Manual for APS-MG (A-Posteriori-Steered Multigrid)

Ani Miraçi, Jan Papež

version of January 17, 2023

Abstract

The package of MATLAB scripts is an implementation of a geometric multigrid method that is steered by an a posteriori estimator of the algebraic error. This APS-MG algebraic solver treats linear systems arising from arbitrary-order $p \geq 1$ conforming finite element discretizations of second-order elliptic diffusion problems on triangular meshes in two space dimensions. One solver iteration corresponds to a geometric multigrid V-cycle with zero pre- and one post-smoothing step via block-Jacobi (overlapping additive Schwarz/local patchwise problems). This solver, developed in the framework of [Miraçi, Papež, Vohralík. SIAM J. Sci. Comput. (2021)], contracts the algebraic error independently of the used polynomial degree (p-robust). Moreover, two adaptive multigrid extensions are implemented: adaptive choice of the number of smoothing steps per level and adaptive local smoothing. The latter was developed in the framework of [Miraçi, Papež, Vohralík. Comput. Methods Appl. Math. (2021)]. The code is available at `https://github.com/JanPapez/APS-MG`

Contents

1 Purpose of the code

Consider the following 2D second-order diffusion problem

$$\begin{aligned} \nabla \cdot \mathbf{K} \nabla u &= f \quad \text{in } \Omega \subset \mathbb{R}^2, \\ u &= u_{\mathrm{D}} \quad \text{on } \partial\Omega, \end{aligned} \tag{1}$$

where $f \in L^2$ is the source term, the domain Ω is an open bounded polygon with a Lipschitz-continuous boundary, and the Dirichlet boundary condition u_{D} is continuous on $\partial\Omega$. We assume that the symmetric, uniformly positive definite tensor $\mathbf{K} \in [L^\infty(\Omega)]^{2\times 2}$ is piecewise constant and the possible discontinuities can

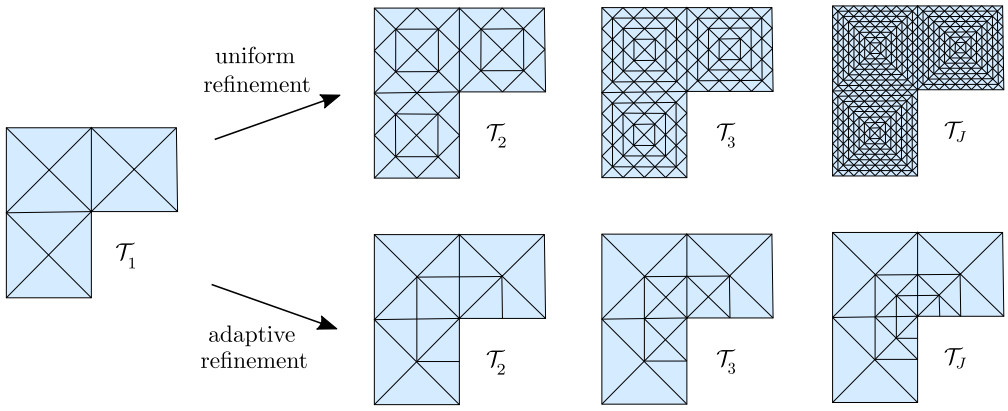


Figure 1: Illustration of mesh hierarchies that can be handled by the code

be matched by a triangulation of the domain (i.e. that the diffusion tensor is constant on each element of the triangulation).

To discretize the problem, we consider a computational mesh $\mathcal{T}_J$, $J \geq 1$, of the domain Ω arising from a sequence of nested conforming triangulations[1] $\mathcal{T}_1$, $\mathcal{T}_2$, ..., $\mathcal{T}_J$. This hierarchy is constructed by $\mathcal{T}_{j+1}$ being a refinement (either uniform or local) of $\mathcal{T}_j$; an illustration is given in Figure 1. Note that one can also consider a single-level mesh hierarchy, i.e., $J = 1$.

