AMGLAB: AN INTERACTIVE MATLAB TESTBENCH FOR LEARNING AND EXPERIMENTATION WITH ALGEBRAIC MULTIGRID METHODS[*]

RYAN MCKENZIE[†], CRAIG C. DOUGLAS[‡], AND GUNDOLF HAASE[§]

Abstract. We describe a first step towards a general algebraic multigrid expert system that we expect to become a community project in the multigrid field.

1. Introduction to the AMGLab Project.

The AMGLab project is the pilot work for an algebraic multigrid (AMG) expert system. This section will introduce the motivation for such an expert system as well as outline in some detail what this expert system must consist of.

Currently, in industry and academia, algebraic multigrid algorithms are custom tailored and implemented on an application by application basis. For instance, what if a research group in physics or environmental sciences wishes to attempt an algebraic multigrid approach on their wildfire or ocean modeling problem? There is very little or no prototyping available in the community to help them in their implementation. They must derive a particular algebraic multigrid algorithm suited for their application with little or no guidance. Issues such as which class of setup algorithm to use, what kind of multigrid cycle to use, which smoother should be used, and what kind of coarsening heuristics to use must be answered by the group itself on a near guess and check basis. This can greatly increase the development time for an AMG application. In some cases, this process has also been known to force the abandonment of an algebraic multigrid approach altogether.

The motivation for an expert system is three pronged. First, help groups in industry and research determine if an algebraic multigrid algorithm is appropriate for their application. Second, if so, help that group select an algebraic multigrid algorithm having the best properties for their application. Finally, provide some example implementation of such an algorithm in a rapid prototyping language. The group can then choose to use the example implementation as-is or use that example to implement their version in a much more efficient environment or use a production version on a serial or parallel computer.

The AMGLab project provides some of the basic framework for such an expert system. We have three goals presently.

- We implemented AMGLab in a modular, extensible manner using a rapid prototyping language. This allows members of the algebraic multigrid community to quickly add functionality to the test bench. The more algorithm options that can be represented in the testbench, the better our eventual expert system will be.
- Implement at least one setup algorithm from representative classes of algebraic multigrid setup and allow options for finite element, difference, or volume discretizations. We also wanted to allow users to import their own problem definitions directly into the testbench in addition to "canned" prob-

[*]This project is supported in part by NSF grants EIA-0219627, OISE-0405349, ACI-0324876, ACI-0324876, and CNS-0540178.

[†]University of Kentucky, Computer Science, ryan.n.mckenzie@gmail.com

[‡]University of Kentucky and Yale University, Computer Science, douglas-craig@cs.yale.edu

[§]University of Graz, Mathematics, gundolf.haase@uni-graz.at

lems such as classical Poisson. These features make AMGLab immediately useful to the multigrid community and to potential users of algebraic multigrid algorithms. Without having to wait for extensions and features to be added, the project already has some limited use as a rudimentary expert system.
- We provide users full access to as many factors as possible which affect the performance of the algorithms. We do this to help users explore and understand the effect each change of the algorithm properties has on the solution of their particular problem. This helps them make better decisions when they are ready to implement their efficient solution algorithms.

2. How and Why AMGLab Meets our Goals. Let us demonstrate that AMGLab is a software suite designed to meet the goals of this first step toward an expert system. Perhaps the simplest fact to demonstrate is that AMGLab meets our goals for being implemented in a rapid prototyping language. We chose Matlab as our environment for implementation for the following reasons:
- Matlab is the de facto problem solving environment used in prototyping by the scientific computing community. Thus, our potential users and contributors are likely to be already familiar with Matlab.
- Matlab has some powerful features that make it perfect for implementing an algebraic multigrid solution system.
- Many common linear algebra subroutines have been built into Matlab such as direct elimination and common iterative solvers for sparse linear systems.
- Matlab offers some convenient graphical features well suited for numerical applications.
- There is a profiler that measures how much time each subroutine in the system executes and displays this information in the form of a bar chart.
- There is a powerful plotting tool on which to display things like computed solutions and residual reduction rates.

These features of Matlab will allow users of AMGLab to quickly add options and algorithms as well as gage their performance quite easily.

