MULTIGRID FOR FENICSX

ABSTRACT. Multigrid methods shape the method of choice when a self containing family of triangulations of a FEM discretization is at hand to achieve an optimal order solver. Optimal order meaning linear complexity in the number of DOFs, which is the same for the assembly of the associated linear system of equations, making this a completely astonishing and non-trivial result of the theory of multigrid methods.

But also in applications this outperforms classical, algebraic based, solver routines when it comes to large scale simulations. Although, the flexibility of modern AMG implementations do not require any such structure of meshes, multigrid methods remain of great interest not only in academia. So an implementation in the DOLFINx framework should allow for versatile extensions and applications.

To achieve this, multiple components of the data structure associated with mesh, geometry and topology require close analysis and a fundamentally new feature, the necessary prolongation and restriction operators, need to be implemented. This process is split into multiple isolated, but chronologically dependent phases.

1. Technical Details

The multigrid method is an optimized solving routine for discretized variational problems. Let us shortly state the general setting we consider. For this proposal we stick to the setting of linear variational problems on an open bounded d-dimensional domain $\Omega \subset \mathbb{R}^d$, $d \in \{1, 2, 3\}$. So for a bilinear form $a : H^1(\Omega) \times H^1(\Omega) \to \mathbb{R}$ and a linear form $b : H^1(\Omega) \to \mathbb{R}$ we are interested in approximating the solution of the variational problem of finding $u \in H^1(\Omega)$ s.t.

$$a(u, v) = b(v) \qquad \forall v \in H^1(\Omega).$$

Remark. The extension of multigrid methods to non-linear problems is outside the reach of this project, but nevertheless a possible extension relies on an implementation of a linear method, as one then considers the (linear) variations as descent directions of the non-linear energy.

Definition. A *mesh* is a decomposition $\mathcal{T} = \{T_1, \dots, T_N\}$ of Ω into simple polyhedra T_i, here we assume them to be of same type.

Definition. We say a mesh $\mathcal{T}$ is *admissible*, if it holds

(1) it covers Ω, i.e. $\bar{\Omega} = \bigcup_{i=i}^{N} T_i$,
(2) if $T_i \cap T_j = \{x\}$, $x \in \mathbb{R}^d$, then x is a vertex of T_i and T_j and
(3) if $T_i \cap T_j \neq \{x\}$, $x \in \mathbb{R}^d$, then the intersection is a common edge ($d = 2$) or face ($d = 3$) of T_i and T_j.

In the following all meshes are silently assumed to be admissible. The multigrid requires the presence of a family of meshes $\{\mathcal{T}_0, \dots, \mathcal{T}_n\}$ of Ω, which are ordered in the sense of inclusion (vertices are a subset and every edge/face/element is expressible as union of finer ones) as

$$\mathcal{T}_0 \subset \mathcal{T}_1 \subset \cdots \subset \mathcal{T}_n.$$

Remark. The index i of the triangulations $\mathcal{T}_i$ is often called *level*, originating from the case where the $\mathcal{T}_i$ are the product of different uniformly refined meshes.

Every of those triangulations is associated with an approximation space $V_i = V_i(\mathcal{T}_i) \subset H^1(\Omega)$. We will only consider spaces built from same elements, i.e. no p-adaptivity and mixing of elements is considered.

The multigrid method now uses this self contained structure to speed up the solving step of the discrete variational problem

$$a(u_N, v_N) = b(v_N) \qquad \forall v_N \in V_N.$$

This critically depends on the *restriction* $R_i : V_{i+1} \to V_i$ and *prolongation operators* $P_i : V_i \to V_{i+1}$ that transfer functions between these different discretizations. These are assumed to be linear maps and expected to be self-adjoint, i.e.

$$R_i^\star = P_i$$

which for the matrix representation of the mappings simply becomes

$$R_i^T = P_i \in \mathbb{R}^{\dim(V_i)\times\dim(V_{i+1})}.$$

Remark. Self adjoint-ness guarantees that it holds

$$\langle R_i u_{i+1}, u_i\rangle_{V_i} = \langle u_{i+1}, P_i u_i\rangle_{V_{i+1}} \qquad \forall u_i \in V_i,\ u_{i+1} \in V_{i+1}.$$

So it does not matter, wether we compare two different fine functions in terms of restriction in a coarser space, or via a prolongation in the finer space. This is our compatibility requirement on the interaction of the two spaces.