Next, we fix a polynomial degree $p \geq 1$. We seek the conforming finite element (FE) approximation u_J of (1) in the discrete space

$$V_J := \{v_J \in H_0^1(\Omega) \ ; \ v_J|_K \in \mathbb{P}^p(K) \quad \forall K \in \mathcal{T}_J\}.$$

Thus, the resulting problem is now discrete and consists in searching for $u_J \in V_J$ such that

$$(\mathbf{K}\nabla u_J, \nabla v_J) = (f, v_J) \quad \forall v_J \in V_J. \tag{2}$$

A-Posteriori-Steered Multigrid (APS-MG) is developed to treat the linear system for a hierarchical structure fitting within the pre-computed mesh hierarchy. Moreover, it allows for more flexible hierarchies by introducing a sequence of non-decreasing polynomial degrees $\{p_j\}_{j=1}^J$, i.e., $1 = p_1 \leq p_2 \leq \cdots \leq p_J = p$, and the level-wise conforming finite element spaces

$$V_j := \{v_j \in H_0^1(\Omega) \ ; \ v_j|_K \in \mathbb{P}^{p_j}(K) \quad \forall K \in \mathcal{T}_j\}.$$

The assumptions on triangulations and polynomial degrees ensure that $V_1 \subset V_2 \subset \cdots \subset V_J$ and that the functions in V_j are continuous, $j = 1, \ldots, J$.

The core feature of APS-MG is that its construction yields simultaneously the built-in a posteriori estimator for the algebraic error. It is in this sense that we refer to this multigrid solver as a-posteriori-steered. This feature allows to extend the approach to two different adaptive multigrid solvers. Full details on the construction of the a-posteriori-steered multigrid solver and its two adaptive variants are given in Section 6. The advantages of employing APS-MG are as follows:

- the contraction of the algebraic error is robust with respect to the polynomial degree p of the underlying finite element discretization
- the built-in a posteriori estimator for the algebraic error is p-robustly efficient, i.e., it provides a two-sided p-robust bound on the algebraic error ; while the package only includes the algebraic solvers, the estimators could be easily obtained as additional output from the solver routines; see [1] and [2] for the details

[1] A conforming triangulation consists of triangles such that neighboring elements share either a common face or a vertex.

- optimal step-sizes are used in the error correction stage of multigrid, making the method completely parameter-free
- the hierarchy of meshes used in the multigrid structure can be uniformly-refined or a graded one generated by adaptive bisection-based refinement
- the hierarchy of finite element spaces is flexible in the sense that the user can employ an increasing sequence of polynomial degrees associated to the mesh hierarchy
- two adaptive multigrid extensions are implemented: adaptive choice of the number of smoothing steps per level, cf. [1], and adaptive local smoothing, cf. [2]

The code was intentionally written to allow modifications in a simple way and to be used as a proof of concept for research papers. It does not aim at providing the fastest computational time or the lowest memory usage. Therefore, some data are saved more than once and/or are not necessary for the actual computation. On the other hand, whenever possible, advantages of MATLAB language (such as so-called vectorization, quick matrix operations, and built-in functions) are used.

Finally, we note that a set of experiments is provided with the package reflecting upon numerical results presented in [1, 2].

2 Requirements

The code has been used with MATLAB R2019b. However, we believe that it can be used also with other versions. The mesh generation and plotting requires MATLAB PDE Toolbox.

APS-MG is installed by simply copying the files into a desired folder.

3 What is in the package

The package includes the following algebraic solvers

- `p_robust_MG01_solver`: the p-robust non-adaptive version of a-posteriori-multigrid solver from [1]. This is a geometric multigrid solver whose iteration is given by a V-cycle with zero pre- and one post-smoothing. The coarse solve is done through lowest-order discretization and is thus inexpensive. The smoothing on each level can be seen as an additive Schwarz method/block-Jacobi associated to patches of elements. Optimal step-sizes are employed in the multigrid error correction stage.
- `p_robust_MG0adapt_solver`: the adaptive number of post-smoothing steps extension of APS-MG from [1]
- `p_robust_MG01_locallyadaptive_solver`: the adaptive local smoothing extension of APS-MG from [2]
- `run_paper_examples`: a script running several experiments presented in [1, 2] of the above solvers
- auxiliary scripts for initialization of data structures and plotting the results

If you lack any functionality, please feel free to contact us to discuss if this functionality is available.

4 Data structures

Solution

The data structure `solution` contains information about the problem to be solved – the solution (if known), the right-hand side, and the geometry.

u: solution as a MATLAB function handle. If unknown, use a function which prescribes the Dirichlet boundary condition u_{D}.

`ugrad:` the gradient of the exact solution as a function handle. It must be given only to evaluate the discretization and total errors on each element; set zero otherwise.

`f:` the right-hand side as a MATLAB function handle.

`fh_coef:` a structure which contains, for each level, the coefficients of the elementwise p-order polynomial representation of the right-hand side f. Filled in the initialization by `project_f.m`

`name:` name of the test problem, can be used, for instance, in saving the figures automatically.

`geometry:` MATLAB PDE Toolbox specification of the geometry.

`diamOmega:` diameter of the domain Omega; not used for most of the calculations.

