




# PyAMG: Algebraic Multigrid Solvers in Python

Nathan Bell<sup>1</sup>, Luke N. Olson <sup>2</sup>, Jacob Schroder <sup>3</sup>, and Ben Southworth <sup>4</sup>

<sup>1</sup> Google, Mountain View, CA, USA <sup>2</sup> Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana, IL USA 61801 <sup>3</sup> Department of Mathematics and Statistics, University of New Mexico, Albuquerque, NM USA 87131 <sup>4</sup> Los Alamos National Laboratory, Los Alamos, NM USA 87545

DOI: [10.21105/joss.05495](https://doi.org/10.21105/joss.05495)

## Software

- [Review](#) 
- [Repository](#) 
- [Archive](#) 

Editor: [Jed Brown](#)  

## Reviewers:

- [@mayrmt](#)
- [@Jeff-Hadley](#)
- [@victorapm](#)

Submitted: 05 May 2023

Published: 04 July 2023

## License

Authors of papers retain copyright and release the work under a Creative Commons Attribution 4.0 International License ([CC BY 4.0](https://creativecommons.org/licenses/by/4.0/)).

## Statement of need

PyAMG is a Python package of algebraic multigrid (AMG) solvers and supporting tools for approximating the solution to large, sparse linear systems of algebraic equations,

$$Ax = b,$$

where  $A$  is an  $n \times n$  sparse matrix. Sparse linear systems arise in a range of problems in science, from fluid flows to solid mechanics to data analysis. While the direct solvers available in SciPy's sparse linear algebra package (`scipy.sparse.linalg`) are highly efficient, in many cases *iterative* methods are preferred due to overall complexity. However, the iterative methods in SciPy, such as CG and GMRES, often require an efficient preconditioner in order to achieve a lower complexity. Preconditioning is a powerful tool whereby the conditioning of the linear system and convergence rate of the iterative method are both dramatically improved. PyAMG constructs multigrid solvers for use as a preconditioner in this setting. A summary of multigrid and algebraic multigrid solvers can be found in Olson (2015a), in Olson (2015b), and in Falgout (2006); a detailed description can be found in Briggs et al. (2000) and Trottenberg et al. (2001). PyAMG provides a comprehensive suite of AMG solvers (see Methods), which is beneficial because many AMG solvers are specialized for particular problem types.

## Summary

The overarching goals of PyAMG include both readability and performance. This includes readable implementations of many popular variations of AMG (see the Methods section), the ability to reproduce results in the literature, and a user-friendly interface to AMG allowing straightforward access to the variety of AMG parameters in the method(s). Additionally, pure Python implementations are not efficient for many sparse matrix operations not already available in `scipy.sparse` — e.g., the sparse matrix graph coarsening algorithms needed by AMG. For such cases in PyAMG, the compute (or memory) intensive kernels are typically expressed in C++ and wrapped through PyBind11, while the method interface and error handling is implemented directly in Python (more in the next section).

In the end, the goal of PyAMG is to provide quick access, rapid prototyping of new AMG solvers, including easy comparison with many existing variations of AMG in the literature, and performant execution of AMG methods. The extensive PyAMG [Examples](#) page highlights many of the package's advanced AMG capabilities, e.g., for Hermitian, complex, nonsymmetric, and other challenging system types. It is important to note that many other AMG packages exist, mainly with a focus on parallelism and performance, rather than quick access and rapid prototyping. This includes BoomerAMG in hypre (Henson & Yang, 2002; hypre, 2022), MueLu in Trilinos (MueLu Project Team, 2020; Trilinos Project Team, 2020), and GAMG within PETSc (Balay et al., 2021), along with other packages focused on accelerators (Bell et al.,

2012), such as AmgX (Naumov et al., 2015), CUSP (Dalton et al., 2014), and AMGCL (Demidov, 2019).

