An Investigation on the Optimal Parameters of the SOR and GMRES Solvers

0.1 Work Breakdown

Class Taught by Professor Fournier and sponsored by Dr. Stephen Thomas

Jacob Petersen: Data generation, building the AMG solver, computation of optimal strength of connection/relaxation parameter using fixed tolerance value and loops, optimal configuration of the post and pre-iterations for omega, and miscellaneous formatting and code optimization.

Alex Semyonov: Computation of the theoretical optimal omega value, optimal configuration of the post and pre-iterations for theta, estimating theta using the standard form of a continuous optimization problem, miscellaneous formatting and code optimization.

Christian Stout: Machine learning research and miscellaneous formatting and code optimization.

0.2 Abstract

Algebraic multigrid (AMG) is an efficient method often utilized to solve large, sparse systems of equations. This method is advantageous relative to other multigrid methods as it does not require any geometric information and can be readily applied to various problems. The primary goal of this project is to determine the optimal parameters which enable the AMG to solve a system of equations with maximum efficiency. While working on this project, the team determined an optimal value for the relaxation parameter for the SOR algorithm and determined an optimal value for the strength of parameter for the GMRES solver.

0.3 Introduction

NREL (The National Renewable Energy Laboratory) is interested in the optimal construction of wind turbines. Flow dynamics are a particularly important factor to consider when constructing these wind turbines (as the air flow generated by a turbine may influence surrounding turbines). This air flow can be represented using the incompressible Navier Stokes equations which, when discretized, result in an exceptionally large system of equations. Solving these equations can often be very computationally taxing and time consuming. In order to efficiently solve this resulting system of equations, NREL has investigated the performance of several AMG methods in "A Comparison of Classical and Aggregation-Based Algebraic Multigrid Preconditioners for High-Fidelity Simulation of

Wind Turbine Incompressible Flows" by Thomas et al. This paper also examines the incompressible Navier-Stokes equations and their discretization. There are several parameters which may influence the computing speed and the generated residuals. By identifying the optimal parameters for a particular solving algorithm, one can reduce the number of iterations necessary to find a solution and minimize the residual associated with this solution.

Ideally, the methods utilized to identify the optimal parameters can be generalized to other projects and perhaps can illuminate efficient methods to optimize the parameters of a given solver. As the discretization of partial differential equations often results in a large system of equations, identifying and optimizing the parameters which influence the iterations and residuals is an important task which may drastically influence computing requirements and solver accuracy.

Our general approach to this optimization problem can be divided into four essential steps:

1. Generating the data (an *n x n* matrix **A** and an *n x 1* vector **b**).
2. Defining and building the appropriate solver (specifically one utilizing the SOR algorithm and another utilizing GMRES).
3. Optimizing the appropriate parameters using a brute-force approach (cycling through various parameter values and plotting residuals for each parameter)
4. Optimizing our method for determining the optimal parameter.

When possible, the group computed optimal parameters both mathematically and numerically (although this was only possible with the SOR algorithm). This enabled the group to identify and explain potential discrepancies between the numerical and theoretical results. This, however, was more of a side-step than a progression step and thus is not included as an essential step.

1 Methods

Generally, the method for each set of experiments use a `for` loop to vary the desired settings (omega, theta, pre-, and post-iterations) while constructing and running the AMG solver for each setting combination.

`For` loops are used to construct and run the solver for each combination of settings--where the `for` loop is changing the desired setting, it mostly runs multiple values of omega or theta. If multiple nested for loops are used, it is to account for the combinations of pre- and post-smoother iteration settings and how those affect the solver as well.

For the SOR solver, the theoretical optimal relaxation parameter was computed mathematically (using eigenvalues). For both the SOR and GMRES solvers, the optimal parameters (θ and ω) are also calculated by minimizing an objective function with scipy.optimize.minimize_scalar.