`tenzorsqrt:` square root of 2×2 diffusion coefficient $\mathbf{K}$ for each SUBDOMAIN – the numbering of subdomain is specified in `solution.geometry` and for each element, say `index` on level `j`, it is saved in `meshdata(j).elements(4,index)`.

Meshdata

The data structure `meshdata` contains the information about the meshes (triangulations) and the interlevel operators, where the number of levels in the hierarchy is `J`. Note that since we work with flexible hierarchies (where from one level to the next, not only the mesh is refined but possibly the polynomial degree also increases) the prolongation of an existing level to the new one consists in: first an h-interpolation (mesh-related), then a p-interpolation (high-order-related). In our multigrid implementation, the restriction is given as a transpose of the prolongation operator. The `coord`, `edges`, and `elements` are the structures adopted from MATLAB PDE Toolbox. `meshdata(j)` consists of

`coord:` coordinates of the vertices of the level index `j` mesh (for each vertex, the column contains x and y coordinates).

`edges:` array of information about the edges as used by MATLAB PDE Toolbox.

`elements:` array of information about the elements as used by MATLAB PDE Toolbox. In particular, for each element, the first three entries of the column contains the indices of the mesh vertices numbered anticlockwise.

`nc:` number of vertices of the mesh.

`ne:` number of elements of the mesh.

`I:` interpolation matrix for p-order functions from the previous level to the current one.

`I1:` interpolation matrix for first-order (piecewise affine) functions from the previous level to the current one.

`refine:` information about the refinement of the current level to the next one. Each row corresponds to an element, first entry gives how the element was refined (zero for no refinement, 1 for uniform refinement into four elements, 2–4 for other variants) and the other entries give indices of the new elements (using the numbering on the next level).

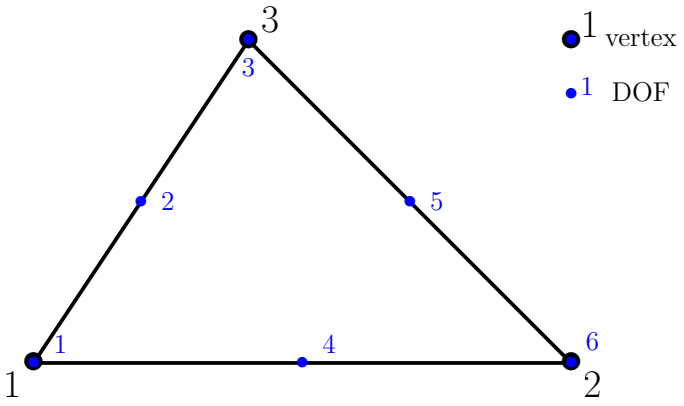


Figure 2: Illustration how the DOFs are locally numbered for each element in `DOF(j).elementFEM`, here for $p = 2$. The first vertex coincides with the first DOF. Please note that for $p = 1$, the DOFs are numbered clockwise while the vertices of the element in `meshdata(j).element` are numbered anticlockwise.

Degrees of freedom

The data structure `DOF` is associated with each level. Then `DOF(j)` consists of

`m`: polynomial order p of FEM approximation.

`nFEM`: number of FEM degrees of freedom per element, `nFEM` $= \frac{p}{2}(p+1)$.

`lastFEMindex`: total number of FEM degrees of freedom.

`BC`: information if the given DOF is on boundary (`DOF(j).BC(i) = -1`) or not (`DOF(j).BC(i) = 0`).

`freenodes`: array containing indices of DOFs that are not on the boundary (their value is not prescribed by the boundary condition).

`dirichlet`: array containing indices of DOFs that are on the boundary (their value is prescribed by the boundary condition).

`dirichletvalues`: sparse array containing the values of `dirichlet` DOFs prescribed by the boundary condition.

`elementFEM`: indices of the DOFs associated to each element. They are given in the columns of the array `elementFEM` with the numbering of the DOFs illustrated in Figure 2.

`freenodesnumbering_toglobal`: array for passing from global indexing of DOFs to "local" numbering in the array `DOF(j).freenodes` .

To handle the possible varying polynomial degrees over the levels, we keep two structures, `DOF1` and `DOF2`. The structure `DOF1` corresponds to the polynomial degree from the previous level but represented on the current mesh and `DOF2` corresponds to the current level polynomial degree on the current mesh.

