \neq j \end{cases}
\]

where \( \{ \text{dof}^T_1(\varphi^T_1) \}_{j=1}^{\frac{(k+1)(k+2)}{2}} \) are the local degrees of freedom of \( v \) on \( T \) given by

1. value of \( v \) at the \( i \)th vertex of \( T \), for all vertex \( i \) of \( T \),
2. values of \( v \) at \( k – 1 \) uniformly spaced points on \( e \), for each edge \( e \) of \( T \), for \( k \geq 2 \),
3. values of \( v \) at \( \frac{k(k-1)}{2} \) uniformly spaced points in the interior of \( T \), for \( k \geq 3 \).

It is not difficult to see that the local basis \( \{ \varphi^T_1 \}_{i=1}^{\frac{(k+1)(k+2)}{2}} \) is a restriction of \( \{ \varphi^h_1 \}_{i=1}^{\dim H_h} \) on \( T \). Furthermore, the local discrete operators of \( T \) that contribute to the global linear system of problem (3) are given by

\[
A^T = \left[ \int_T \nabla \varphi^T_i \cdot \nabla \varphi^T_j \right] \quad \text{and} \quad b^T = \left[ \int_T f \varphi^T_i \right].
\]

Using these discrete operators, we can assembly the global system of problem (3) following the algorithm:

1. Define \( A := 0 \) and \( b := 0 \)
2. Define \( m := \frac{(k+1)(k+2)}{2} \) and \( m_o := \frac{k(k-1)}{2} \)
3. For each \( T \in T_h \) do
4. Define \( p \in \mathbb{R}^m \)
5. For each vertex \( i \) of \( T \)
6. \( p(i) = \) global index of \( i \) on \( T_h \)
7. End For
8. For each edge \( e \) of \( T \)
9. \( p = (3 + (k – 1)(e-1) + j) \) (number of nodes of \( T_h \))
10. \( + (k – 1)(\) (global index of \( e \) on \( T_h \) – 1) + j)
11. End For
12. End For
15. For each \( p \) from 1 to \( m_0 \\
16. \quad p(3k+j) = (\text{number of nodes of } \mathcal{T}_h) + (k-1)(\text{number of edges of } \mathcal{T}_h) + j \\
17. \quad \text{End For} \\
18. \quad \mathbf{A}(p,p) = \mathbf{A}(p,p) + \mathbf{A}^T \\
19. \quad \mathbf{b}(p) = \mathbf{b}(p) + \mathbf{b}^T \\
20. \quad \text{End For} \\

Finally, for the incorporation of boundary conditions, we modify the matrix \( \mathbf{A} \) and the vector \( \mathbf{b} \) on the degrees of freedom corresponding to the boundary edges of \( \mathcal{T}_h \). The main idea here, is that each row of the linear system associated to the boundary degrees of freedom becomes exactly the boundary condition of problem (1). The incorporation can be done as we described before. However, in order to define a graph-strategy for the construction of an algebraic multigrid for the global linear system, we actually eliminate the degrees of freedom corresponding to the boundary edges of \( \mathcal{T}_h \), and then the main unknowns of the system are the values of the solution in the interior nodes of \( \mathcal{T}_h \).

**Preconditioning techniques**

In practice, the linear system associated to problem (3) has a large number of unknowns, then, solving it by Gaussian elimination is quite slow, because this method has a complexity of \( O(n^3) \) (see, for details, Saad (2003)). It is well-known (see, for example, Ciarlet (2002)) that \( \mathbf{A} \) is a symmetric positive definite sparse matrix. Then, the Conjugate Gradient method (CG) is the most appropriate to approximate the system solution. However, for the \( m \)th iteration of CG holds that

\[
\| \mathbf{x} - \mathbf{x}^{(m)} \|_A \leq 2 \left( \frac{\sqrt{k} - 1}{\sqrt{k} + 1} \right)^m \| \mathbf{x} - \mathbf{x}^{(0)} \|_A \\
\]

where \( k \) is the condition number of \( \mathbf{A} \). It follows for \( k \gg 1 \) that the term \( \frac{\sqrt{k} - 1}{\sqrt{k} + 1} \) behaves like \( 1 - \frac{2}{\sqrt{k} + 1} \), which converges towards 1.

Thus, inequality (4) predicts a slow convergence of the CG. On the other hand, the condition number of the matrix \( \mathbf{A} \) of linear system (3) has an asymptotic behavior of \( O(h^{-2}) \), which establishes that \( k \gg 1 \) when we use very fine triangulations.

According to the previous discussion, it is necessary to use techniques that reduce the condition number of \( \mathbf{A} \). In general, these techniques construct a nonsingular matrix \( \mathbf{M} \), which is called a *preconditioner*, such that the matrix \( \mathbf{M}^{-1}\mathbf{A} \) of the equivalent system \( \mathbf{M}^{-1}\mathbf{A}\mathbf{x} = \mathbf{M}^{-1}\mathbf{b} \) has a better condition number than the matrix \( \mathbf{A} \) of the original system. Note that if \( \mathbf{M} = \mathbf{A} \) the system is solved immediately. In the case of CG, which requires symmetry in the matrix, the preconditioned system possess the matrix \( \mathbf{M}^{-1} \cdot \mathbf{A} \cdot \text{transpose} (\mathbf{M}^{-1}) \), which is symmetric. The method of solving a linear system by CG using some preconditioner, is known as a Preconditioned Conjugate Gradient (PCG). Some of the most common preconditioners for symmetric systems are Jacobi, Gauss-Seidel, Cholesky and multigrid techniques (see, for example, Saad (2003)).

