1 Introduction

Here, we are exploring the use of AMG methods for the *unsteady Navier-Stokes* equations. In particular, we are attempting to answer the following research questions:

1. If an algebraic multigrid method (AMG) is used in the numerical solution to the Navier-Stokes equations, are we able to learn anything (in a machine-learning sense) about the solver? Particularly, which AMG setups are most convergent and which fail to solve the problem for a specific timestep?

2. If we do indeed learn something about the solver, can we use this to optimize and boost the solver efficiency (in terms of computational complexity or running time)? A few examples where machine learning may be helpful:

   (a) Allowing us to reuse AMG setups until we can confidently *guess* that they will no longer be effective in solving the problem, and

   (b) giving us a fast way to turn an AMG setup for an older, out-of-date timestep into a convergent setup for a new timestep.

2 Background

Here is a bit of a background because Luke wanted to see the math. Consider the nondimensional incompressible Navier-Stokes equations,

$$\frac{\partial \boldsymbol{u}}{\partial t} + (\boldsymbol{u} \cdot \nabla)\, \boldsymbol{u} + \nabla p - \frac{1}{\mathrm{Re}} \nabla^2 \boldsymbol{u} = \boldsymbol{f} \tag{1}$$

$$\nabla \cdot \boldsymbol{u} = 0.p \tag{2}$$

To obtain the weak form, first define the function space for the velocity space

$$\boldsymbol{H}^1 := \left\{ \boldsymbol{u} \in \mathcal{H}^1 (\Omega)^d \;\middle|\; \boldsymbol{u} = \boldsymbol{w} \text{ on } \partial\Omega_D \right\}, \tag{3}$$

i.e. the set of functions that are square integrable and have value $\boldsymbol{w}$ on the Dirichlet boundary. Also, define the related function space that vanishes on the boundary,

$$\boldsymbol{H}^1_0 := \left\{ \boldsymbol{u} \in \mathcal{H}^1 (\Omega)^d \;\middle|\; \boldsymbol{u} = \boldsymbol{0} \text{ on } \partial\Omega_D \right\}. \tag{4}$$

Let $\boldsymbol{v} \in \boldsymbol{H}^1_0$ and $q \in L^2$. To obtain the weak form, multiply equation (1) by $\boldsymbol{v}$ and (2) by q and integrate to get

$$\int_\Omega \boldsymbol{v} \cdot \frac{\partial \boldsymbol{u}}{\partial t} + \int_\Omega \boldsymbol{v} \cdot (\boldsymbol{u} \cdot \nabla)\, \boldsymbol{u} + \frac{1}{\mathrm{Re}} \int_\Omega \nabla \boldsymbol{v} \cdot \nabla \boldsymbol{u} - \int_\Omega p\, (\nabla \cdot \boldsymbol{v}) = \int_\Omega \boldsymbol{v} \cdot \boldsymbol{f} \quad \forall \boldsymbol{v} \in \boldsymbol{H}^1_0 \tag{5}$$

$$- \int_\Omega q\, (\nabla \cdot \boldsymbol{u}) = 0 \qquad \forall q \in L^2. \tag{6}$$

With introduction of the following inner products

$$a(\boldsymbol{u},\boldsymbol{v}) = \int_\Omega \nabla\boldsymbol{v}\cdot\nabla\boldsymbol{u} \tag{7}$$

$$c(\boldsymbol{z};\boldsymbol{u},\boldsymbol{v}) = \int_\Omega \boldsymbol{v}\cdot(\boldsymbol{z}\cdot\nabla)\,\boldsymbol{u} \tag{8}$$

$$d(p,\boldsymbol{v}) = \int_\Omega p\,(\nabla\cdot\boldsymbol{v}) \tag{9}$$

$$m(\boldsymbol{u},\boldsymbol{v}) = \int_\Omega \boldsymbol{v}\cdot\boldsymbol{u}, \tag{10}$$