Reference element

In the code, the basis function values are evaluated using the affine transformation from the reference triangle. The data structure `ref_triangle` keeps the information corresponding to the reference triangle, namely

`quadrature`: a structure consisting of coordinates of quadrature nodes (arrays `X` and `Y`), the associated weights (`Weights`) and the number of quadrature points (`s`). The quadrature is Gaussian with the algebraic degree (the order of polynomials that are integrated accurately) equal to $2p + 2$.

`FEMvalue`: value of the FEM basis functions in the quadrature nodes; array of size = # quadrature nodes × # basis functions. We consider lagrangian basis functions (i.e. each basis function is of exact order p, takes value one in an associated node and vanishes in the other nodes) with respect to a non-uniform grid with Gauss–Lobatto–Legendre node distribution. This is done as in Warburton [3] in order to improve the accuracy for higher polynomial orders.

`FEMgrad`: gradient of the FEM basis functions in the quadrature nodes, for each basis function, two consecutive columns are used ; array of size = # quadrature nodes × 2x# basis functions.

`FEMmassmatrix`: the mass matrix of FEM basis associated with the reference triangle. Mass matrix associated to any element is the same up to a constant.

Elements

The data structure `element` keeps, for each level, the stiffness matrices associated to each element of the triangulation. The matrices in `element(j)` are stored as a 3D array, the last dimension corresponds to the index of the element. This structure is needed if the algebraic error is evaluated elementwise and/or to construct local matrices for block smoothers.

Algebraic problems

The (global) stiffness matrices, right-hand sides, and the algebraic solutions on each level are stored in the cells `A`, `F`, and `x` respectively. The vectors `x{j}` are computed as `x{j} = A{j}\b{j}` in `initialize.m`. If the exact solution is not necessary, this step can be avoided — then `x` only represents the boundary condition and vanishes in the indices corresponding to the rest of the domain.

Patches

This structures contains information related to patches of elements that share one common vertex. In particular, `elements` gives elements forming the patch, `FEM0indices` contains the indices of the degrees of freedom inside the patch (i.e. not on the boundary of the patch/boundary of the domain), `hatfunction` gives values of the hat function associated to the patch (i.e. piecewise-affine function taking value one at the vertex whose patch we are considering, and zero in all neighboring vertices) in all the DOFs inside the patch. Finally, we store in `loccholfactor` the Cholesky factor of the local stiffness matrix associated to the patch.

Vectorization of block smoothers

To efficiently implement (overlapping) block smoothers in MATLAB, we consider the structure `extended`. The idea is to expand the algebraic residual vector into a longer vector (with repetitions) where the *non-overlapping* residual vectors of each patch are appended one after the other. This way it becomes easier to multiply with a block diagonal matrix representing local solves, as well as to multiply with a (long) vector representing local weighting by hat functions, and to sum together the contributions coming from different overlapping patches which correspond to the same degrees of freedom.

5 Initialization (offline phase)

The initialization sets up the problem for the chosen test problem, prepares the data structures of the chosen hierarchy, i.e., nested meshes, nested finite element spaces, assembly of the levelwise linear systems, and patches. These data structures to be initialized are global variables. A pseudocode for the multilevel initialization is given in Algorithm 1 and the key parameters to be set prior to the initialization script are:

`m`: polynomial degree of the approximation; this is a non-decreasing sequence of length `J` (and `J` is set implicitly from `m`)

`problem_number`: choice of the test problem, several standard test problems are available

`Hmax`: maximal diameter of the elements in the initial Delaunay mesh

`mesh_uniformity`: marking parameter for adaptive mesh refinement. The choice `mesh_uniformity = 1` means uniform mesh refinement on all the levels. Otherwise, it corresponds to a standard Dörfler marking parameter [4].

Algorithm 1 Pseudoalgorithm of the initialization

set the initialization parameters `m`, `problem_number`, `Hmax`, `mesh_uniformity`
call `initialize.m`, which involves the following steps:
  call `solution_def` to set up `solution` data structure
  for the coarsest level $j = 1$:
  call `fill_meshdata` to set up initial mesh in `meshdata(1)`
  call `fill_DOF1`, `fill_DOF2` to set up `DOF1(1)`, `DOF2(1)`, respectively
  call `fill_ref_element` to set up `ref_element{1}`
  call `project_f_varp` to compute the piecewise polynomial representation of source f on level $j = 1$
  assemble the stiffness matrix, right-hand side, and the algebraic solution `A{j}`, `F{j}`, `x{j}`
  for finer levels $j = 2 : J$ **do**
    call `refinemesh_mine` to refine the mesh and set up `meshdata(j)`
    call `fill_DOF1`, `fill_DOF2`, `fill_ref_element`, `project_f_varp`
    construct the interpolation matrices (and store them in `meshdata(j)`)
    assemble the stiffness matrix `A{j}`, right-hand side `F{j}`, and the algebraic solution `x{j}`
  end for
  call `fill_patch` to set up `patches` structure
  for levels $j = 2 : J$ build the structure `extended` and multigrid prolongation matrices `prolong_matrices`

Now we comment on components of the initialization script in more detail. We omit the description of `solution_def`, `fill_ref_element`, and `fill_patch` as their functionality is straightforward from the description of `solution`, `ref_triangle`, and `patches` data structures in Section 4.