Multigrid techniques are the most attractive, because they reduce the size of the matrix of the linear system, by a sequence of nested meshes. Consequently, the number of iterations does not depend on the size of the matrix. More precisely, the matrix of the system is reduced by merging or deleting matrix entries. After the new matrix is generated, the corresponding new system is also...
recursively reduced with the help of the next mesh. Thus, this process is continued and stops when all meshes are used. This sequence of matrices is known as preconditioner levels. For example, in Figure 1 we show a sequence of meshes that satisfy the nested condition. This multigrid preconditioner is known as a geometric multigrid preconditioner (see Trottenberg (2000)).

Once all the preconditioner levels have been generated, some steps of a relaxation method, such as Jacobi or Gauss-Seidel, are performed to solve the linear system. The approximation obtained is restricted to be used as the initial value of the auxiliary linear system given by the next lower level. In the same way this process is followed to the last level. Then, when the $\ell + 1$ system is solved, the solution is prolonged to level $\ell$, where again some steps of a relaxation method are performed before moving on to the next level. This whole process is known as the $V$-cycle, which is illustrated in Figure 2.

Unfortunately, in practice this sequence of nested meshes is not available, only the finest mesh. For this reason, it is necessary to generate this sequence through the fine mesh by agglomerating cells. In Cartesian meshes this process is simpler than in unstructured meshes, where in most cases the agglomeration of cells is not possible. For example, in the case of a triangle mesh, the agglomeration of triangles does not always result in another triangle. An alternative to solve this problem is to create the next level, using the graph that represents the matrix of the system and not by the graph that represents the mesh. This process is known as algebraic multilevel and was introduced by Ruge and Stüben (1987).

**Figure 1. Sequence of nested meshes.** Note: derived from research.

**Figure 2. Scheme for a V-cycle.** Note: derived from research.
Methodology employed

The research proposed in this work considers a scientific approach to a particular problem in the area of Applied Mathematics. It is done under an exploratory quantitative approach. Also, the methods presented are commonly used in problems related to finding approximations for solutions of boundary problems in partial differential equations.

Protocol

1. A particular algebraic multigrid preconditioner is described in detail for the model problem, introduced in the previous section. We consider the method introduced by Beck (1999), which is an interesting work, but it does not show a clear description, making it not accessible for beginning readers. Thus, we intend to explain in more detail some relevant elements of the preconditioner.

2. A proposal for computational implementation for the Beck’s preconditioner (see Beck (1999)) is presented in MATLAB®. The code follows an object-oriented paradigm and, according to its organization, can even be used for the implementation of other multilevel preconditioners different than Beck’s.

3. Once a code has been established in MATLAB® for the Beck’s preconditioner, it is shown its behavior, in order to demonstrate its usefulness. Thus, two model examples of the Poisson problem are considered, whose solution is approximated by FEM scheme. To study the behavior of the preconditioner, only the results related to the resolution of the associated linear systems will be reported. Essentially, the number of iterations performed by the Conjugated Gradient method is shown, when it is used without a preconditioner and with the Beck’s preconditioner.

Additionally, the Incomplete Cholesky method will be considered to compare the behavior between preconditioners. For this purpose, we use meshes that do not exceed 800 000 degrees of freedom. Also, a maximum of 500 iterations and a tolerance relative to the first residue not exceeding $10^{-6}$, since these parameters are sufficient to illustrate the desired behavior. On the other hand, it is important to consider one of the examples with a high number of condition to observe that the AMG method retains adequate behavior, which will allow to demonstrate the robustness of this preconditioner.

4. The results are tabulated and plotted to show the good behavior of Beck’s preconditioner over the other methods considered. In particular, the execution time will not be reported because its measurement depends on various factors that are not necessarily controlled by the authors. This should not be construed as a disadvantage of the AMG method, but as a consequence of implementation in software such as MATLAB® and low performance computers.

Beck’s algebraic multigrid preconditioner

In Ruge and Stüben (1987), Saad (1996), Beck (1999), Castillo and Sequeira (2013), the authors present several algebraic multigrid preconditioners. In particular, in this section we describe an algebraic multigrid technique proposed by Beck (1999), which take advantage of the characteristics of the finite element scheme (3). More
precisely, the author identifies the geometric relations of the mesh in the sparse structure of the matrix.

The first algebraic multigrid preconditioner is presented in Ruge and Stüben (1987), which is based on partitioning the variables in each level of the multigrid in two sets by determining strong connection relations imposed by the structure of the matrix associated to the problem. The idea proposed in Beck’s method is a simple coarsening strategy related to the Ruge-Stüben’s method, which define a graph for the matrix using only the sparse structure and ignoring the strength of the connections, i.e. all connections are treated equally (the value of the entries of the matrix are not considered in each partition).

According to the previous discussion, we explain the algebraic multigrid approach using a general linear system:

\[ Ax = b \]

where \( A \in \mathbb{R}^{m \times m} \) and \( b \in \mathbb{R}^m \). Now we define \( A_1 := A \) and \( m_1 := m \) as the data of the first level, where the new right-hand side will be explained later. Thus, the main idea of this method is to reduce \( A_1 \) to a new matrix \( A_2 \in \mathbb{R}^{m_2 \times m_2} \) with \( m_2 \) considerably smaller than \( m_1 \). Then, this process is repeated in order to generate linear systems with associated matrices \( A_{\ell+1} \in \mathbb{R}^{m_{\ell+1} \times m_{\ell+1}} \) such that \( m_\ell > m_{\ell+1} \), for \( \ell = 1, 2, \ldots, N - 1 \), where \( N \) is the maximum number of levels selected such that the last linear system can be solved efficiently.