One of AMGLab's prime strengths is that it has been designed to be easily extensible. It is the goal of our expert system to allow users to add algorithms to the arsenal already in the tool, making it more complete and helpful. The current implementation of AMGLab has many features that make it extensible. The storage infrastructure for the system is contained in a single file. Therefore, if the current infrastructure does not offer storage for some information a new method requires (for instance spectral methods require storage for eigenvalues and vectors) it can be added here. All major components of the solution cycle have been written as switches rather than explicitly calling a single subroutine. In order to add a new setup subroutine, a user would only have to change the AMGLab infrastructure in two places. They would need to add an option name for their method in the globals file and then add a branch for their option to the switch logic in the setup subroutine. A graphical user interface has been provided for the base implementation for convenience. In order to incorporate any extensions they have made to the functionality of the system accessible in the GUI, the contributor must make several other changes to the GUI system. Matlab makes this easy by providing GUIde. GUIde is Matlab's built in GUI creator and editor. For flexibility, all system code can also be invoked from the Matlab command line.

AMGLab has implemented the base multigrid cycle solution system as well as some of the canonical algebraic multigrid algorithms. AMGLab is also capable of reading in user-specified problems in finite difference and finite element form. There is also a simple built-in Poisson solution to verify the correctness of combinations of algorithms and options. These features combine to ensure that AMGLab is a strong contribution to the multigrid community from the start. AMGLab in its current form offers these capabilities as a rudimentary expert system. Users from industry and academia can immediately begin choosing appropriate AMG frameworks for their applications immediately. Also, contributors and colleagues can begin expanding the expert system by adding options and algorithms to the suite.

3. Core Algorithms. Now let us explore some of the key algorithms which are included in AMGLab. First, let the reader be reminded that the algebraic multigrid system takes place in two phases on the global level. Those phases are Setup and Solution respectively. Let us insert a brief outline of those global phases.

AMGLab includes V-, W-, and F-Cycles [3], [11]. Additional cycles can be defined by the user.

The three step process of transferring to successively coarser grids, a coarsest grid solve, and then a reversal of the grid transfer is particularly well suited for recursion. This is due to the fact that it is normally unknown how many levels of coarsening to execute a priori. The base case of the algorithm is satisfied when the current level is "sufficiently coarse." This is a very broad term and is left up to the implementer to decide. For instance, one could check for the number of data points or simply limit the number of levels. In the case of AMGLab, at most five levels of coarsening can occur. If the base case is not satisfied, meaning we are not at the coarsest level, the "downward" section of the algorithm is first executed. This section uses the restriction operator derived during setup to transfer our approximation to the next coarsest grid. If setup is not done at-once or up-front, a coarsening strategy must be used at each level to determine the restriction operator. Next is the recursive call. At this point, the V-Cycle algorithm is executed on this next coarsest grid. Once the recursive call returns, the "upward" side of the algorithm takes the coarse level approximations and interpolates them to the current level. These interpolated coarse results are used to correct the current approximation, and then the corrected results are returned to be used at the next finest grid level. The reader should also note that other "shapes" exist for the multigrid cycle.

Now let us consider the algorithms used to perform the work necessary to transfer grid levels in a multigrid scheme. At either the global setup or the restriction step of the multigrid process, work must be transferred from the current grid to the next coarser grid. This activity is known as coarsening. In geometric multigrid, coarsening may be accomplished by a weighted average of the coefficients on the fine grid. In algebraic multigrid, we must use a coarsening strategy involving heuristics to choose the coarse grid nodes. Then, a grid transfer weight (often called an interpolation weight) must be generated in order to mathematically resolve the numbering of the nodes (or data points) from the fine to the coarse grid. The algorithms which perform coarse grid selection and weight generation are called coarsening strategies. AMGLab implements three such algorithms.

AMGLab supports the classical coarsening strategy proposed by Ruge and Stueben in 1987 [9]. The Ruge-Stueben algorithm relies on a system coefficient matrix (stiffness matrix) in order to complete coarsening.