equations (5), (6) can be rewritten as

$$m\left(\frac{\partial\boldsymbol{u}}{\partial t},\boldsymbol{v}\right) + c(\boldsymbol{u};\boldsymbol{u},\boldsymbol{v}) + \frac{1}{\mathrm{Re}}a(\boldsymbol{u},\boldsymbol{v}) - d(\boldsymbol{v},p) = m(\boldsymbol{f},\boldsymbol{v}) \qquad \forall\boldsymbol{v}\in\boldsymbol{H}_0^1 \tag{11}$$

$$-d(\boldsymbol{u},q) = 0 \qquad \forall q\in L^2. \tag{12}$$

To discretize in space, we use P_2 finite-element triangles for the velocity and P_1 triangles for the pressure. This, according to the Ladyzhenskaya–Babuška–Brezzi (LBB) conditions, are a stable mixed-function space for the Navier-Stokes equations.

We can thus rewrite (11), (12) in the matrix form

$$\boldsymbol{M}\frac{\partial\boldsymbol{u}}{\partial t} + \boldsymbol{C_u}\boldsymbol{u} + \frac{1}{\mathrm{Re}}\boldsymbol{A}\boldsymbol{u} - \boldsymbol{D}^T\boldsymbol{p} = \boldsymbol{M}\boldsymbol{f} \tag{13}$$

$$-\boldsymbol{D}\boldsymbol{u} = 0. \tag{14}$$

To discretize the time component, we apply a backward finite difference in time (i.e., implicit euler),

$$\frac{\partial\boldsymbol{u}}{\partial t} = \frac{1}{\Delta t}(\boldsymbol{u}-\boldsymbol{u}_0) + \mathcal{O}\big(\Delta t^2\big). \tag{15}$$

Substituting (15) into (13), (14) and dropping the $\mathcal{O}\big(\Delta t^2\big)$ term results in

$$\frac{1}{\Delta t}\boldsymbol{M}\boldsymbol{u} + \boldsymbol{C_u}\boldsymbol{u} + \frac{1}{\mathrm{Re}}\boldsymbol{A}\boldsymbol{u} - \boldsymbol{D}^T\boldsymbol{p} = \frac{1}{\Delta t}\boldsymbol{M}\boldsymbol{u}_0 + \boldsymbol{M}\boldsymbol{f} \tag{16}$$

$$-\boldsymbol{D}\boldsymbol{u} = 0. \tag{17}$$

We can define

$$\boldsymbol{F}(\boldsymbol{u}) = \frac{1}{\Delta t}\boldsymbol{M} + \boldsymbol{C_u} + \frac{1}{\mathrm{Re}}\boldsymbol{A} \tag{18}$$

to result in the final system

$$\boldsymbol{F}(\boldsymbol{u})\,\boldsymbol{u} - \boldsymbol{D}^T\boldsymbol{p} = \frac{1}{\Delta t}\boldsymbol{M}\boldsymbol{u}_0 + \boldsymbol{M}\boldsymbol{f} \tag{19}$$

$$-\boldsymbol{D}\boldsymbol{u} = 0. \tag{20}$$

2.1 Block Linear Form

Since the system described in (19)-(20) is nonlinear, we wrap it in some nonlinear solver (i.e., Picard or Newton iteration) and obtain the following linearization:

$$\begin{bmatrix} \boldsymbol{F} & -\boldsymbol{D}^T \\ -\boldsymbol{D} & \boldsymbol{0} \end{bmatrix}\begin{bmatrix} \boldsymbol{u} \\ p \end{bmatrix} = \begin{bmatrix} \frac{1}{\Delta t}\boldsymbol{M}\boldsymbol{u}_0 + \boldsymbol{M}\boldsymbol{f} \\ \boldsymbol{0} \end{bmatrix} \tag{21}$$