To perform this procedure for each level \( \ell = 1, 2, \ldots, N - 1 \) we construct matrices \( R_\ell \in \mathbb{R}^{m_{\ell+1} \times m_\ell} \) and \( P_\ell \in \mathbb{R}^{m_\ell \times m_{\ell+1}} \) in order to compute \( A_{\ell+1} \in \mathbb{R}^{m_{\ell+1} \times m_{\ell+1}} \). The matrix \( R_\ell \) is known as the restriction operator, whereas the matrix \( P_\ell \) is the prolongation operator. Using these operators, we obtain the matrix \( A_\ell \) from the following algorithm, which define the matrices of each level of the algebraic multigrid preconditioner, summarized in Table 1.

1. For \( \ell = 1 \) until \( N - 1 \) do
2. Construct \( R_\ell \) and \( P_\ell \), usually from \( A_\ell \)
3. Define \( A_{\ell+1} = R_\ell A_\ell P_\ell \)
4. End For

**Table 1**

<table>
<thead>
<tr>
<th>Level</th>
<th>Matrix</th>
<th>Restriction</th>
<th>Prolongation</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \ell = 1 )</td>
<td>( A_1 )</td>
<td>( R_1 )</td>
<td>( P_1 )</td>
</tr>
<tr>
<td>( \ell = 2 )</td>
<td>( A_2 )</td>
<td>( R_2 )</td>
<td>( P_2 )</td>
</tr>
<tr>
<td>( \vdots )</td>
<td>( \vdots )</td>
<td>( \vdots )</td>
<td>( \vdots )</td>
</tr>
<tr>
<td>( \ell = N - 1 )</td>
<td>( A_{N-1} )</td>
<td>( R_{N-1} )</td>
<td>( P_{N-1} )</td>
</tr>
<tr>
<td>( \ell = N )</td>
<td>( A_N )</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

*Note: derived from research.*

Finally, in order to describe the construction of each level of the preconditioner (i.e. \( R_\ell \) and \( P_\ell \)), in the following section we analyze a coarsening strategy for sparse matrices, which allow us to construct the restriction and prolongation operators. After that, we introduce the iterative solver based on the V-cycle approach.

**Coarsening**

We now explain the classical design of the coarsening technique briefly described in Section 3 of Beck (1999). In addition, we incorporate more explanations and examples, including a suggestion for a MATLAB® implementation.

The main idea of the coarsening algorithm is to select a favorably distributed subset of the matrix nodes (i.e. indexes of rows and columns), which are to become
the nodes of the new matrix (coarse grid matrix). More precisely, the nodes for the new matrix are known as *master nodes*, whereas the remaining (which will be dropped) are known as *slave nodes*. In particular, this coarsening algorithm satisfies that the number of unknowns is reduced substantially in every coarsening step, and the coarse grid matrices preserve the sparsity pattern of the fine grid matrix, that is, coarse grid basis functions retain a local support and a restricted overlap with neighboring ones.

According to Section 3 in Beck (1999), there are three rules for the selection of master nodes. That is:

1. No master node may be directly connected to another master node.
2. There should be as many master nodes as possible.
3. The values of all master nodes are transferred with weight 1. The value for a slave node $s$ is interpolated from the $n_s$ master nodes it is connected to, where each master node contributes with weight $\frac{1}{n_s}$. This is used in the construction of the restriction operator below.

Here is important to remember that every Dirichlet node have been eliminated in order that the sparsity pattern of the matrix allows the separation of its nodes into two disjoint sets.

On the other hand, let $A \in \mathbb{R}^m$ be a symmetric sparse matrix. Now consider the graph that represents this matrix. More precisely, letting $m$ nodes (numbering from 1 to $m$), we say that node $i$ is connect to node $j$, with $i \neq j$, if entry $A_{ij} \neq 0$. For example, consider matrix

\[
A := \begin{pmatrix}
9 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 0 \\
1 & 9 & 1 & 0 & 0 & 0 & 1 & 1 & 0 \\
0 & 1 & 9 & 1 & 0 & 0 & 0 & 1 & 1 \\
0 & 0 & 1 & 9 & 1 & 0 & 0 & 0 & 1 \\
0 & 0 & 0 & 1 & 9 & 1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 1 & 9 & 1 & 1 & 1 \\
1 & 1 & 0 & 0 & 0 & 1 & 9 & 1 & 0 \\
1 & 1 & 1 & 1 & 1 & 0 & 1 & 1 & 0 \\
0 & 1 & 1 & 1 & 1 & 0 & 1 & 1 & 9 \\
1 & 9 & 1 & 0 & 0 & 0 & 1 & 1 & 0
\end{pmatrix}
\]

\[(5)\]

which graph is given in Figure 3.

**Figure 3. Graph that represents the matrix $A$. Note:** derived from research.

Note that the graph follows from the sparsity pattern of $A$. Now, we select one node to be our first master node. For example, we take node 1 and then, every neighboring node of 1 will be mark as slave node (i.e. node 2, 7 and 8). From the remaining nodes (3, 4, 5, 6 and 9) take the next master node. We take node 3, and then its neighbors (2, 4 and 9) are now slave nodes. Following this procedure, we take node 5 as master node and mark its neighbors as slaves. In Figure 4, we summarize the previous classification strategy.
Figure 4. Master (black) and slave (white) nodes for matrix $A$. Note: derived from research.