After computing an optimal parameter, various visual graphs are implemented to demonstrate how

```
# https://github.com/pyamg/pyamg
!pip install pyamg


import scipy
import numpy
import pyamg
import pylab
import random
import pandas as pd
import seaborn as sb
import matplotlib.pyplot as plt
from matplotlib.pyplot import figure
from matplotlib import cm
from random import seed
from random import randint

#Make sure this runtime is connected to a tf device
#Edit -> Notebook Settings -> Hardware accelerator: GPU
import tensorflow as tf
from tensorflow import keras
from keras import layers

device_name = tf.test.gpu_device_name()
if device_name != '/device:GPU:0':
  print('No tf GPU found')
  raise SystemError('GPU device not found')
print(device_name)

#Global Variables for consistancy in seed, size and tolerance level
seednum= 5
size = 100
tol = 1e-13
```

1.1 Problem Generation and AMG Construction

After importing all of the necessary packages, the next step was to create matrices to run experiments on. To do this the `generate_problem` function was written, which produce a semi-random, sparse, and diagonally dominant matrix. This matrix is based off of a global seed variable to maintain consistency across all experiments.

The next function is the `build_sor_amg` which is the function that is used when running experiments concering ω, where the ω value and the pre- and post-smoother iterations can be set when the function is called.

Finally, the last function, `build_theta_amg`, operates in the same way as previous function, except

```
def generate_problem(seednum, size, verbose=False):
  seed(seednum)

  diag = (randint(-size, size))+1000  # this is the random diagonal generated for the

  a = randint(-(abs(diag//6)),(abs(diag//6))) # these variables are the random stencil
  b = randint(-(abs(diag//6)),(abs(diag//6)))
  c = randint(-(abs(diag//6)),(abs(diag//6)))
  if verbose:
    print(f'A= {a}\nB= {b}\nC= {c}\nDiagonal= {diag}')

  stencil = [[a ,0 ,b], [c, diag, c], [b, 0, a]] # This stencil is set to be mostly sp
  A = pyamg.gallery.stencil_grid(stencil, (size,size), dtype=float, format='csr')
  A = .5*(A + A.transpose())                  # This makes A symmetric
  B = numpy.ones((A.shape[0],1))
  return A, B, diag

def build_sor_amg(A, B, omega, preIts, postIts, maxCourse=10, verbose=False):
  ml = pyamg.smoothed_aggregation_solver(
      A, B, max_coarse=maxCourse,
      presmoother=('sor', {'sweep':'symmetric', 'omega':omega, 'iterations' : preIts})
      postsmoother=('sor', {'sweep':'symmetric', 'omega':omega, 'iterations' : postIts

  if verbose:
    print (ml)
  b = numpy.random.rand(A.shape[0],1)
  x0 = numpy.random.rand(A.shape[0],1)
  return ml, b, x0

def build_theta_amg(A, B, theta, preIts, postIts, maxCourse=10, verbose=False):
  ml = pyamg.smoothed_aggregation_solver(
      A, B, max_coarse=maxCourse,
      strength=('symmetric',{'theta':theta}),
      presmoother=('gmres', {'maxiter': preIts}),
      postsmoother=('gmres', {'maxiter' : postIts}))

  if verbose:
    print (ml)
  b = numpy.random.rand(A.shape[0],1)
  x0 = numpy.random.rand(A.shape[0],1)
  return ml, b, x0
```

1.2 Testing Omega with Default Pre and Post Iterations Values

Using the two functions that are defined above, the problem matrix is generated and the SOR-based AMG is constructed. Then using a command from the PyAMG package we can just run the solver on our defined problem. It is in this step that we are varying ω, the relaxation parameter for SOR. The optimization of ω is the first step in optimizing the efficiency of the AMG solver with SOR pre- and post-smoothing methods.