Algorithm 1 The complete multigrid algorithm for a discretized linear variational problem.

```
Require: γ ∈ {1,2}                                   ▷ V- or W-cycle
Require: A_1,...,A_n, A_i ∈ R^{dim(V_i)×dim V_i}     ▷ Stiffness matrix (per level)
Require: S_l : R^{dim V_i} → R^{dim V_i}             ▷ Smoother (per level)
Require: ν_1, ν_2 ∈ N                                ▷ No. of pre-/post-smoothing steps
 1: procedure MGM(u_l, f_l)
 2:     if l is 0 then
 3:         u_l ← A_l^{-1} f_l                       ▷ Solve A_l u_l = f_l exactly
 4:         return u_l
 5:     end if
 6:     for all i ∈ {1,...,γ} do
 7:         u_l ← S^{ν_1} u_l                        ▷ Pre-smoothing
 8:         r_l ← f_l − A_l u_l                      ▷ Compute residual
 9:         r_{l−1} ← R_{l−1} r_l                    ▷ Restrict residual
10:         r_{l−1} ← MGM(0, r_{l−1})                ▷ Recursive MGM on coarser mesh
11:         u_l ← u_l + P_{l−1} r_{l−1}              ▷ Apply prolongated correction
12:         u_l ← S^{ν_2} u_l                        ▷ Post-smoothing
13:     end for
14:     return u_l
15: end procedure
```

So understanding one of the operation also implies the same for its counter part, equivalently regarding the implementation. But at this point it is not clear how these operators may be constructed.

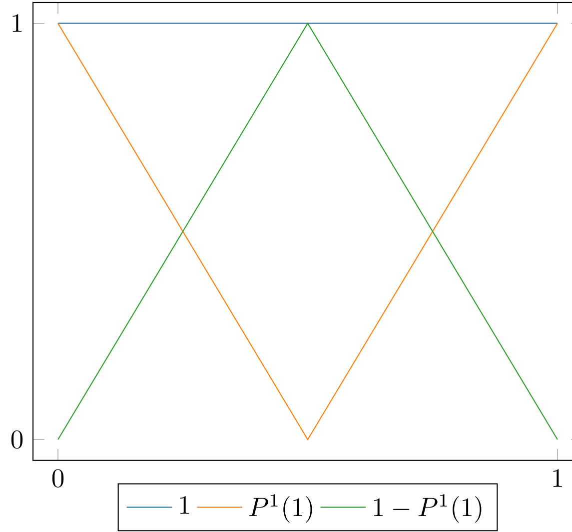

FIGURE 1. Plot of the constant one function in the coarse space, its prolonged version $P^1(1)$ and the difference between the expected prolongation, which is simply the ignored basis function of the finer space.

Let us consider an example. We consider $\Omega = (0, 1)$ and choose classic first order Lagrange elements on $T_0 = \{0, 1\}$ and $T_1 = \{0, \frac{1}{2}, 1\}$. Especially T_1 is produced by uniform refinement of T_0 and $T_0 \subset T_1$ as well as $V_0 \subset V_1$. A first possible prolongation P^1 (the index here does not indicate the level) is given by

$$P^1 = \begin{bmatrix} 1 & 0 \\ 0 & 0 \\ 0 & 1 \end{bmatrix} \implies R^1 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}.$$

At a first glance, this seems like the perfect candidate, especially due to its nice matrix structure which we can definitely find for any family of meshes. We assign every coefficient (DOF) that has a counter part in the finer space to its value in the coarser space. However it is not quite what we need, consider the function $1 = 1 \times x + 1 \times (1 - x) \in V_0$ which becomes after prolongation $1 \times (-2x + 1)|_{[0, \frac{1}{2}]} + 1 \times (2x - 1)|_{[\frac{1}{2}, 1]} \in V_1$ definitely not being the constant one function (see Figure 1), even though $1 \in V_1$.