Observe that the coarsening process operates in a geometric way by sequentially choosing a master node and eliminating the corresponding neighboring nodes of the graph. Moreover, a primary criterion for selecting the next master node is to take the node with the minimum number of connections (taking into account the eliminations). And a secondary criterion is to follow the original numbering. We use the first choice by listing the nodes in ascending order, according to the number of non-zero elements of each row of $A$. We now present a MATLAB® implementation for previous coarsening strategy.

```matlab
function [marked, sizeM] = coarsening(A)
    m = length(A); % Order of the matrix
    % ----------------------------------------------------
    nnzRow = zeros(1, m); % Number of non-zero elements of
                          % A by row
    for i = 1 : m
        nnzRow(i) = nnz( A(i,:) );
    end
    [~, idxRow] = sort(nnzRow); % Sort indices according to
                               % the number of non-zero elements
    % First, every node is marked with a negative number,
    % in order to indicate that the node is unmarked.
    marked = -1 * ones(1, m);
    sizeM = 0;
    % Using the symmetry of A, we follow the columns of A:
    for j = idxRow
        c = A(:,j); % Define the jth column of A
        % ----------------------------------------------------
```

Helen Guillén-Oviedo, Jeremías Ramírez-Jiménez, Esteban Segura-Ugalde y Filándar Sequeira-Chavarría

Artículo protegido por licencia Creative Commons: BY-NC-ND / Protected by Creative Commons: BY-NC-ND

Uniciencia es una revista de acceso abierto/ Uniciencia is an Open Access Journal.
which establish that the current node 1, 3 and 5 will be, respectively, the node 1, 2 and 3 of the matrix in the next level of the preconditioner.

**Restriction and prolongation operators**

After choosing master nodes, we can assemble the restriction matrix $R_\ell$. Indeed, given $A_\ell \in \mathbb{R}^{m_\ell \times m_\ell}$ the matrix of level $\ell$, let $m_{\ell+1}$ be the number of master nodes obtained by the previous coarsening algorithm, such that $m_{\ell+1} < m_\ell$. Then, letting $n_s$ be the number of neighboring master nodes of slave node $s$, we define $R_\ell \in \mathbb{R}^{m_{\ell+1} \times m_\ell}$ as

\[
(R_\ell)_{ij} := \begin{cases} 
1 & \text{if } j \text{ is } i\text{th master node} \\
1/n_j & \text{if } i \text{ is connected to } j \\
0 & \text{otherwise}
\end{cases}
\]

Observe that the values of all master nodes are transferred with weight 1, whereas the values of all slave nodes are interpolated from its neighboring master nodes. This selection is strongly related to the definition of the degrees of freedom of scheme (3), at least for $k = 1$. For example, the restriction matrix for graph in Figure 4 is given by

\[
R_\ell = \begin{pmatrix}
1 & 1/2 & 0 & 0 & 0 & 0 & 1 & 1 & 0 \\
0 & 1/2 & 1 & 1/2 & 0 & 0 & 0 & 0 & 1/2 \\
0 & 0 & 0 & 1/2 & 1 & 1 & 0 & 1/2 & 1/2
\end{pmatrix}
\]

On the other hand, Beck (1999) suggests a similar construction for prolongation operator $P_\ell \in \mathbb{R}^{m_\ell \times m_{\ell+1}}$. However, in order to preserve symmetry in the matrix of the next level, we define

\[
P_\ell := \text{transpose}(R_\ell)
\]
Finally, using the restriction and prolongation operators we define $A_{\ell+1} := R_\ell A_\ell P_\ell$ as the matrix of next level. Then, we continue the process employing $A_{\ell+1}$ to construct $R_{\ell+1}$ and $P_{\ell+1}$. The stopping criterion corresponds to a maximum order for the last matrix. In addition, the last matrix presents a considerably reduction in the order of the original matrix, then direct methods (for example LU factorization) can be used for solver effects.

According to the above discussion, we now propose the following computational implementation for the construction of all the matrices required by the algebraic multigrid preconditioner.

```matlab
% This function constructs the levels of the algebraic % multigrid preconditioner.
function [MA, MR, MP, L, U, steps] = createLevels(A, maxSize)
    MA = {}; % List of matrices
    MR = {}; % List of restriction operators
    MP = {}; % List of prolongation operators
    numLevels = 0; % Number of levels
    m = length(A); % Order of the matrix
    while m >= maxSize
        % Mark nodes as Master and Slave
        [marked, sizeM] = coarsening(A);
        % Construct Restriction matrix
        R = sparse(sizeM, m);
        for j = 1 : m
            if marked(j) == 0
                % SLAVE NODE
                c = A(:,j); % Define the jth column of A
                I = find(c); % Find indices of non-zero
```
The vector \( \text{steps} \), defined by function \( \text{createLevels}() \), contains the number of relaxation steps for the V-cycles of the preconditioner. For simplicity, we perform \( \ell + 1 \) relaxation steps for level \( \ell \).

**V-cycles iteration**

According to the previous sections, we now aim to present an iterative solver for linear system \( \mathbf{A}\mathbf{x} = \mathbf{b} \) based on the iterative refinement method (see, e.g, Wilkinson (1994) or Burden, Faires and Burden (2015)). Indeed, given, \( \mathbf{A} \in \mathbb{R}^{n \times n} \), and \( \mathbf{b} \in \mathbb{R}^{n} \) and \( \mathbf{x}_0 \in \mathbb{R}^{n} \) an initial guess, we consider the following algorithm:

1. Define \( \mathbf{r}_1 = \mathbf{b} - \mathbf{A}\mathbf{x}_0 \)
2. For \( k = 1 \) until number of V-cycles do
3. For \( \ell = 1 \) until \( N - 1 \) do
4. Define \( \mathbf{x}_\ell = \mathbf{0} \in \mathbb{R}^{m_\ell} \)
5. Perform some steps of a relaxation method to the system \( \mathbf{A}_\ell \mathbf{x}_\ell = \mathbf{r}_\ell \)
6. Define \( \mathbf{r}_{\ell+1} = \mathbf{R}_\ell (\mathbf{r}_\ell - \mathbf{A}_\ell \mathbf{x}_\ell) \)
7. End For
8. Solve \( \mathbf{A}_n \mathbf{x}_n = \mathbf{r}_n \) by a direct method
9. For \( \ell = N - 1 \) until 1 do
10. Define \( x_\ell = x_\ell + p_\ell \cdot x_{\ell+1} \)
11. Perform some steps of a relaxation method to the system \( A_\ell x_\ell = r_\ell \)
12. End For
13. End For