The original Ruge-Stueben coarsening strategy as proposed in [9] differs from the algorithm in AMGLab slightly. In the proposed algorithm, there are two phases of coarse point selection, initial selection and a secondary selection where the initial classifications are reviewed according to more stringent heuristics. Most implementations of the algorithm, especially those intended for rapid prototyping, skip this secondary coarsening phase due to the complexity which that phase adds to the overall algorithm. In many cases, implementing the second phase of coarsening does not sufficiently improve the coarsening to justify the added time needed to execute it. Therefore, we have skipped the second phase in the AMGLab implementation. The AMGLab implementation of Ruge-Stueben coarsening begins with an initialization phase. A set consisting of all nodes on the fine grid is created for future reference. The set of selected coarse nodes is initialized to the empty set, unless the user decides to pre-select some coarse points. If the user pre-selected coarse nodes, some fine nodes are also set aside accordingly. Finally, the working set is created containing all nodes that have not been set aside as coarse or fine yet. In the normal case, this set is equivalent to the set of all nodes on the fine grid.

To begin coarse point selection, the algorithm evaluates each point in the working set according to two heuristics. First, we count the number of strong connections that point has to other points in the grid. Second, we count the number of weak connections the considered point has to already selected fine points. Once each point is evaluated, the one having the maximum score on those two heuristics is added to the coarse set and any of its weak neighbors that are in the working set are added to the fine set. The working set is then updated to reflect any nodes which have been assigned to coarse or fine status. The whole process is repeated until there are no points remaining to process in the working set.

Many coarsening strategies take the same general approach. They create initial groupings of coarse, fine, and working set nodes. Then, the algorithms will evaluate each node in the working set according to some heuristics. Then, according to this evaluation, nodes are removed from the working set and placed in either the fine or coarse set. Some coarsening strategies will only tentatively assign a node to the coarse or fine set. This allows such algorithms to redistribute the coarse point selection at a later time if a previous decision is determined to be inappropriate. Such algorithms usually have substantially longer execution times as a result of the repeated shuffling of points from one selection to another.

The second coarsening strategy implemented in AMGLab is the coarsening by smoothed aggregation [2], [12]. The particular algorithm implemented is based on work by Vanek, Mandel, and Brezina [11]. Like the Ruge-Stueben strategy, smoothed aggregation requires the system stiffness matrix.

A major difference between the smoothed aggregation approach and many other coarsening methods is that interpolation is coupled with a smoothing step in order to reduce errors associated with the interpolation operator. Also, the selection of coarse points is not done by iteratively evaluating each node and classifying it according to its heuristic score. Instead, the coarsening begins by grouping the fine grid into several disjoint sets of strongly coupled neighbors. This grouping into neighborhoods is called aggregation. If there are any fine grid points remaining which do not belong to a neighborhood, we attempt to add them to one of the neighborhoods. If the orphaned point has a strong connection to a point already assigned to a neighborhood, this is the neighborhood to which it is added. Finally, any remaining fine points should be grouped into aggregates consisting of strongly coupled sub-neighborhoods. It should be noted that often the sub-neighborhoods often only consist of one point. However, this third step of aggregation is rarely executed since all nodes are usually added to a neighborhood in the first two steps.

The third and final coarsening strategy is an element-based strategy. The particular algorithm used in AMGLab is known as AMGm and was recently proposed by Kraus [6]. Unlike the previous two coarsening strategies, AMGm reads in element matrices which describe the problem's discretization mesh. These element matrices are then converted into analogous matrices which describe the mesh on the edges of the discretization rather than on the elements themselves.

The AMGm coarsening algorithm takes a similar approach to coarse point selection as the Ruge-Stueben approach in that heuristics are used to evaluate each node in the working grid. However, AMGm is an element-based algorithm which is appropriate for finite element discretizations. These FEM grids are input in the form of element connectivity matrices which vary in dimension depending on the shape of the elements. For example, elements having three vertices will be described by 3×3 matrices. The AMGLab implementation supports elements having three vertices and six vertices. These structures are stored in Matlab as a structure having three dimensions. This presents a special problem with the solution system which relies on a 2D system stiffness matrix. To execute the smoothing and solution phases of the code, the element matrices, which are 3 dimensional, can be accumulated into the 2 dimensional system stiffness matrix using an auxiliary algorithm provided Kraus [6]. Another major difference in dealing with FEM problems is that of the intergrid transfer weights. As opposed to Ruge-Stueben and Smoothed Aggregation, which use a threshold method to generate the transfer weights, FEM edge matrices must be given special treatment in applying the weights. In the case of AMGm, the transfer weights are calculated via multilevel Schur Complements as proposed by Kraus [7].