## Design

The central data model in PyAMG is that of a `MultiLevel` object, which is constructed in the `setup` phase of AMG. The multigrid hierarchy is expressed in this object (details below) along with information for the `solve` phase, which can be executed on various input data,  $b$ , to solve  $Ax = b$ .

The `MultiLevel` object consists of a list of multigrid `Level` objects and diagnostic information. For example, a `MultiLevel` object named `ml` contains the list `ml.levels`. Then, the data on level  $i$  (with the finest level denoted  $i=0$ ) accessible in `ml.levels[i]` includes the following information:

- `A`: the sparse matrix operator, in CSR or BSR format, on level  $i$ ;
- `P`: a sparse matrix interpolation operator to transfer grid vectors from level  $i+1$  to  $i$ ;
- `R`: a sparse matrix restriction operator to transfer grid vectors from level  $i$  to  $i+1$ ; and
- `presmoothen`, `postsmoothen`: functions that implement pre/post-relaxation in the solve phase, such as weighted Jacobi or Gauss-Seidel.

Other data may be retained for additional diagnostics, such as grid splitting information, aggregation information, etc., and would be included in each level.

Specific multigrid methods (next section) in PyAMG and their parameters are generally described and constructed in Python, while key performance components of both the setup and solve phase are written in C++. Heavy looping that cannot be accomplished with vectorized or efficient calls to NumPy or sparse matrix operations that are not readily expressed as SciPy sparse (CSR or CSC) operations are contained in short, templated C++ functions. The templates are used to avoid type recasting the variety of input arrays. The direct wrapping to Python is handled through another layer with PyBind11. Roughly 26% of PyAMG is in C++, with the rest in Python.

## Methods

PyAMG implements among the most wide-ranging suites of base AMG methods, each with a range of options. The base forms for a solver include

- `ruge_stuben_solver()`: the classical form of C/F-type AMG (Ruge & Stüben, 1987);
- `smoothed_aggregation_solver()`: smoothed aggregation based AMG as introduced in (Vaněk et al., 1996);
- `pairwise_solver()`: pairwise (unsmoothed) aggregation based AMG as introduced in (Notay, 2010);
- `adaptive_sa_solver()`: a so-called adaptive form of smoothed aggregation from (Brezina et al., 2005); and
- `rootnode_solver()`: the root-node AMG method from (Thomas A. Manteuffel et al., 2017), applicable also to some nonsymmetric systems.
- `air_solver()`: the nonsymmetric AMG method based on approximate ideal restriction (AIR) from (Thomas A. Manteuffel et al., 2018, 2019), which is highly effective for many upwind discretizations of advection-dominated problems.

In each of these, the *base* algorithm is available but defaults may be modified for robustness. Options such as the default pre/postsmoother or smoothing the input candidate vectors (in the case of smoothed aggregation or root-node AMG), can be modified to tune the solver. In addition, several cycles are available, including the standard V, F, and W cycles, for the solve phase. The resulting method can also be used as a preconditioner within the Krylov methods

available in PyAMG or with SciPy's Krylov methods. The methods in PyAMG generally support complex data types and nonsymmetric matrices. All `MultiLevel` objects provide a detailed measure of the grid complexity (number of unknowns on all levels / number of unknowns on the finest level), operator complexity (number of nonzeros in the matrix on all levels / number of nonzeros in the matrix on the finest level), and cycle complexity (approximate cost in floating point operations (FLOPs) of a single multigrid cycle relative to a single matrix-vector multiply).