Here \( N \) is the total number of levels defined by createLevels() in the previous section. Notice that, after the calculation of the first residual \( r_1 \) and according to the iterative refinement method, we need to solve alternative linear system \( A y = r_1 \) in order to make the correction \( x_0 = x_0 + y \). Thus, for solving \( A y = r_1 \) we perform some steps of a relaxation method, such as the Jacobi iteration or Gauss-Siedel iteration. Next, we restrict the new residual to the following level. We continue this process until level \( N - 1 \). In last level, we compute the solution of the corresponding system employing a direct solver. In particular, for simplicity we use LU Factorization, but Cholesky Factorization is also a good choice. Finally, through each level, the solution is prolonged and updated until a new approximation to the original system is obtained.

It is important to observe that the original residual \( r_1 \) is not changed. In addition, in order to preverse symmetry (see, e.g., Chen (2005)), we utilize Forward Gauss-Seidel iteration when each residual is restricted. On the other hand, we use Backward Gauss-Seidel iteration when each residual is prolonged. We end this section by illustrating the definition of every \( V \)-cycle in Figure 5. The MATLAB® code for this solver will be presented in function apply() in a section below.

**Preconditioned Conjugate Gradient Method**

The \( V \)-cycle iteration is a solver for the linear system \( Ax = b \). However, this is used as a preconditioner for the usual preconditioned conjugate gradient (PCG) method. More precisely, consider the PCG method (see, e.g., Saad (2003)) given by

1. Define \( r = b - Ax_0 \)
2. Apply preconditioner to solve \( Az = r \)
3. Define \( p = z \)
4. Define \( x = x_0 \)
5. For \( k = 0 \) until convergence do
6. Define \( v = Ap \)
7. Define \( w = r^T z \)
8. Define \( \alpha = \frac{w}{v^T p} \)
9. Update \( x = x + \alpha p \)
10. Update \( r = r - \alpha v \)
11. Apply preconditioner to solve \( Az = r \)
12. Define \( \beta = \frac{r^T z}{w} \)
13. Update \( p = z + \beta p \)
14. End For

Next, note from steps 2 and 11 that it is necessary to solve linear systems for every preconditioner. In this case, we will use the iterative solver defined in previous section.

![Figure 5. Definition of each V-cycle of the algebraic multigrid preconditioner.](image-url)
Complete MATLAB® code

In this section, we propose a simplified computational implementation for the algebraic multigrid preconditioner introduced by Beck (1999). All procedures described in the foregoing sections are unified in a MATLAB® class called amgRB, which is presents at end of this paper (see Annex 1).

On the other hand, consider the following implementation for PCG method:

```matlab
% Preconditioned conjugate gradient method.

function [x, iterUsed] = pcg(P, A, b, x, tol, iterMax)
    r = b - A*x;
    z = P.apply(r);
    p = z;
    nrm0Tol = tol * norm(r);
    iterUsed = 0;
    while iterUsed < iterMax && norm(r) >= nrm0Tol
        v = A * p;
        w = r' * z;
        alpha = w / (v' * p);
        x = x + alpha * p;
        r = r - alpha * v;
        z = P.apply(r);
        beta = (r' * z) / w;
        p = z + beta * p;
        iterUsed = iterUsed + 1;
    end
end
```

Then, the class amgRB, defined at the beginning of this section, can be used as in the following example:

```matlab
P = amgRB(A, maxSize, numVCycles); % Define preconditioner
x0 = zeros(length(A),1); % Define initial guess
[x, k] = pcg(P, A, b, x0, 1.0e-6, 1000)
```

where A, b, maxSize, and numVCycles must be previously defined.
Numerical experiments

In this section, we carry out two numerical experiments to validate our code and illustrate the behavior of the algebraic multigrid preconditioner proposed in Beck (1999) at least for \( k = 1 \). For all the examples, triangular meshes were generated using the software TRIANGLE developed by Shewchuk (1996). In addition, we modify the continuous problem (1) to

\[-\nabla(K\nabla u) = f \text{ in } \Omega \text{ and } u = 0 \text{ on } \Gamma\]

where \( K \in \left[L^2(\Omega)\right]^{2 \times 2} \) is a uniformly positive definite tensor describing the material properties of \( \Omega \). The idea of introducing the tensor \( K \) is to consider a problem with high condition number (see Example 2 below).

The numerical results were obtained using the MATLAB\textsuperscript{®} code introduced in previous section. The corresponding linear systems are solved by the Conjugate-Gradient method with a tolerance of \( 10^{-6} \), 500 as the maximum number of iterations, and taking as initial guess the vector which each entry is equal to one. Here, we use ndofs for the number of unknown of each linear system, that is, the value of \( m_1 \). In addition, we introduce the integer \( \mu > 0 \) in order to define the number of smoothing steps used by Gauss-Seidel methods. More precisely, for a level \( \ell = 1,2, \ldots, N - 1 \), the number of relaxation steps are given by \( \mu + \ell - 1 \). Moreover, we only use one V-Cycle for the AMG preconditioner. Finally, we also consider the classical Incomplete Cholesky preconditioner (see, e.g., Saad (2003)) as an alternative solver.

In Example 1, we consider \( \Omega := (0,1)^2 \), \( K \) the identity matrix, and choose the data \( \ell \) so that the exact solution is given by

\[u(x,y) := \sin(\pi x) \sin(\pi y)\]

for all \((x,y) \in \Omega\). In Figure 6, we show a triangular decomposition for our domain obtained with TRIANGLE, where it is important to note that only unstructured meshes are used.