Carrying out one round of block Gaussian elimination results in the block upper-triangular form

$$\begin{bmatrix} \boldsymbol{F} & -\boldsymbol{D}^T \\ \boldsymbol{0} & -\boldsymbol{D}\boldsymbol{F}^{-1}\boldsymbol{D}^T \end{bmatrix} \begin{bmatrix} \boldsymbol{u} \\ p \end{bmatrix} = \begin{bmatrix} \frac{1}{\Delta t}\boldsymbol{M}\boldsymbol{u}_0 + \boldsymbol{M}\boldsymbol{f} \\ \boldsymbol{F}^{-1}\boldsymbol{D}\left(\frac{1}{\Delta t}\boldsymbol{M}\boldsymbol{u}_0 + \boldsymbol{M}\boldsymbol{f}\right) \end{bmatrix}, \tag{22}$$

We can define $\boldsymbol{S} := \boldsymbol{D}\boldsymbol{F}^{-1}\boldsymbol{D}^T$ and $\boldsymbol{b} := \frac{1}{\Delta t}\boldsymbol{M}\boldsymbol{u}_0 + \boldsymbol{M}\boldsymbol{f}$. This allows us to simplify (22) to the system

$$\begin{bmatrix} \boldsymbol{F} & -\boldsymbol{D}^T \\ \boldsymbol{0} & -\boldsymbol{S} \end{bmatrix} \begin{bmatrix} \boldsymbol{u} \\ p \end{bmatrix} = \begin{bmatrix} \boldsymbol{b} \\ \boldsymbol{F}^{-1}\boldsymbol{D}\boldsymbol{b} \end{bmatrix}. \tag{23}$$

However, $\boldsymbol{S}$ is now completely full; for large systems it is infeasible to actually form the full matrix. Hence we now turn to iterative methods that only need the operation of the matrix-vector product (which we will approximate).

2.2 Preconditioning

One widely-used preconditioner to solve the Schur complement system from the previous section is the *Pressure Convection-Diffusion* (PCD) preconditioner. Introduce the concept of a commutator ε between the divergence and convection-diffusion operators,

$$\varepsilon_h = \boldsymbol{M}_p^{-1}\boldsymbol{D}\boldsymbol{M}_v^{-1}\boldsymbol{F}_v - \boldsymbol{M}_p^{-1}\boldsymbol{F}_p\boldsymbol{Q}_p^{-1}\boldsymbol{D}. \tag{24}$$

Note the subscripts p and v to differentiate between matrices in the *pressure* and *velocity* spaces, respectively. In short, (24) is a measure of how well the divergence and convection-diffusion operators commute. In certain conditions when there is no advection, $\varepsilon_h = 0$.

For the purposes of the PCD preconditioner, we will assume the commutator is small (i.e., equal to zero). This results in the inequality

$$\boldsymbol{M}_p^{-1}\boldsymbol{D}\boldsymbol{M}_v^{-1}\boldsymbol{F}_v = \boldsymbol{M}_p^{-1}\boldsymbol{F}_p\boldsymbol{M}_p^{-1}\boldsymbol{D}. \tag{25}$$

Pre-multiplying (25) by $\boldsymbol{M}_p\boldsymbol{F}_p^{-1}\boldsymbol{M}_p$ and post multiplying by $\boldsymbol{F}_v^{-1}\boldsymbol{D}^T$ gives

$$\boldsymbol{D}\boldsymbol{F}_v^{-1}\boldsymbol{D}^T = \boldsymbol{M}_p\boldsymbol{F}_p^{-1}\boldsymbol{D}\boldsymbol{M}_v^{-1}\boldsymbol{D}^T, \tag{26}$$

which if recalled from above, gives an approximation to the Schur complement $\boldsymbol{S}$. Furthermore, the product $\boldsymbol{D}\boldsymbol{M}_v^{-1}\boldsymbol{D}^T$ can be replaced with the pressure Laplacian $\boldsymbol{A}_p$ to obtain

$$\boldsymbol{D}\boldsymbol{F}_v^{-1}\boldsymbol{D}^T = \boldsymbol{M}_p\boldsymbol{F}_p^{-1}\boldsymbol{A}_p, \tag{27}$$

which grants us several nice properties. Each individual matrix is sparse and invertible, giving us a closed form approximate inverse to the Schur complement as

$$\boldsymbol{S}^{-1} \approx \boldsymbol{A}_p^{-1}\boldsymbol{F}_p\boldsymbol{M}_p^{-1}. \tag{28}$$