So an additional requirement, should maybe be the following, for $u_i \in V_i$ it should hold

$$u_i \in V_{i+1} \quad \implies \quad P_i u_i = u_i.$$

For the previous example such a prolongation is given by

$$P^2 = \begin{bmatrix} 1 & 0 \\ \frac{1}{2} & \frac{1}{2} \\ 0 & 1 \end{bmatrix} \implies R^2 = \begin{bmatrix} 1 & \frac{1}{2} & 0 \\ 0 & \frac{1}{2} & 1 \end{bmatrix}.$$

However when we consider the same setting with now a second order Lagrange element this is no longer a reasonable requirement, as there are no functions which

equally exist in both approximation spaces. The piecewise C^0 nature of the finer basis and the C^1 regularity of the basis of the coarse space prohibit this.

This is even worse when we think about how we may recover from this compatibility problem. We now know, that any interaction of the prolongation or restriction will introduce error, and thus optimality of such operators is a hard task and a zoo of operators has emerged that attempts to address this.

For this work, I suggest to stick to the linearized approach, i.e. assume a linear Lagrange element associated (with the DOFs not the vertices) and construct the prolongation and restriction operator to recover the previously suggested compatibility condition. For the previous example, this results in

$$P^3 = \begin{bmatrix} 1 & 0 & 0 \\ \frac{1}{2} & \frac{1}{2} & 0 \\ 0 & 1 & 0 \\ 0 & \frac{1}{2} & \frac{1}{2} \\ 0 & 0 & 1 \end{bmatrix} \implies R^3 = \begin{bmatrix} 1 & \frac{1}{2} & 0 & 0 & 0 \\ 0 & \frac{1}{2} & 1 & \frac{1}{2} & 0 \\ 0 & 0 & 0 & \frac{1}{2} & 1 \end{bmatrix}.$$

The implementation of prolongation or restriction relies on the identification of the neighboring DOFs in the coarser or finer mesh and is never to be implemented using the matrix representation as this inevitably slows the performance down. To make these computations feasible, we restrict for this project to the cases of uniformly and adaptively refined meshes using a refinement rule, that

(1) produces between two levels at most one additional vertex on a previous edge/face (thus limiting the maximal level difference to one) and
(2) we do not allow for hanging nodes.

Now we are ready to start planning an implementation of this.

2. Schedule of Deliverables

The critical approach to make this not trivial implementation of multigrid work is to break it up into reasonable verifiable intermediate checkpoints. This motivates the following timeline.

2.1. **Phase 1 - Algebraic Interpolation and Restriction on Meshes.**

In this phase we aim to implement a single level interpolation, i.e. prolongation and restriction operations, in the C++ core as a strictly algebraic undertaking. Here we aim for a efficient and verifiable minimal working example of the transfer of linear algebra objects, not yet involving any solving, approximation spaces or other variational problem aspects. We do not think about the implications of a FEM space on a given mesh, and if at all think about the classic piece-wise linear hat functions FEM setting for now.

Test cases might include (increasing in complexity, with varying uses of adaptive and uniform refinement strategies),

- interval meshes,
- triangle and quadrilateral meshes,
- tetrahedron, hexahedron, prism and pyramid meshes.

This list is given by the supported elements of Basix and thus DOLFINx.

Also this is the critical phase for making the underlying assumptions precise and defending against inappropriate use of the provided functionality. As for example by calling with non-applicable combinations of meshes or meshes that we do not

know how to produce an interpolation/restriction operator between without further information. This includes edge cases, that should be treated with great caution, and thus unit tested, such as

- empty mesh,
- prolongation/restriction between the same spaces, and
- unusual linear algebra object dimensions.

As a byproduct of this, we identify the fundamentally important data structures and objects that later on need to be managed for a multigrid FEM implementation.