This concludes the discussion of the core algorithms implemented in AMGLab. There are many other subroutines included in AMGLab which contribute to the project in an auxiliary manner. These subroutines include, but are not limited to, GUI components, smoothers and linear solvers, switching logic to execute user selected options, and scripts to set up system infrastructure. All of these algorithms exist in support of the core algorithms and are very important to the operation of the AMGLab system. Here we will provide short descriptions of some of these support algorithms in the context of what they do in the AMGLab system. The algorithms hereafter discussed are ones that AMGLab users and contributors alike may be interested in understanding in greater detail.

The global solution algorithm used in AMGLab is of great importance to the system. This algorithm, which is located in the file "amg_solve.m" is the root subroutine which is called to start up a solution process using the current settings. This is done in three primary phases. The first phase sets up all solution-specific storage such as the system matrix by calling the appropriate primer script. There are three primer scripts built into AMGLab. The primer "amg_example1" sets up the solution storage with a 2D Poisson example of specified size. The "amg_usersetFD" primer reads in a user specified file containing a finite difference discretization and sets up the solution storage accordingly. The "amg_usersetFEM" performs the same task on the user's finite element file. Once the system is primed, the second phase is setup. This is done by a call to "amg_setup" which will be discussed next. Finally, the V-Cycle algorithm is started on the first (finest) level via a function call. The amg_solve algorithm here contains a loop in case the user has selected to run subsequent passes through the V-Cycle. In addition to these core functions, the amg_solve algorithm communicates what is happening to the user via the GUI status window and utilizes the Matlab profiler to collect statistical information on the performance of the system.

The setup phase for algebraic multigrid is done in AMGLab by the "amg_setup" algorithm located in the file "amg_setup.m". This algorithm has two primary features. First, it checks to make sure we are not at the coarsest level. If not, the user's selected coarsening strategy is invoked to select our coarse grid points and produce the intergrid transfer operators. Second, the algorithm checks to see if it is doing up-front setup. If so, then the algorithm recursively calls itself on the next coarsest level to continue the coarsening.

During the execution of the V-Cycle itself, there are several algorithms which are used to provide primary services for the AMGLab system. These algorithms, which are invoked from within the V-Cycle are called smooth, residual, restrict, and interpolate respectively. The algorithm in "smooth.m" simply invokes the iterative method selected by the user on the algebraic system at the current level. In "residual.m" is a simple calculation of the residual of our approximation at the current level in the usual manner of subtracting our computed solution from the actual solution (right hand side). The restriction performed in "restrict.m" is slightly more complicated. By using the intergrid transfer weights calculated during setup, both the system matrix and residual are restricted onto the next coarsest grid. Finally, the interpolation step performed in "interpolate.m" by directly applying the interpolation weight to our computed residual solution from the next coarsest grid. This allows us to apply the interpolated result as a correction to the current approximation, which is done in the V-Cycle.

This concludes the detailed discussion of the algorithms implemented in AMGLab. It is hoped that this information will act as a guide to future users and contributors of AMGLab.

4. From Design to Implementation. To begin the project, we considered using the framework from MGLab as it was. We identified some design problems that we could vastly improve on. Hence, we decided to start fresh with a new infrastructure. There were two major differences that were implemented in the AMGLab system. First, MGLab used global flag variables to keep track of user option selections. These global flags shared the name of the subroutine that they represented with a different capitalization pattern, making the code difficult to trace and at times executing unintentional function calls. We decided to use global flags, but enforced a different naming convention. Second, and most importantly, MGLab used global variables to hold variables for the solution system. Each level of the coarsening had a separate variable in which to store its system matrix, residual, right hand side, etc. In contrast, AMGLab uses Matlab structures to store this information. By using the level number to index into these structures a subroutine can access the solution information for that level. This provides a much more general framework for accessing and modifying vital information about the problem. It also allows more varied information to be stored, such as element matrices for FEM discretizations. In the first versions of AMGLab, the method of using simple global variables was used but became inadequate when trying to add AMGm to the system.