2.3 Time dependent case

Since the matrix $\boldsymbol{F}$ includes the time dependent reaction term, with careful setup the above preconditioner should also work to solve the unsteady Navier Stokes equations. However, we

may also treat this reaction term explicitly. Consider (28) with the value of $\boldsymbol{F}$ substituted,

$$\boldsymbol{S}^{-1} \approx \boldsymbol{A}_p^{-1}\left(\frac{1}{\Delta t}\boldsymbol{M} + \boldsymbol{C}_{\boldsymbol{u}} + \frac{1}{\mathrm{Re}}\boldsymbol{A}\right)\boldsymbol{M}_p^{-1} \tag{29}$$

$$= \boldsymbol{A}_p^{-1}\left(\boldsymbol{C}_{\boldsymbol{u}} + \frac{1}{\mathrm{Re}}\boldsymbol{A}\right)\boldsymbol{M}_p^{-1} + \frac{1}{\Delta t}\boldsymbol{A}_p^{-1} \tag{30}$$

$$= \boldsymbol{A}_p^{-1}\left(\boldsymbol{C}_{\boldsymbol{u}} + \frac{1}{\mathrm{Re}}\boldsymbol{A}\right)\boldsymbol{M}_p^{-1} + \frac{1}{\Delta t}\boldsymbol{D}\boldsymbol{M}_v^{-1}\boldsymbol{D}^T \tag{31}$$

3 Cylinder Flow Problem

The model problem being solved is flow in a rectangular cavity around a cylindrical obstacle, the mesh used is shown in figure 1. Dirichlet "no-slip" ($\boldsymbol{v} = \boldsymbol{0}$) conditions are prescribed on the top, bottom, and cylindrical wall. On the left inlet, a velocity Dirichlet condition of $\boldsymbol{v} = (2, 0)$ is prescribed. On the right wall, the velocity has a Neumann ($\nabla \boldsymbol{v} \cdot \hat{\boldsymbol{n}} = 0$) to allow the outlet velocity value to "float" to satisfy the other conditions.

For the pressures space, homogeneous Dirichlet conditions are defined at the top, bottom, and cylindrical walls. The left and right walls have Neumann boundary conditions. Note that these boundary conditions do not need to be explicitly defined, rather they are imposed by the weak formulation.

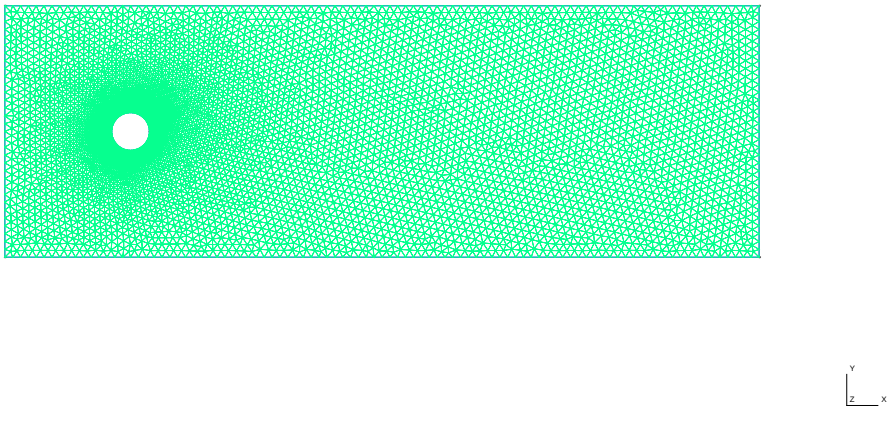

Figure 1: Cylinder flow domain.