Script `fill_meshdata`

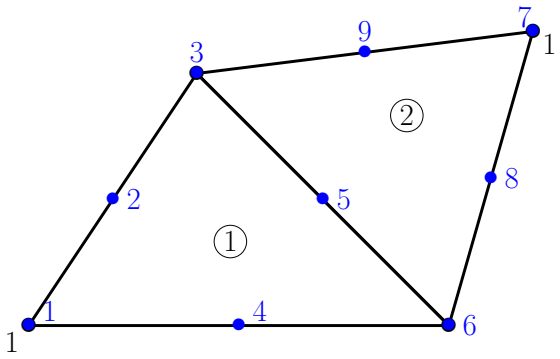
The script constructs an initial mesh with the size prescribed by the parameter `Hmax` and the geometry specified by `solution.geometry`. The construction uses the PDE Toolbox function `init_mesh` that creates a Delaunay triangulation. Then we permute the local indices of the vertices in each of the elements such that the longest edge is opposite to the third vertex.

Scripts `fill_DOF1`, `fill_DOF2`

This script sets up global to local (per element) indexing of the degrees of freedom. We go through all the elements and consecutively order DOFs not yet indexed or use the already assigned indices. This is illustrated in Figure 3 for two elements and $p = 2$. Recall that the local indexing of the DOFs on an element is illustrated in Figure 2 (also for $p = 2$).

Script `project_f_varp`

The script computes the piecewise polynomial representation $f_j \in \mathbb{P}^{p_j}(\mathcal{T}_j)$ of the source term f with respect to the given level (mesh) and polynomial degree, such that $(f_j, v_j) = (f, v_j)$ for all $v_j \in V_j$. The oscillation term $\|f - f_j\|^2$ is estimated using a quadrature rule of polynomial degree $2p_j + 2$.




```
DOF(j).elementFEM = [ 1 7
                      2 8
                      3 6
                      4 9
                      5 5
                      6 3 ]
```

Figure 3: Illustration of global to local indexing of the degrees of freedom in `fill_DOFs` for two elements and $p = 2$. The global DOF indices are given in blue. The black '1' indicates the first vertex of the elements and the circled numbers give the indices of the elements. For local DOF numbering see Figure 2.

The local (elementwise) coefficients of f_j w.r.t. the FEM basis are stored in `solution.fh_coef{j}`.

Assembling the algebraic system

The stiffness matrix `A{j}` and the right-hand side `F{j}` are assembled, element by element, using the script `globalstiffmatrix_varp`. The exact algebraic solution `x{j}` is then computed in the `initialize` script using MATLAB backslash solver. Whenever the exact solution is not needed (it is used, for example, for adaptive mesh refinement), the backslash can be avoided.

Marking the elements to refine

When the uniform refinement is used, we simply mark for refinement all the elements. Whenever the parameter `mesh_uniformity` is set smaller than 1, an adaptive refinement is employed. In such a case (and in order to ensure reproducibility of the experiments), instead of an error estimator, we require the knowledge of the exact (infinite-dimensional) solution u and the Galerkin solution $u_J \in V_J$ to evaluate, using the script `evaluate_error_on_elements`, the energy norm of the discretization error $\|\mathbf{K}^{1/2}\nabla(u - u_J)\|_{K_j}$ on each element $K_j \in \mathcal{T}_j$. Then, following the standard Dörfler marking strategy [4], we mark the smallest subset of elements such that they represent `mesh_uniformity` percentage of the discretization error over the whole domain Ω.

Script `refinemesh_mine`, `refinemesh_mine_nvb`

This script is a modification of a PDE Toolbox script `refinemesh` that refines the marked elements of the mesh and the neighboring ones if necessary to maintain conformity of the mesh. We modified the local indexing of vertices and of the new elements and remove the permutation of local indices after the refinement.

The refinement allows four situations depending on which edges are refined (and the fifth situation where none of the edges is refined, the element then stays as it is). These situations and the way how the new elements are created and indexed are depicted in Figure 4.