## Example

As an example, consider a five-point finite difference approximation to a Poisson problem,  $-\Delta u = f$ , given in matrix form as  $Ax = b$ . The AMG setup phase is called with

---

```
1 import pyamg
2 A = pyamg.gallery.poisson((1000,1000), format='csr')
3 ml = pyamg.smoothed_aggregation_solver(A, max_coarse=10)
```

---

For this case, with 1M unknowns, the following multilevel hierarchy is generated for smoothed aggregation (using `print(ml)`):

```
MultilevelSolver
Number of Levels:      7
Operator Complexity:  1.338
Grid Complexity:      1.188
Coarse Solver:        'pinv'
  level  unknowns  nonzeros
    0     1000000   4996000 [74.75%]
    1     167000   1499328 [22.43%]
    2      18579   167051 [2.50%]
    3       2086   18870 [0.28%]
    4        233    2109 [0.03%]
    5         28     248 [0.00%]
    6          3         9 [0.00%]
```

In this case, the hierarchy consists of seven levels, with SciPy's pseudoinverse (`pinv`) being used on the coarsest level. Also displayed is the ratio of unknowns (nonzeros) on all levels compared to the fine level, also known as the grid (operator) complexity.

The solve phase, using standard V-cycles, is executed with the object's `solve`:

---

```
1 import numpy as np
2 x0 = np.random.rand(A.shape[0])
3 b = np.zeros(A.shape[0])
4 res = []
5 x = ml.solve(b, x0, tol=1e-10, residuals=res)
```

---

This leads to the residual history shown in [Figure 1](#). Additional examples can be found at [github.com/pyamg/pyamg-examples](https://github.com/pyamg/pyamg-examples), including examples with classical AMG using AIR, building solvers with rootnode, and nonsymmetric use cases.

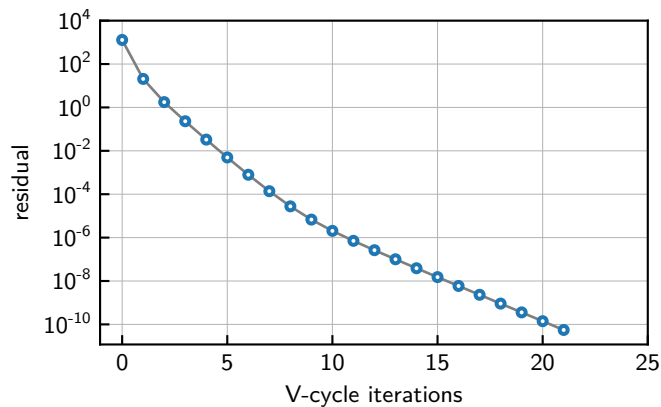


Figure 1: Algebraic multigrid convergence (relative residual).

## References

- Balay, S., Abhyankar, S., Adams, M. F., Benson, S., Brown, J., Brune, P., Buschelman, K., Constantinescu, E. M., Dalcin, L., Dener, A., Eijkhout, V., Gropp, W. D., Hapla, V., Isaac, T., Jolivet, P., Karpeev, D., Kaushik, D., Knepley, M. G., Kong, F., ... Zhang, J. (2021). *PETSc Web page*. <https://petsc.org/>
- Bell, N., Dalton, S., & Olson, L. N. (2012). Exposing fine-grained parallelism in algebraic multigrid methods. *SIAM Journal on Scientific Computing*, 34(4), C123–C152. <https://doi.org/10.1137/110838844>
- Brezina, M., Falgout, R., MacLachlan, S., Manteuffel, T., McCormick, S., & Ruge, J. (2005). Adaptive smoothed aggregation ( $\alpha$  SA) multigrid. *SIAM Review*, 47(2), 317–346. <https://doi.org/10.1137/050626272>
- Briggs, W. L., Henson, V. E., & McCormick, S. F. (2000). *A multigrid tutorial* (Second). Society for Industrial and Applied Mathematics. <https://doi.org/10.1137/1.9780898719505>
- Dalton, S., Bell, N., Olson, L., & Garland, M. (2014). *Cusp: Generic parallel algorithms for sparse matrix and graph computations*. <http://cusplibrary.github.io/>
- Demidov, D. (2019). AMGCL: An efficient, flexible, and extensible algebraic multigrid implementation. *Lobachevskii Journal of Mathematics*, 40(5), 535–546. <https://doi.org/10.1134/s1995080219050056>
- Falgout, R. D. (2006). An introduction to algebraic multigrid. *Computing in Science & Engineering*, 8(6), 24–33. <https://doi.org/10.1109/MCSE.2006.105>
- Henson, V. E., & Yang, U. M. (2002). BoomerAMG: A parallel algebraic multigrid solver and preconditioner. *Applied Numerical Mathematics*, 41(1), 155–177. [https://doi.org/10.1016/S0168-9274\(01\)00115-5](https://doi.org/10.1016/S0168-9274(01)00115-5)
- hypre. (2022). *High performance preconditioners*. <https://github.com/hypre-space/hypre>
- Manteuffel, Thomas A., Münzenmaier, S., Ruge, J., & Southworth, B. (2019). Nonsymmetric reduction-based algebraic multigrid. *SIAM Journal on Scientific Computing*, 41(5), S242–S268. <https://doi.org/10.1137/18M1193761>
- Manteuffel, Thomas A., Olson, L. N., Schroder, J. B., & Southworth, B. S. (2017). A root-node-based algebraic multigrid method. *SIAM Journal on Scientific Computing*, 39(5), S723–S756. <https://doi.org/10.1137/16M1082706>
- Manteuffel, Thomas A., Ruge, J., & Southworth, B. S. (2018). Nonsymmetric algebraic multigrid based on local approximate ideal restriction ( $\ell$  AIR). *SIAM Journal on Scientific*