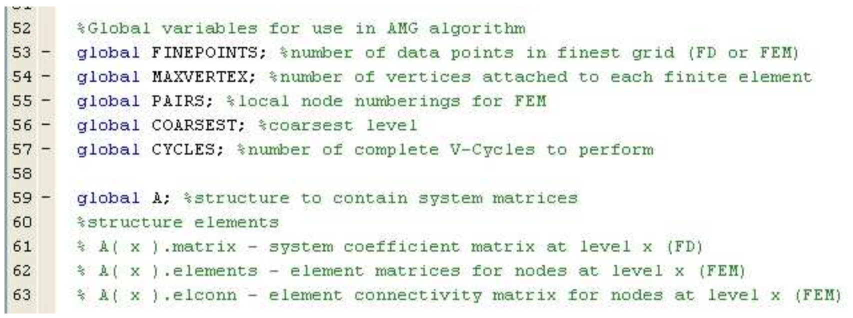
The AMGLab implementation began with one coarsening strategy, namely Ruge-Stueben. We implemented two smoothers (Jacobi and SOR iterations) to perform the smoothing at each stage and stored all options and solution-relevant data in global variables. These variables included the system matrix for each level, the residual history, and the computed solution vector. At this point, the entire solution system was run on the command line via a script and there were few options available. The system could use one of the two different smoothers, select between three coarse grid solution methods, and modify iteration stopping criteria. Each option was changed by changing the value of a global option variable in the system code.

At this point, we began implementing the graphical user interface. Once we had a functioning GUI which enabled users to modify options and make subsequent test runs, the next task was to map out all the options we desired for the AMGLab system. Once the desired scope of the system was agreed upon, those options were added to the GUI and their corresponding global option variables were added to the infrastructure.

The next task was to add code to fulfill all of these options. Many of these options, such as stopping criteria, setup placement, etc. simply required modification of existing code. However, the core option in AMGLab, namely multiple coarsening strategies, still had to be implemented.

In adding the second and third coarsening strategies, the existing system for storing solution data became inadequate. This is the point at which we decided to use Matlab structures to store the varied information needed by the different coarsening strategies. For instance, instead of referencing the variable "A" for the system matrix at the current level, you use the level number as an index. In the current implementation "A(level).matrix" references the system matrix at the specified level, "A(level).elements" references the collection of element matrices at the current level. This framework is much more flexible, especially with the dynamic typing of Matlab. If a new construct needs to be stored in the A structure for some future extension, it need only be named and used in a consistent manner to the existing constructs. The next section of this paper explains in detail what steps must be taken to extend AMGLab by adding new algorithms.

5. Extending the AMGLab System. This section is intended mostly for the use of contributors to the AMGLab system. Here, we will provide some guidance to contributors for adding their own algorithms into the system. There are four main steps which must be taken in order to fully integrate a new feature into the system. The first step is to look into the system infrastructure to determine if there is currently sufficient support for your intended option or algorithm. Even if there are sufficient storage constructs available for your algorithm, you must add an option variable in the appropriate section for your addition. The core system code for storage and options is contained in the file "amg_globals.m". For instance, when adding AMGm to the system, we had to modify the core system in order to store the three dimensional matrices of element connectivity. See Fig.5.1 for the code we added to "amg_globals.m" in order to add this support.

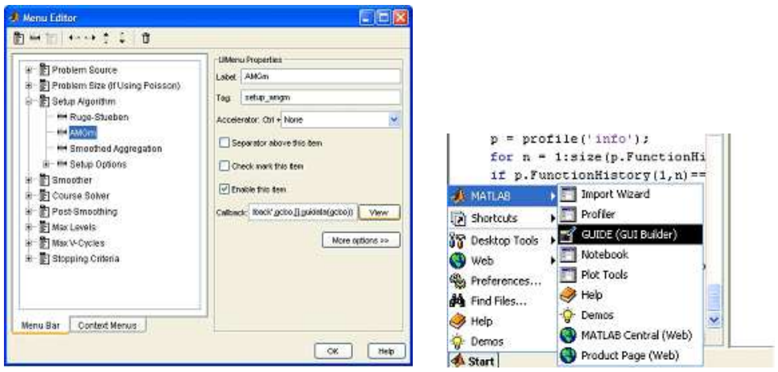


```
%Global variables for use in AMG algorithm
global FINEPOINTS; %number of data points in finest grid (FD or FEM)
global MAXVERTEX; %number of vertices attached to each finite element
global PAIRS; %local node numberings for FEM
global COARSEST; %coarsest level
global CYCLES; %number of complete V-Cycles to perform

global A; %structure to contain system matrices
%structure elements
% A( x ).matrix - system coefficient matrix at level x (FD)
% A( x ).elements - element matrices for nodes at level x (FEM)
% A( x ).elconn - element connectivity matrix for nodes at level x (FEM)
```