The so-called Newest-Vertex-Bisection (NVB) refinement, see, e.g., Sewell [5] and Mitchell [6], Traxler [7], Stevenson [8], differs from the above one only in the way how the element is refined into four elements, i.e. when all three edges are refined. The modification of the code to have fully NVB refinement is given in `refinemesh_mine_nvb`.

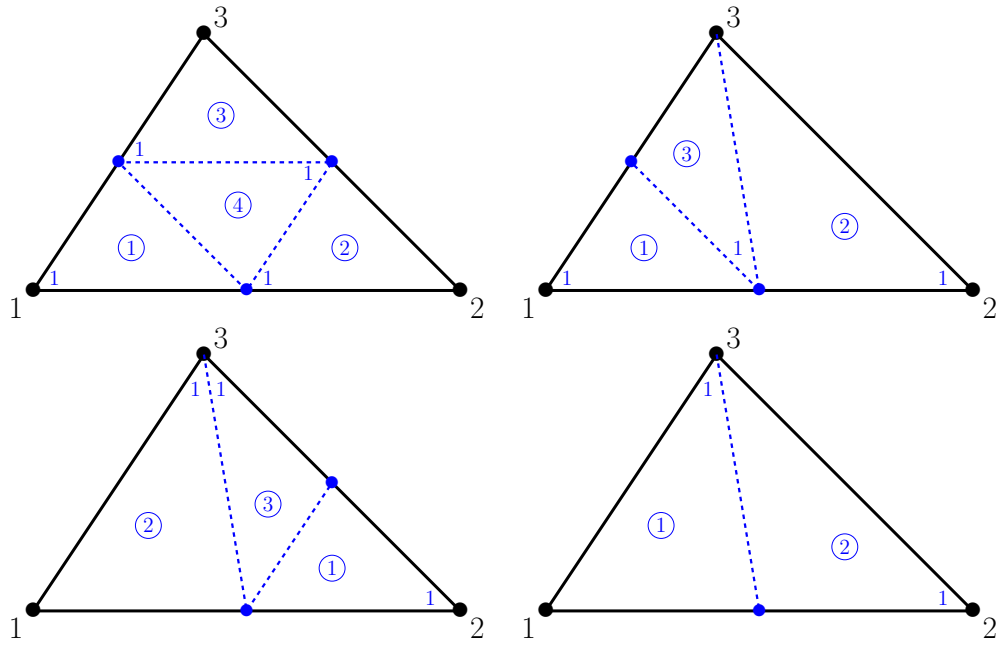


Figure 4: Mesh refinement variants, depending on which edges are refined. The circled number gives the local index of newly constructed elements. The blue '1' indicates which vertex is in the new triangles indexed as the first one, the numbering of the vertices in the triangles is always anticlockwise. In `meshdata(j).refine`, these variants (read along the lines) are numbered 1–4.

Construction of interpolation matrices

The script `interpolation_matrix_varp` (respectively `interpolation_matrix_varp_NVB` if nonuniform mesh refinement is used) builds a matrix to interpolate a vector from a level $j-1$ onto j. The interpolation corresponds to the mesh refinement and not a possible change (increase) of the polynomial degree on the level j. This means that, denoting by I_{j-1}^{j} the interpolation matrix and by Φ_j^p the Lagrangian FEM basis of degree p associated with level j, we have $\Phi_j^p(I_{j-1}^{j}\mathbf{v}_{j-1}) = \Phi_{j-1}^p\mathbf{v}_{j-1}$ for any vector $\mathbf{v}_{j-1}$ of a proper length.

Construction of multigrid prolongation and restriction matrices

Prolongation multigrid matrices correspond both to mesh (h-)refinement and the increase of polynomial degree (p-refinement) if $p_j > p_{j-1}$. We construct them by multiplying the interpolation matrix (that is associated with two consecutive meshes) with a polynomial interpolation matrix on a single level that corresponds to the change of the polynomial degree. The polynomial interpolation matrix is assembled by calling the script `p_interpolation` in the initialization. The restriction matrix is given as a transpose of the prolongation matrix.

6 Computation (online phase)

We now describe the phase where the actual computation of the iterates approximating the discrete solution u_J of (2) by the multigrid solvers is being performed using the problem and data from the offline phase.

The solvers are initialized by zero initial vector so that the initial residual is equal to the right-hand side `F{J}` of the algebraic system. A modification for non-zero initial guess is possible. The solver iterates until a maximum number of iterations fixed by `maxiter` (that is by default set to 100) is reached or the

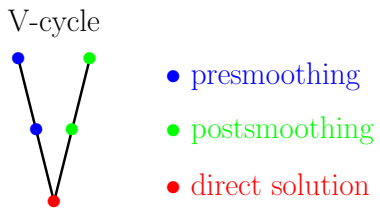


Figure 5: Illustration of the V-cycle in multigrid for a hierarchy of three meshes.