The rectangular domain spans the space $[0, 6] \times [0, 2] \in \mathbb{R}^2$. The cylindrical hole is centered at the point $(1, 1)$ and has radius of 0.15. This problem has an effective Reynolds number of 62.5, which is conducive to creating vortex shedding (Fig 2).

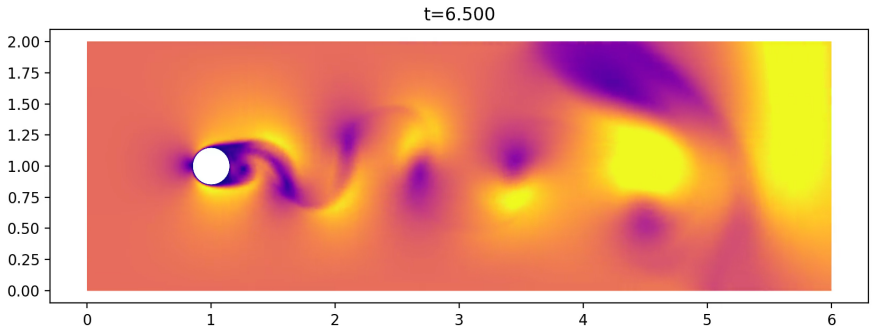

Figure 2: Solution to the problem at $t = 6.5$. Note the vortex shedding phenomena that is visible to the right of the cylinder.

4 Firedrake Solver

The Firedrake[6] solution to the above problem uses a "Flexible" GMRES KSP solver to solve the overall linearized system – *flexible* in that the preconditioner is allowed to change between iterations. This is preconditioned by a PETSc fieldsplit PC, allowing the velocity and pressure blocks to be separately solved.

The velocity block is solved using an AMG-preconditioned GMRES solver. The AMG routine is a Ruge-Stüben solver courtesy of PyAMG[5].

The pressure Schur complement is solved using a time-dependent PCD preconditioner as outlined in (31). This is a slight modification of the PCD preconditioner that is built-in to Firedrake itself. Each of the sub-matrices in the PCD preconditioner (pressure laplacian, pressure mass, and reaction compnent) are solved using a direct LU solver.

5 Learning Interpolation

Below are a few ideas on how to effectively learn an AMG interpolation operator for the above Navier-Stokes problem. These can be considered an extension of the techniques presented by Luz, Galun, Maron, Basri, and Yavneh [4] and follow the classical AMG setup of first creating the C/F partitioning and the interpolation operator itself. The basic method is to have two separate graph networks working serially:

1. A first graph network that, for a given A matrix, will output an optimal C/F partitioning for the problem. The output should be a vector $\boldsymbol{c}$ whose entries lie in the range $[0, 1]$ where 0 corresponds to fine points and 1 to coarse points. Denote this as the *P-network*.

2. Another graph network that takes a C/F partitioning and outputs the columns of the interpolation operator $\boldsymbol{P}$. If necessary, the graph edge data could be used to impose a sparsity pattern on the resultant columns. Denote this as the *CF-network*.

Having the two "steps" as separate networks can bring a few benefits. One is that each network could be trained separately and then combined later to create a full AMG method – each having a different loss function, training set, etc.