FIG. 5.1. *Excerpt from amg_globals.m*

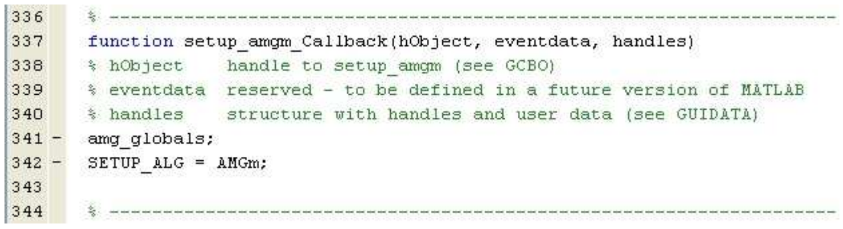
The second step is optional if you are only adding an algorithm for your personal testing or research. This step is to add your option or algorithm to the graphical user interface of AMGLab. This step is done in two phases. The first phase is to edit the setting window portion of the GUI either manually or by using the GUIde editor. This setting window is contained in the two files “settingsGUI.m” and “settingsGUI.fig.” You must edit the .fig file to add your option or algorithm to the list of available options in the menu system. Then, you must edit the .m file to add an event handler for your new menu item. The event handler should set the appropriate option variable which you created in step one. See Fig. 5.2 for how to start GUIde. Once you have opened the settingsGUI, go to the menu editor and add an appropriate menu item. Click “view” next to the listed event handler for your menu option to edit the event handler. See Fig. 5.3 for an example. Now note in Fig. 5.4 that the menu item, the event handler, and the global option variable are all corresponding.



Label: AMGm
Tag: setup_amgm
Accelerator: Ctrl + None
Separator above this item
Check mark this item
Enable this item
Callback: lback',gcbo,[],guidata(gcbo))

```
p = profile('info');
for n = 1:size(p.FunctionHi
if p.FunctionHistory(1,n)==
```

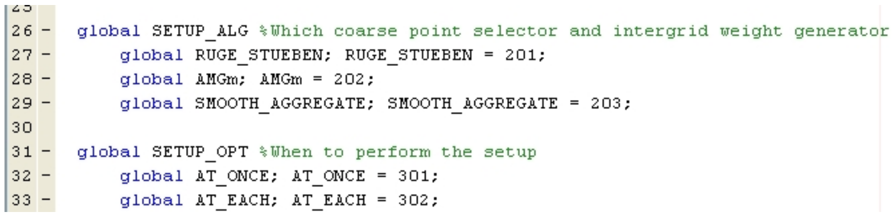
FIG. 5.2. *GUIde*



```
% -----------------------------------------------------------------------
function setup_amgm_Callback(hObject, eventdata, handles)
% hObject    handle to setup_amgm (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
amg_globals;
SETUP_ALG = AMGm;

% -----------------------------------------------------------------------
```