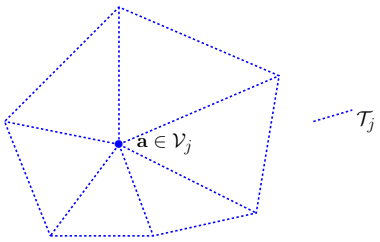


Figure 6: Illustration of a patch

relative residual drops below 10^{-5}. To use another stopping criterion or tolerance, the code of the solver can be easily modified.

All the solvers are based on a so-called V-cycle, illustrated in Figure 5. In our approach, the number of pre-smoothing steps is zero (only level-wise restriction of the algebraic residual) and the smoothing is done by block-Jacobi methods. We consider the variant of block-Jacobi smoothers associated to patches of elements sharing a vertex; see Figure 6 for an illustration. Note that computations with block smoothers are more implementationally demanding than pointwise smoothers. Principally, there are two variants. First, the matrix of the smoother on each level is assembled in the offline phase. Then the computation (with our code in MATLAB) is faster but more memory is required to store these matrices. Second, smoothing is implemented using a for-cycle over vertices of the mesh. For each vertex, the local matrix associated with a patch is extracted on-the-fly from the global stiffness matrix (or assembled using the local element matrices) and the associated local problem is solved. Such smoothing is significantly slower (due to additional work done in each iteration and due to slow for-cycle implementation in MATLAB) but significantly less memory is needed. In this script, the first variant is chosen. In all solvers variants below, the new approximation is computed by adding to the old approximation an update (residual lifting) constructed by the block smoothers.

6.1 `p_robust_MG01_solver`

This script implements the non-adaptive solver from [1]. In each iteration, the update/residual lifting is computed by the script `smoother_bJ`. Therein, the V-cycle first restricts the algebraic residual on each level (no pre-smoothing step) thanks to the interlevel operators, solves a coarse-grid global residual problem, and then proceeds to the ascending branch of the V-cycle with one post-smoothing step. On each level, we solve patch problems/smooth by block Jacobi (an additive method which can be naturally parallelized) and sum these contributions together to give an error correction. Moreover, we compute here the optimal step-size in the direction of the correction before interpolating it to the next level (the method is thus multiplicative in the levels). The resulting update at the finest level ensures that the new approximation will contract the algebraic error p-robustly.

6.2 p_robust_MG0adapt_solver

In this script, the adaptive version of the solver from [1, Section 7] is implemented. In each iteration, the update is computed by the script `smoother_bJ_adaptnumber`. The construction is analogous to that of the solver `p_robust_MG01_solver`, but an *adaptive number of post-smoothing steps* is employed in the V-cycle. Having more than one post-smoothing step means that the local patch problems are smoothed once, a first error correction and associated optimal step-size is constructed, then the algebraic residual of the level is updated and the procedure possibly starts again by doing another sweep of the patches. The condition for performing more than one post-smoothing step, cf. [1, Definition 7.1], relies on whether the latest smoothing step considerably (by user-prescribed adaptivity parameter `theta`) decreased the algebraic error with respect to previous smoothings (in that case, another smoothing step can be beneficial) or not (in that case, it seems to be better to proceed to the next level).

The number of post-smoothing steps on each level can be limit by the constant `maxlocsmooths` that is by default set to 5. This value has never been reached in our experiments.

6.3 p_robust_MG01_locallyadaptive_solver

In this script, the adaptive multigrid solver from [2] is implemented. In each iteration, the update is computed in two stages, a first V-cycle by calling the script `W_full_smoother` and a second inexpensive adaptive V-cycle by `W_adapt_smoother`. In the first V-cycle, all patches are smoothed once, similarly to the non adaptive solver `p_robust_MG01_solver`. Additionally, `W_full_smoother` returns computable and localized (levelwise and patchwise) estimators of the algebraic error. These estimators are used together with a bulk-chasing criterion (requiring a user-prescribed adaptivity parameter `theta`) to determine in which patches the algebraic error is high. Once the problematic patches are identified, a test from [2, Section 3.1.6] (requiring a second adaptivity parameter `gamma`) runs to guarantee that the second adaptive V-cycle of `W_adapt_smoother`, which smoothes only on the problematic patches, will also contract the algebraic error p-robustly.