Figure 6. Mesh for Example 1 with 1531 triangles. Note: derived from research.

In Table 2, the results of the AMG preconditioner studied in this work are presented and compared to the classical CG method. We also obtain results for the Incomplete Cholesky method. The AMG method shows an improvement with respect to the classical methods, in terms of the number of iterations and the obtained residuals. For instance, note that the CG method almost never converges after 500 iterations for any of the experiments, while the Incomplete Cholesky method, although converging, the number of iterations is significantly increasing compare to AMG. In fact, the most important aspect to observe is that the AMG method always converges, and its number of iterations does not exceed 22 (considering all experiments), which allow us to see the robustness of the preconditioner. In addition, these iterations decrease as the parameter \( \mu \) increases, which establishes that several relaxation steps carry
out fewer iterations, but will require more execution time, which is not shown in this paper because the measurement of time depends on multiple reasons related to particular situations.

On the other hand, for the last mesh of 782939 degrees of freedom, we show in Table 3 a description of each level created by the AMG preconditioner. There, it is possible to see a significant reduction in the unknowns of the system. Finally, in Figure 7, we show that the residuals of the AMG method decreases faster.

**Table 2**

*Example 1: Number of iterations and last residual.*

<table>
<thead>
<tr>
<th>ndofs</th>
<th>Iter.</th>
<th>CG</th>
<th>Residual</th>
<th>Iter.</th>
<th>Incomplete Cholesky</th>
<th>Residual</th>
<th>Iter.</th>
<th>AMG</th>
<th>Residual</th>
</tr>
</thead>
<tbody>
<tr>
<td>30802</td>
<td>418</td>
<td>5.3970e-08</td>
<td>52</td>
<td>3.5578e-05</td>
<td>2</td>
<td>15</td>
<td>3.6363e-08</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>258153</td>
<td>500</td>
<td>1.0513e-04</td>
<td>146</td>
<td>6.3646e-05</td>
<td>2</td>
<td>21</td>
<td>9.4378e-09</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>516513</td>
<td>500</td>
<td>1.2038e-04</td>
<td>199</td>
<td>8.1081e-05</td>
<td>2</td>
<td>22</td>
<td>6.4887e-09</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>782939</td>
<td>500</td>
<td>9.8561e-02</td>
<td>228</td>
<td>9.8735e-05</td>
<td>2</td>
<td>18</td>
<td>6.0660e-05</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Table 3**

*Example 1: Description of the levels of the AMG preconditioner for the mesh of 782939 degrees of freedom.*

<table>
<thead>
<tr>
<th>Level</th>
<th>Order of the matrix</th>
<th>Number of non-zero entries</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>782939</td>
<td>5472891</td>
</tr>
<tr>
<td>2</td>
<td>207841</td>
<td>2001147</td>
</tr>
<tr>
<td>3</td>
<td>42911</td>
<td>478627</td>
</tr>
<tr>
<td>4</td>
<td>7873</td>
<td>94105</td>
</tr>
<tr>
<td>5</td>
<td>1364</td>
<td>16372</td>
</tr>
<tr>
<td>6</td>
<td>237</td>
<td>6010</td>
</tr>
</tbody>
</table>

**Figure 7.** *Example 1: Residual history for 100 iterations (left) and 20 iterations (right).*

*Note: derived from research.*
In Example 2, we consider the model problem on a domain with two different materials and an isotropic diffusion. More precisely, $\Omega$ consists of two squares, the first one is the unit square $(0,1)^2$, which has diffusion $\mathbf{K}$ equal to the identity matrix $\mathbf{I}_2$, and within this square, a second square $\left(\frac{1}{3}, \frac{2}{3}\right)^2$ with diffusion $\mathbf{K} = 10^{-10}\mathbf{I}_2$. An example of the meshes for this domain is shown in Figure 8. Here the source function $\ell$ is equal to zero.

![Figure 8](image)

**Figure 8. Mesh for Example 2 with 1543 triangles. Note:** derived from research.

In Table 3, we note that the AMG method performs a behavior similar to the obtained in Example 1, although the condition number of the system matrix in Example 2 is $10^9$ greater than the condition number of Example 1. Thus, the AMG preconditioner does not show a significant change by increasing the conditioning of the matrix. Finally, in Table 4, a level description is presented, whereas in Figure 9, we show the residuals history, all this for the last mesh used in Table 4.

<table>
<thead>
<tr>
<th>Level</th>
<th>Order of the matrix</th>
<th>Number of non-zero entries</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>781506</td>
<td>5458246</td>
</tr>
<tr>
<td>2</td>
<td>208021</td>
<td>2003307</td>
</tr>
<tr>
<td>3</td>
<td>42978</td>
<td>477048</td>
</tr>
<tr>
<td>4</td>
<td>7909</td>
<td>93343</td>
</tr>
<tr>
<td>5</td>
<td>1404</td>
<td>16444</td>
</tr>
<tr>
<td>6</td>
<td>250</td>
<td>4927</td>
</tr>
</tbody>
</table>

**Table 4**

**Example 2: Number of iterations and last residual.**

<table>
<thead>
<tr>
<th>ndofs</th>
<th>Iter.</th>
<th>CG Residual</th>
<th>Incomplete Cholesky Iter.</th>
<th>$\mu$</th>
<th>Iter.</th>
<th>AMG Residual</th>
</tr>
</thead>
<tbody>
<tr>
<td>30649</td>
<td>255</td>
<td>4.3738e-05</td>
<td>42</td>
<td>3.4858e-05</td>
<td>2</td>
<td>11</td>
</tr>
<tr>
<td>257621</td>
<td>500</td>
<td>4.2431e-04</td>
<td>105</td>
<td>7.4950e-05</td>
<td>2</td>
<td>15</td>
</tr>
<tr>
<td>515705</td>
<td>500</td>
<td>1.1529e-02</td>
<td>145</td>
<td>8.6156e-05</td>
<td>2</td>
<td>15</td>
</tr>
<tr>
<td>781506</td>
<td>500</td>
<td>7.3480e-02</td>
<td>174</td>
<td>1.0637e-04</td>
<td>2</td>
<td>17</td>
</tr>
</tbody>
</table>