FIG. 5.3. *The Event Handler*

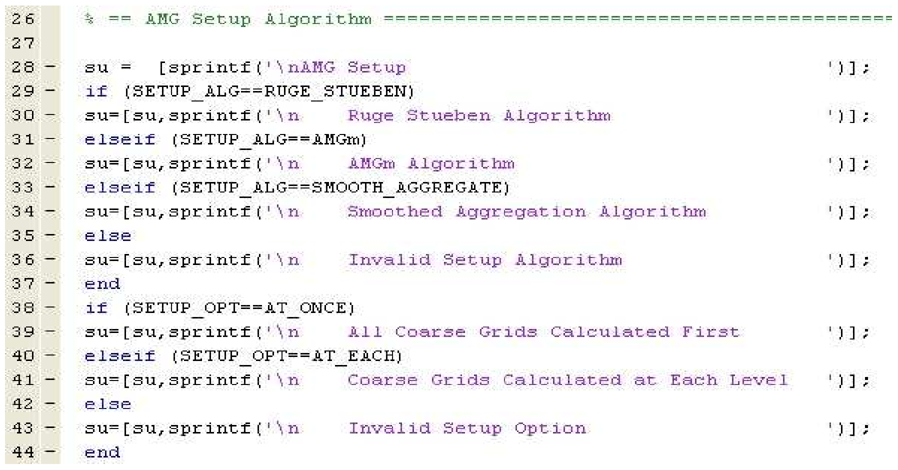


```
global SETUP_ALG %Which coarse point selector and intergrid weight generator
    global RUGE_STUEBEN; RUGE_STUEBEN = 201;
    global AMGm; AMGm = 202;
    global SMOOTH_AGGREGATE; SMOOTH_AGGREGATE = 203;

global SETUP_OPT %When to perform the setup
    global AT_ONCE; AT_ONCE = 301;
    global AT_EACH; AT_EACH = 302;
```

FIG. 5.4. *The User Option Variable in amg_globals.m*

Phase two of this step requires you to edit the “get_settings.m” file. This will give users a visual confirmation of your option or algorithm. As you can see from Fig. 5.5, once you have opened this file the code in here is laid out to mirror he option variable system you saw in the amg_globals file. Once you have located the appropriate place to add an output line for your new option and added it in, you have finished updating the GUI components of AMGLab to accommodate your new contribution.

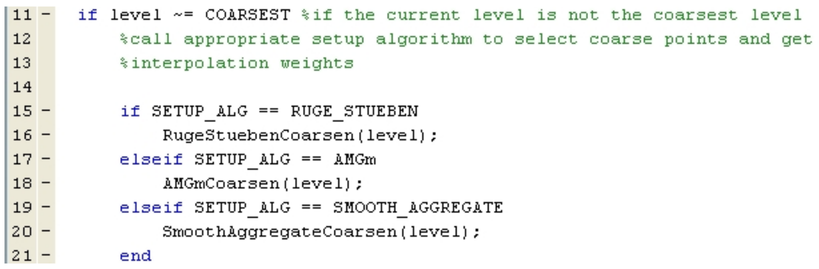


```
% == AMG Setup Algorithm ==========================================

su =  [sprintf('\nAMG Setup                                        ')];
if (SETUP_ALG==RUGE_STUEBEN)
su=[su,sprintf('\n    Ruge Stueben Algorithm                       ')];
elseif (SETUP_ALG==AMGm)
su=[su,sprintf('\n    AMGm Algorithm                               ')];
elseif (SETUP_ALG==SMOOTH_AGGREGATE)
su=[su,sprintf('\n    Smoothed Aggregation Algorithm               ')];
else
su=[su,sprintf('\n    Invalid Setup Algorithm                      ')];
end
if (SETUP_OPT==AT_ONCE)
su=[su,sprintf('\n    All Coarse Grids Calculated First            ')];
elseif (SETUP_OPT==AT_EACH)
su=[su,sprintf('\n    Coarse Grids Calculated at Each Level        ')];
else
su=[su,sprintf('\n    Invalid Setup Option                         ')];
end
```

FIG. 5.5. *Modifying "get_settings.m"*

The third step is to locate the appropriate switching point for your option or algorithm and add logic to that switch so that your code will be executed when the system runs. For instance, if you are adding a smoother, the switch file is "smooth.m". If you are adding a new coarsening strategy, the switch file is "amg_setup.m". See Fig. 5.6 for an example of adding AMGm into the switching algorithm.



```
if level ~= COARSEST %if the current level is not the coarsest level
    %call appropriate setup algorithm to select coarse points and get
    %interpolation weights

    if SETUP_ALG == RUGE_STUEBEN
        RugeStuebenCoarsen(level);
    elseif SETUP_ALG == AMGm
        AMGmCoarsen(level);
    elseif SETUP_ALG == SMOOTH_AGGREGATE
        SmoothAggregateCoarsen(level);
    end
```

FIG. 5.6. *Modifying "amg_setup.m"*

The fourth and final step in your process would be to implement your addition to the system, as well as any utility functions necessary, in Matlab. When implementing your code, please take a look at already implemented codes which perform the same task as yours. In particular, be careful to obey the pre and post conditions for similar algorithms. For instance, any coarsening strategy must at minimum generate the coarse grid system (or stiffness) matrix and intergrid transfer weights in order for the system integrity to be satisfied. Without these components, other parts of the system, such as smoothing, coarse level solving, restriction, and interpolation could fail.