5.1 Loss Function

To train the networks, we will use a loss function based on the spectral radius of the error propagator:

$$\boldsymbol{E} := \boldsymbol{R}^{\nu}\boldsymbol{C}\boldsymbol{R}^{\nu} \tag{32}$$

where $\boldsymbol{R}$ is the error propagation per iteration of the relaxation scheme, ν is the number of pre-and-post-relaxation steps, and $\boldsymbol{G}$ is the coarse-grid correction error propagator,

$$\boldsymbol{G} := \boldsymbol{I} - \boldsymbol{P}\left(\boldsymbol{P}^T\boldsymbol{A}\boldsymbol{P}\right)^{-1}\boldsymbol{P}^T\boldsymbol{A}. \tag{33}$$

In this case, weighted Jacobi iteration is used for the relaxation scheme with a weight of $\omega = \frac{2}{3}$, giving an error propagator of

$$\boldsymbol{R} := \boldsymbol{I} - \frac{2}{3}\boldsymbol{D}^{-1}\boldsymbol{A}, \tag{34}$$

where $\boldsymbol{D} = \operatorname{diag}(\boldsymbol{A})$. In theory, an optimal interpolation operator should minimize the spectral radius of the error propagator, $\rho(\boldsymbol{E})$. However, backpropagation of the maximal eigenvector computed through, power iteration for example, tends to be rather unstable and can lead to numerical overflow[7]. Therefore, as proposed in [4], minimizing the squared Frobenius norm, $\|\boldsymbol{A}\|_F^2 = \sum \lambda_i^2(\boldsymbol{A})$ for SPD $\boldsymbol{A}$ can be done as a proxy.

Using purely the error propagator as the loss will simply minimize the number coarse points in the final interpolation operator, which is unwanted. Therefore, we will want to add a penalty proportional to the number of coarse points. If we allow $\boldsymbol{c}$ to be the vector containing the C/F splitting and whose entries float between 0 (fine) and 1 (coarse), we can use the L_1 norm $\|\boldsymbol{c}\|_1$ to penalize how "dense" the vector is.

Bringing this all together, we obtain the final loss function

$$\ell := \|\boldsymbol{E}\|_F^2 + \alpha\|\boldsymbol{c}\|_1^2 \tag{35}$$

for some scaling coefficient α.

There still remains the question of how do we represent the interchange of data between the "CF" network and the "P" network? If we were to discretely split the C/F space at 0.5, and for example take $c_i \geq 0.5$ to be coarse points and $c_i < 0.5$ to be fine points, we would run into the issue of having a discontinuous loss function in that gradient-descent based methods could not effectively train our pair of networks. If our points are *either* coarse *or* fine, will we have any gradient information that would allow us to switch assignments?

After some discussion with Matt, a few possible strategies are outlined below.

5.1.1 Continuous Loss

In this stategy, instead of generating $\boldsymbol{P} \in \mathbb{R}^{N_F \times N_C}$, we instead generate the "full" operator $\hat{\boldsymbol{P}} \in \mathbb{R}^{N_F \times N_F}$. I.e., we compute all columns of the interpolation operator as if each node

was coarse. Let $\boldsymbol{C} := \operatorname{diag}(\boldsymbol{c})$, then we can update the coarse-grid propagator by somehow replacing $\boldsymbol{P}$ with $\hat{\boldsymbol{P}}$ and $\boldsymbol{C}$:

$$\boldsymbol{G} := \boldsymbol{I} - \boldsymbol{P}\left(\boldsymbol{P}^T\boldsymbol{A}\boldsymbol{P}\right)^{-1}\boldsymbol{P}^T\boldsymbol{A}. \tag{36}$$

I don't know. Matt and I got kind of stuck here. Matt suggested

$$\boldsymbol{P} = \boldsymbol{C}\hat{\boldsymbol{P}}\boldsymbol{C} + (\boldsymbol{I} - \boldsymbol{C}), \tag{37}$$

effectively inserting zeros into the columns of $\hat{\boldsymbol{P}}$ then replacing them with 1. The idea here is to find *"some way to stick* $\operatorname{diag}(\boldsymbol{c})$ *into the calculation of* $\boldsymbol{E}$ *in a way that, if* $\boldsymbol{c}$ *is just* 0*s and* 1*s we get exactly the right* $\boldsymbol{E}$*".*

5.1.2 Reinforcement Learning, "Generalized Ali Method"

Here, we reframe the problem into a RL context. Training is initialized with a partitioning of only fine points, then after each training step some node is flipped from fine to coarse according to the following algorithm:

1. For timestep t, let $\hat{\boldsymbol{P}}_t$ and $\boldsymbol{c}_t$ be the interpolation and c/f splitting, respectively.
2. Have an RL network that takes $\hat{\boldsymbol{P}}_t$, $\boldsymbol{c}_t$, outputs $\widetilde{\boldsymbol{P}}_t$, $\widetilde{\boldsymbol{c}}_t$.
3. Take the largest entry in $\widetilde{\boldsymbol{c}}_t - \boldsymbol{c}_t$ as an indicator of needing to flip entry i from fine to coarse. Generate $\hat{\boldsymbol{P}}_{t+1}$ and $\boldsymbol{c}_{t+1}$ from this new information.
4. Repeat until some specified stopping condition.

At each timestep, we can directly use the discontinuous loss from (35).

5.1.3 Reinforcement Learning, "Generalized AlphaGo Method"

Use MCTS like AlphaGo. This would have the benefit of having another network that could out some measure of "goodness" for a given interpolation operator. This could be helpful in answering the research question in the introduction: can we reuse AMG setups for multiple timesteps until we can confidently *guess* that they will no longer be effective?

I don't have a lot to say about this one. I would need to do some reading about this.

5.1.4 Hybrid RL and Unsupervised Method, "Ali + Nicolas Method"

Here, we decouple the *P-net* and *CF-net* networks and learn the former by an RL method and the latter by some unsupervised loss. The training algorithm would follow this outline:

1. First, train the *P-net* on various randomly-generated CF partitions. Here, we would only use the squared Frobenius norm, $\|\boldsymbol{E}\|_F^2$, as the loss.
2. Now, train the *CF-net* as in the *Generalized Ali Method* using the loss function $\ell := \|\boldsymbol{E}(P(\boldsymbol{c}))\|_F^2 + \alpha\|\boldsymbol{c}\|_1$, where the $\boldsymbol{E}(P(\boldsymbol{c}))$ refers to the error propagator of the interpolation operator generated from coarsening $\boldsymbol{c}$ by the *P-net*.
3. After each episode of RL training, train the *P-net* on the coarsenings generated by the *CF-net* so that the *P-net* can learn to effectively generate interpolations for the coarsenings we find.
4. Repeat steps (2) and (3) until some specified stopping condition.

References

[1] J. Blechta and M. Řehoř, *Mathematical background to the Navier-Stokes PCD Preconditioner*. `https://fenapack.readthedocs.io/en/2019.1.0/math.html#math-background`, 2019. Accessed: 2021-6-17.

[2] N. Bootland, A. Bentley, C. Kees, and A. Wathen, *Preconditioners for two-phase incompressible navier–stokes flow*, SIAM Journal on Scientific Computing, 41 (2019), pp. B843–B869.

[3] H. C. Elman, D. J. Silvester, and A. J. Wathen, *Finite elements and fast iterative solvers: with applications in incompressible fluid dynamics*, Numerical mathematics and scientific computation, Oxford University Press, Oxford, United Kingdom, second edition ed., 2014.

[4] I. Luz, M. Galun, H. Maron, R. Basri, and I. Yavneh, *Learning algebraic multigrid using graph neural networks*, 2020.

[5] L. N. Olson and J. B. Schroder, *Pyamg: Algebraic multigrid solvers in python v4.0*. `https://github.com/pyamg/pyamg`, 2018. Release 4.0.

[6] F. Rathgeber, D. A. Ham, L. Mitchell, M. Lange, F. Luporini, A. T. T. McRae, G. Bercea, G. R. Markall, and P. H. J. Kelly, *Firedrake: automating the finite element method by composing abstractions*, CoRR, abs/1501.01809 (2015).

[7] W. Wang, Z. Dang, Y. Hu, P. Fua, and M. Salzmann, *Backpropagation-friendly eigendecomposition*, 2019.