- Computing*, 40(6), A4105–A4130. <https://doi.org/10.1137/17M1144350>
- MueLu Project Team. (2020). *MueLu Project website*. <https://trilinos.github.io/muelu.html>
- Naumov, M., Arsaev, M., Castonguay, P., Cohen, J., Demouth, J., Eaton, J., Layton, S., Markovskiy, N., Reguly, I., Sakharnykh, N., Sellappan, V., & Strzodka, R. (2015). AmgX: A library for GPU accelerated algebraic multigrid and preconditioned iterative methods. *SIAM Journal on Scientific Computing*, 37(5), S602–S626. <https://doi.org/10.1137/140980260>
- Notay, Y. (2010). An aggregation-based algebraic multigrid method. *Electronic Transactions on Numerical Analysis*, 37(6), 123–146. <https://etna.math.kent.edu/vol.37.2010/pp123-146.dir/pp123-146.html>
- Olson, L. (2015a). Multigrid methods: algebraic. In B. Engquist (Ed.), *Encyclopedia of applied and computational mathematics* (pp. 977–981). Springer Berlin Heidelberg. [https://doi.org/10.1007/978-3-540-70529-1\\_337](https://doi.org/10.1007/978-3-540-70529-1_337)
- Olson, L. (2015b). Multigrid methods: geometric. In B. Engquist (Ed.), *Encyclopedia of applied and computational mathematics* (pp. 981–987). Springer Berlin Heidelberg. [https://doi.org/10.1007/978-3-540-70529-1\\_338](https://doi.org/10.1007/978-3-540-70529-1_338)
- Ruge, J. W., & Stüben, K. (1987). Algebraic multigrid (AMG). In S. F. McCormick (Ed.), *Multigrid methods* (Vol. 3, pp. 73–130). SIAM. <https://doi.org/10.1137/1.9781611971057>
- Trilinos Project Team. (2020). *Trilinos Project website*. <https://trilinos.github.io>
- Trottenberg, U., Oosterlee, C. W., & Schüller, A. (2001). *Multigrid* (p. xvi+631). Academic Press, Inc., San Diego, CA. ISBN: 0-12-701070-X
- Vaněk, P., Mandel, J., & Brezina, M. (1996). Algebraic multigrid by smoothed aggregation for second and fourth order elliptic problems. *Computing*, 56(3), 179–196. <https://doi.org/10.1007/BF02238511>