Finally, we offer a couple of general tips concerning the modification of this system. As AMGLab is a complex system consisting of many intricately interlaced components, please do not attempt to modify or add to the system without first devoting some time to understanding the system. Also, modification of this system should only be attempted by individuals familiar with multilevel algorithms and programming.

6. Conclusions, Extensions, and Future Work. In conclusion, we believe that the AMGLab project is an adequate first step toward our eventual goal of an algebraic multigrid expert system. This project will give the multigrid community an immediately useful tool to experiment with AMG algorithms while allowing our research to continue in preparing a fully featured expert system. AMGLab contains a minimal set of options for users to experiment with. In future versions of the project it is hoped that our group along with contributors from the community will add most of the known AMG solution strategies to the system. One of the algorithms we wish to add in the near future is the original version of smoothed aggregation [4]. Also, we wish to add one of the true AMGe algorithms [1], [5]. Additionally, AMGLab will soon feature a general AMG cycle through which the user can select the cycle "shape" as discussed in the description of core algorithms. (i.e. V-Cycle, W-Cycle, etc.)

In parallel to adding a variety of prototype-quality codes to the AMGLab system, we also will compile a library of corresponding commercial grade codes in serial and parallel for users to migrate to once they have chosen appropriate options within the prototyping system.

References.

[1] M. Brezina, A.J. Cleary, J.W. Ruge, et al, Algebraic Multigrid based on Element Interpolation (AMGe). SIAM J. Sci. Comput. 22 (2000), 1570-1592
[2] M. Brezina, R. Falgout, J.W. Ruge, et al, Adaptive Smoothed Aggregation (aSA). SIAM J. Sci. Comput. 25 (2004), 1896-1920.
[3] W.L. Briggs, V.E. Henson, S.F. McCormick, A Multigrid Tutorial. Second Edition. SIAM Books, Philadelphia, 2001.
[4] T.F.Chan, P. Vanek, Detection of Strong Coupling in Algebraic Multigrid Solvers. In Multigrid Methods VI, vol. 14, Springer-Verlag, Berlin, 2000, 11-23.
[5] G. Haase, U. Langer, S. Reitzinger, et al, Algebraic Multigrid Methods Based on Element Preconditioning. Intl. J. of Computer Mathematics 78, #4 (2001), 575-598.
[6] J.K. Kraus, J. Schicho, Algebraic Multigrid Based on Computational Molecules. RICAM Report 2005-20, Johann Radon Institute for Computational and Applied Mathematics, Austrian Academy of Sciences.
[7] J.K. Kraus, Computing Interpolation Weights in AMG Based on Multilevel Schur Complements. Computing 74 (2005), 319-335
[8] J.W. Ruge, K. Stueben, Efficient Solution of Finite Difference and Finite Element Equations by Algebraic Multigrid (AMG). In Multigrid Methods for Integral and Differential Equations, D.J. Paddon and H. Holstein eds, The Institute of Mathematics and its Applications Conference Series, Calerdon Press, Oxford, 1985, 169-212
[9] J.W. Ruge, K. Stueben, Algebraic Multigrid (AMG). In Multigrid Methods, vol. 3 of Frontiers in Applied Mathematics, S.F. McCormick ed, SIAM, Philadelphia, 1987, 73-130.
[10] K. Stueben, An Introduction to Algebraic Multigrid. Appendix A In Multigrid, U. Trottenberg, C. Osterlee eds, Academic Press, London, 2001, 413-528
[11] U. Trottenberg, C. Osterlee, A. Schuller, Multigrid. Academic Press, London, 2001
[12] P. Vanek, J. Mandel, M. Brezina, Algebraic Multigrid based on Smoothed Aggregation for Second and Fourth Order Problems. Computing 56 (1996), 179-196.