**Table 5**

**Example 2: Description of the levels of the AMG preconditioner for the mesh of 781506 degrees of freedom.**

<table>
<thead>
<tr>
<th>Level</th>
<th>Order of the matrix</th>
<th>Number of non-zero entries</th>
</tr>
</thead>
</table>
Conclusions and future directions

In this article, we have described an object oriented implementation of the algebraic multigrid preconditioner introduced in Beck (1999), applied to elliptic problems using unstructured triangular meshes. The numerical experiments presented illustrate the appropriate behavior of the preconditioner. In particular, there is not a considerable increase in the number of iterations of PCG method even when the mesh size is decreased.

In the future we would like to consider a new coarsening strategy, which allow us a better selection for master nodes. In addition, in this work we used only H¹-conforming elements in order to explain the preconditioner in a cleaner way. However, the results presented here can be extended to the case of H(div)-conforming elements. Finally, we are interesting to develop an extension of the Beck’s preconditioner to polynomial bases of degree higher than one (that is, \( k > 1 \)), which implies that the degrees of freedom are not only in the vertices, and then the coarsening technique would not be accurate.

Acknowledgements

The work of Jeremías Ramírez was partially supported by Escuela de Matemática de Universidad Nacional, Costa Rica, through the project 0161-17. The work of Esteban Segura was partially supported by Escuela de Matemática and CIMPA, Universidad de Costa Rica, Costa Rica, through the project 821-B7-254. The work of Filánder Sequeira was partially supported by Escuela de Matemática of Universidad Nacional, Costa Rica, through the project 0103-18.

References


Description and implementation of an algebraic multigrid preconditioner for H1-conforming finite element schemes (Helen Guillén-Oviedo, Jeremías Ramírez-Jiménez, Esteban Segura-Ugalde y Filánder Sequeira-Chavarría) in *Uniciencia* is protected by *Attribution-NonCommercial-NoDerivs 3.0 Unported* (CC BY-NC-ND 3.0)
ANNEX 1: MATLAB Class for algebraic multigrid preconditioner

```matlab
classdef amgRB
    properties (GetAccess=private)
        maxSize = 100;  % Default values
        numVCycles = 5;  % Default values
        A = [];          % Matrix for first level
        MA = {};         % List of matrices
        MR = {};         % List of restriction operators
        MP = {};         % List of prolongation operators
        L = [];          % For LU factorization of the last matrix
        U = [];          % For LU factorization of the last matrix
        tol = 1e-6;      % Tolerance for the iterative solver
        steps = [];      % Smoothing steps for Gauss-Seidel (GS)
        ML = {};         % List of lower triangular matrices for GS
        MU = {};         % List of upper triangular matrices for GS
        MFGS = {};       % List of inverse matrices for Forward-GS
        MBGS = {};       % List of inverse matrices for Backward-GS
    end

    % ------------------------------
    % PUBLIC METHODS
    % ------------------------------
methods
    % ------------------------------
    % Construct method
    % This function constructs the levels of the algebraic
    % multigrid preconditioner.
    % ------------------------------
    % Receive:
    % A: is a mxm symmetric positive definite sparse matrix.
    % maxSize: is the maximum size of the matrix in the last
    % level.
    % numVCycles: is the number of V-cycles.
    % Update:
    % MA: is a list of matrices for coarse matrices
    % MR: is a list of matrices for restriction operators
    % MP: is a list of matrices for prolongation operators
    % L: is the lower triangular matrix of the LU factorization
    %    of the last matrix A
    % U: is the upper triangular matrix of the LU factorization
    %    of the last matrix A
    % steps: is a list of the smoothing steps for Gauss-Seidel
    % ML: is a list of lower triangular matrices for GS
end
```
MU: is a list of upper triangular matrices for GS
MFGS: is a list of inverse matrices for Forward-GS
MBGS: is a list of inverse matrices for Backward-GS