2.2. **Phase 2 - Two Level Multigrid FEM Simulation.** Now we are ready to think about the interconnection of the previous strictly algebraic side of things and a FEM simulation, vectors are now interpreted as coefficient vectors and matrices as stiffness/mass matrices.

We identify the interface required to interact with the C++ implementations of the prolongation and restriction operation and export it to the python module. Also we need to verify and possibly adapt to the use of higher order elements. If this requires further adaptations, the previous unit tests are to be extended or adapted to facilitate the extended feature set.

This phase is to be considered completed, when a simple Poisson problem with a finer and a coarser function space passes for multiple element types, here we refrain from using advanced solver specializations and simply use an out of the box CG solver for smoothing purposes.

2.3. **Phase 3 - A Full-blown Multigrid Demo.** At this stage we are ready to implement the necessary abstractions to facilitate a multi-level setup, as the interfaces should not require any further tuning. This is then to be demonstrated in a complete multigrid simulation, that highlights its strengths. For this a multigrid preconditioned CG solver is to be used in the demo, especially as part of the python code.

2.3.1. *A scalar valued problem.* For the scalar case the problem of choice is the Poisson equation, picked due to being a linear elliptic and well studied PDE,

$$\begin{cases} -\Delta u = f & \text{in } \Omega \\ u = 0 & \text{on } \partial\Omega \end{cases}.$$

This setting is also the one, where the showcasing of an adaptive refinement would make the most sense, maybe on a mesh with an re-entrant corner to rectify adaptivity.

2.3.2. *A vector valued problem.* The vector valued problem of choice is the one of linear elasticity

$$\begin{cases} -\nabla \cdot \sigma(u) = f & \text{in } \Omega \\ u = 0 & \text{on } \partial\Omega \end{cases}$$

where σ is the Cauchy stress tensor of a linear isotropic material.

2.4. **Phase 4 - Parallelization and Benchmarking.** This final phase considers the aspects of parallelization, that might slightly alter the implementation, but is really only feasible to test once a complete example is at hand. Especially this might include necessary considerations for the handling of ghost nodes.

After that we are ready to assess the performance and quality of implementation with the now finalized demo, regarding aspects of approximation quality, speed up attained by employing a multigrid approach and parallel scaling of the method.

3. Development Experience

My previous experience in software engineering originates from my studies in numerical and applied mathematics. Namely this includes the work on problems regarding the implementation of

- fast solvers (direct, iterative and Krylov subspace methods),
- randomized linear algebra,
- efficient implementation of quad-/octrees,
- the finite difference and finite element method,
- shape optimization using a phase-field formulation, and
- a (flat-top) partition of unity method.

From a technical standpoint this was achieved using the following tools:

- low level languages, namely C and C++ for performance critical parts,
- export of C++ libraries as Python modules using PyBind11,
- high level languages (Python, Matlab) for rapid prototyping and evaluation,
- MPI for process parallelism, and
- third-party libraries, most prominently for the context of this proposal PETSc, SLEPc, dolfin-adjoint and PyBind11.

This was also accompanied by the necessary tooling for team based and large scale development, as applied in the DOLFINx project, such as clang-tidy, clang-format, GoogleTest, code review processes and last but not least git for version control.

4. Why this project?

The concept of multigrid methods is not only a result of software engineering ingenuity but also from a mathematical standpoint exhibits very nicely the problem's properties and solver characteristics to achieve an optimal complexity solver for the FEM. This makes this for me, as a student of applied mathematics, a very interesting problem and task.

I bring along knowledge of the theoretical background regarding multigrid methods, the more abstract setting of domain decomposition methods (of which the multigrid methods are a subset) and the algebraic multigrid methods, which extend the idea to a mesh-free setting for general linear systems of equations.

Additionally I worked with the now legacy FEniCS and its extensions in multiple occasions. Mostly to compare different approaches with the FEM one, which was easiest to test and verify using FEniCS. So I am aware of the FEniCS project's objective, with the new DOLFINx, quite well, as I myself have been a user of the project on multiple occasions.

In total it would be my pleasure to implement the multigrid method in the DOLFINx framework.