For this solver variant, we moreover allow the smoother to run for two possible block Jacobi methods. The first, coinciding with an additive Schwarz (AS) method, is the one used in `p_robust_MG01_solver` and `p_robust_MG0adapt_solver` and relies on summing the overlapping local patch contributions over the vertices on each level. The second, called weighted restricted additive Schwarz (wRAS), relies on summing the local patch contributions *after* being multiplied by the hat function associated with the vertex (hat functions form a partition of unity) and then summed together. For more tests of the solver smoothing variants, see also numerical tests of [9]. In our experiments, wRAS performed always better than AS and whenever the theoretical study allows it, wRAS smoothing is employed.

One can also set an iteration where the algebraic error distribution over levels and patches and its local estimation is plotted. To do so, set `plot = [1, iter]`, where `iter` is the required iteration.

6.4 run_paper_examples

In this script, six different experiments from [1, 2] are given as examples on how to run the code for a set of input parameters (test problem, polynomial degree per level, initial mesh-size, type of mesh refinement, and potential adaptivity parameters), which can easily be modified. The first two tests correspond to the non-adaptive version of multigrid, one for a uniform mesh refinement hierarchy and the second for a graded one. Next, an example on the adaptive number of smoothing steps solver is given. The following two tests emphasize (for two different test problems) the performance of the locally adaptive solver in identifying the the regions with high algebraic error. The last test is an example of the locally adaptive solver.

To conclude, we present a few remarks:

- Experiments with graded meshes use the exact discretization error in the bulk-chasing marking criterion. This means that they can only work when the PDE solution is known, which is, for example, not the case of the skyscraper test problem (`problem_number = 12` or `13`).

- Using the adaptive local smoothing solver `p_robust_MG01_locallyadaptive_solver` with adaptivity parameter $\theta = 1$ does *not necessarily* lead to the same result as `p_robust_MG01_solver`, since the smoothing is also adaptively chosen between the additive Schwarz or the weighted restricted additive Schwarz method.
- When running `p_robust_MG01_locallyadaptive_solver` for graded hierarchies, one can note that the patches selected for local smoothing are situated around new vertices and immediate neighbors. Choosing patches related to the refinement is in agreement with existing literature, see e.g. [10, 11, 12] on optimal solvers for bisection-generated meshes. Employing such local smoothings, leads to robust behavior of the solver with respect to number of levels in the mesh hierarchy.

References

[1] A. Miraçi, J. Papež, and M. Vohralík. A-posteriori-steered p-robust multigrid with optimal step-sizes and adaptive number of smoothing steps. *SIAM J. Sci. Comput.*, 43(5):S117–S145, 2021.

[2] A. Miraçi, J. Papež, and M. Vohralík. Contractive local adaptive smoothing based on Dörfler's marking in a-posteriori-steered p-robust multigrid solvers. *Comput. Methods Appl. Math.*, 21(2):445–468, 2021.

[3] T. Warburton. An explicit construction of interpolation nodes on the simplex. *J. Engrg. Math.*, 56(3):247–262, 2006.

[4] Willy Dörfler. A convergent adaptive algorithm for Poisson's equation. *SIAM J. Numer. Anal.*, 33(3):1106–1124, 1996.

[5] E. G. Sewell. *Automatic generation of triangulations for piecewise polynomial approximation.* ProQuest LLC, Ann Arbor, MI, 1972. Thesis (Ph.D.)–Purdue University.

[6] W. F. Mitchell. Adaptive refinement for arbitrary finite-element spaces with hierarchical bases. *J. Comput. Appl. Math.*, 36(1):65–78, 1991.

[7] Christoph T. Traxler. An algorithm for adaptive mesh refinement in n dimensions. *Computing*, 59(2):115–137, 1997.

[8] Rob Stevenson. The completion of locally refined simplicial partitions created by bisection. *Math. Comp.*, 77(261):227–241, 2008.

[9] A. Miraçi, J. Papež, and M. Vohralík. A Multilevel Algebraic Error Estimator and the Corresponding Iterative Solver with p-Robust Behavior. *SINUM*, 58(5):2856–2884, 2020.

[10] Long Chen, Ricardo H. Nochetto, and Jinchao Xu. Optimal multilevel methods for graded bisection grids. 120(1):1–34.

[11] Ralf Hiptmair, Haijun Wu, and Weiying Zheng. Uniform convergence of adaptive multigrid methods for elliptic problems and Maxwell's equations. 5(3):297–332.

[12] Jinbiao Wu and Hui Zheng. Uniform convergence of multigrid methods for adaptive meshes. 113(C):109–123.