function obj = amgRB(A, maxSize, numVCycles)
    obj.A = A;
    obj.maxSize = maxSize;
    obj.numVCycles = numVCycles;
    numLevels = 0; % Number of levels without the last one
    m = length(A); % Order of the matrix
    while m >= maxSize
        % Mark nodes as Master and Slave
        [marked, sizeM] = coarsening(obj, A);
        % Construct Restriction matrix
        R = sparse(sizeM, m);
        for j = 1 : m
            if marked(j) == 0
                c = A(:,j);
                I = find(c);
                [~,~,I] = find( marked(I) );
                nnz = length(I);
                R(I, j) = 1 / nnz;
            else
                R(marked(j), j) = 1;
            end
        end
        % Store matrices on lists
        numLevels = numLevels + 1;
        obj.MR{numLevels} = R;
        obj.MA{numLevels} = A;
        obj.MP{numLevels} = R';
        A = R * A * R';
        m = sizeM;
    end
    % ---- Compute LU Factorization for the last matrix ---
    [L,U] = lu(A);
    obj.L = L;
    obj.U = U;
    % ---------- Create list of smoothing steps ----------
obj.steps = zeros(numLevels, 1);
for i = 1 : numLevels
    obj.steps(i) = i + 1;
end
% ---------------- Setup Gauss-Seidel ----------------
obj.ML{numLevels} = [];
obj.MU{numLevels} = [];
obj.MFGS{numLevels} = [];
obj.MBGS{numLevels} = [];
% Create Gauss-Seidel matrices
for l = 1 : numLevels
    A = obj.MA{l};
    I = speye(size(A));
    obj.ML{l} = tril(A,-1);
    obj.MU{l} = triu(A, 1);
    obj.MFGS{l} = tril(A) \ I;
    obj.MBGS{l} = triu(A) \ I;
end
% ---------------------------------------------------------
% Get the number of levels of the preconditioner
% ---------------------------------------------------------
% k: is the number of levels.
% ---------------------------------------------------------
function [k] = getNumLevels(obj)
    k = length(obj.MR) + 1;
end
% ---------------------------------------------------------
% Apply preconditioner to a vector, used by PCG method
% ---------------------------------------------------------
% x, y: are m-size vectors
% ---------------------------------------------------------
function [y] = apply(obj, x)
    y = zeros(size(x));
    [y, ~, ~] = solve(obj, x, y);
end
% ---------------------------------------------------------
% Iterative solver for Ax = b
% ---------------------------------------------------------
% b: is the right-hand side of the system
% x0: is an initial guess for system Ax = b
% x: is the new approximation for system Ax = b
% cyclesUsed: is the number of V-cycles used
% normResidual: is a list for the norm-2 of the
% residual for each V-cycle
% ---------------------------------------------------------
function [x, cyclesUsed, normResidual] = solve(obj, b, x0)
numLevels = length(obj.MR); % Number of levels without
    % the last one, that is, N-1
% -------- When the number of levels is 1 ----------
if numLevels == 0
    % Solve system using LU Factorization
    x = obj.U \ ( obj.L \ b );
    cyclesUsed = 0;
    normResidual(1) = norm(b - obj.A*x);
    return % Leave the function
end
% ------------- Compute the first residual -------------
A = obj.A; % Matrix of the first level
r = b - A*x0; % First residual
nrmR = norm(r); % Norm-2 of the residual
normResidual(1) = nrmR;
nrm0Tol = obj.tol * nrmR; % Tolerance relative to the
    % first residual
% ------------------- Star V-cycles -------------------
cyclesUsed = 0;
while cyclesUsed < obj.numVCycles && nrmR >= nrm0Tol
    % ------------ Go downward in V-cycle ------------
    for l = 1 : numLevels
        A = obj.MA{l}; % Get matrix A_l
        x = zeros( size(r) ); % Create solution x_l
        % ---------------------------------------------
        x = forwardGaussSeidel(obj, r, x, l);
        % ---------------------------------------------
        % Stores new vectors
        approxim{l} = x;
        residual{l} = r;
        % Restrict residual
        r = obj.MR{l} * ( r - A*x );
    end
    % --- Perform LU Factorization in the last level --
    approxim(numLevels+1) = obj.U \ ( obj.L \ r );
% ------------- Go upward in V-cycle -------------
    for l = numLevels : -1 : 1
        A = obj.MA{l}; % Get matrix A_l
        x = approxim{l}; % Get vector x_l
        r = residual{l}; % Get vector r_l
        % Prolonging residual
        x = x + obj.MP{l} * approxim{l + 1};
        % ---------------------------------------------
        x = backwardGaussSeidel(obj, r, x, l);
        % ---------------------------------------------
        approxim{l} = x; % Stores new approximation
    end
% End of V-cycle
r0 = b - A*x; % Create new residual for the first level, but this does not modify the original
nrmR = norm(r0);
normResidual(cyclesUsed + 2) = nrmR;
cyclesUsed = cyclesUsed + 1;
end
diagMethod();
end % End Public Methods

% -------------------------------------------
% PRIVATE METHODS
% -------------------------------------------
methods (Access = private)
% This function marks the master and slave nodes for a given matrix A of order m.
% -------------------------------------------
% marked: is a m-size vector which contains 0 on its ith component if node i is slave. Otherwise, node i is master, and the ith component of the vector contains a new numbering for node i.
% sizeM: is the number of master nodes.
% -------------------------------------------
function [marked, sizeM] = coarsening(obj, A)
    m = length(A); % Order of the matrix
    % -------------------------------------------
nnzRow = zeros(1, m); % Number of non-zero elements of A by row
    for i = 1 : m
        nnzRow(i) = nnz( A(i,:) );
    end
    [~, idxRow] = sort(nnzRow); % Sort indices according to the number of non-zero elements
    % First, every node is marked with a negative number, in order to indicate that the node is unmarked.
    marked = -1 * ones(1, m);
    sizeM = 0;
    % Using the symmetry of A, we follow the columns of A:
    for j = idxRow
        c = A(:,j); % Define the jth column of A
        I = find(c); % Find indices of non-zero elements
        if marked(j) == -1 % Unmarked node
            Node j has not been marked yet. That’s why it will be marked as master and its neighboring nodes will be slaves.
% Mark its neighbors as slaves:
marked(I) = 0;
% Mark as master:
sizeM = sizeM + 1; % A master node was found
marked(j) = sizeM; % New numbering for node j
end
end
% Forward Gauss-Seidel iteration.
% b: is the right-hand side vector.
% x: is the initial vector.
% l: is the corresponding level.
% function x = forwardGaussSeidel(obj, b, x, l)
numSteps = obj.steps(l);
for k = 1 : numSteps
    x = obj.MFGS{1} * ( b - obj.MU{l}*x );
end
end
% Backward Gauss-Seidel iteration.
% b: is the right-hand side vector.
% x: is the initial vector.
% l: is the corresponding level.
% function x = backwardGaussSeidel(obj, b, x, l)
numSteps = obj.steps(l);
for k = 1 : numSteps
    x = obj.MBGS{1} * ( b - obj.ML{l}*x );
end
end
end % End Private Methods
% END OF THE FILE