The program starts ω at 0.05 and ends at 2.0, testing every value of ω with a step size of 0.025. In these tests, the values for the pre- and post-iterations are set to their default values of 1. This is just to analyze how much a change in ω can influence the resulting residuals and iteration counts.

```
#This cell may take some time (typically under 30 seconds) to run
omega_experiments = []
A, B, diag= generate_problem(seednum, size)
for omega in numpy.linspace(0.05, 2.0, 80):
  ml, b, x0 = build_sor_amg(A, B, omega, 1, 1)

  residuals = []
  x = ml.solve(b=b, x0=x0, tol=tol, residuals=residuals)

  omega_experiments.append((omega, residuals))
```

2 Computing the Optimal Parameters

2.1 Calculating the Theoretical Optimal ω

When working with the SOR algorithm, ω denotes the relaxation parameter. This relaxation parameter can drastically influence convergence rate. ω can be mathematically calculated by finding the spectral radius of the Jacobi iteration matrix and utilizing a formula which allowed us to directly compute the optimal relaxation parameter. There are several assumptions made in this calculation, namely: (1) The Jacobi Iteration Matrix has real eigenvalues, (2) that Jacobi's method is convergent, and that a unique solution exist. Furthermore, our eigenvalue computation relies on A being symmetric. If A is not symmetric, the eigenvalue computation can be changed to eigs() but will take longer to compute.

```
A, B, diag = generate_problem(seednum, size)
jacobi=scipy.sparse.identity(size*size)-A/diag #Definition of Jacobi matrix
max_eigenval =scipy.sparse.linalg.eigsh(jacobi, k=1, which='LM',return_eigenvectors=Fa
print("Largest Magnitude Eigenvalue: ", max_eigenval)
optimal_omega = 1 + (max_eigenval/(1+numpy.sqrt(1-max_eigenval**2)))**2#Formula for op
```



```
print("Optimal Omega: ", optimal_omega)
```

2.2 Continuous Optimization of ω

The following code block finds the optimal ω by first defining an objective function: ω2iter(ω) which returns the iteration count at which the residual falls below the desired tolerance level. The function then finds the ω that minimizes the iteration count. Comparing the produced value to the theoretical omega demonstrates that the difference between the two values is rather miniscule, which suggests that the algorithm can accurately estimate the optimal relaxation parameter.

```
#This code was made by prof Fournier, I posted it here to try to disect it and compare
from scipy.optimize import minimize_scalar
A, B, diag= generate_problem(seednum, size)
#This is our objective function
def ω2iter(ω) :
  ml, b, x0 = build_sor_amg(A, B, ω, 1, 1)
  residuals = []
  x = ml.solve(b=b, x0=x0, tol=tol, residuals=residuals)
  iterCount = numpy.nonzero([r == 0#< tolerance is set to 0
                             for r in residuals])[0] # 0th of all indexes of all resic
  if iterCount.size > 0 :
    return float(iterCount)
  else :
    return float(len(residuals))
res_omega = minimize_scalar(ω2iter, bounds=(0.05, 2.0), method='Bounded', tol=0) #finc
res_omega
```

2.3 Using the Theoretical Optimal ω To Find the Best Iteration Settings

In this code, we have fixed the relaxation parameter to be the theoretical optimal parameter calculated above. This allows us to identify which pre/post-iteration settings are optimal in terms of speed and accuracy. To do this, the AMG construction and solver are repeated 25 times for all possible combinations of pre- and post-iterations while keeping the ω value constant between experiments.

```
iteration_experiments = []
A, B, diag= generate_problem(seednum, size)
for preIts in numpy.arange(1, 6, 1):                                    #Pre iterations lc
  for postIts in numpy.arange(1, 6, 1):                                 #Post iterations ]
    ml, b, x0 = build_sor_amg(A, B, optimal_omega, preIts, postIts) #Optimal omega frc
```

```
    residuals = []
    x = ml.solve(b=b, x0=x0, tol=tol, residuals=residuals)

    iteration_experiments.append((preIts, postIts, residuals))
```

2.4 Testing to find the Optimal θ

The optimal θ for running the solver cannot be calculated theoretically in a similar manner to the optimal ω.

Instead the use of a brute force method is applied, in which a number of experiments are done for each combination of pre- and post-smoother settings. There are 25 different combinations for the pre- and post-smoother settings, which can have an integer value from 1 to 5 each, and the range of θ spans 0.0 to 0.6. For every comination of pre- and post- iterations, the program will construct and solve an AMG for every θ value from 0.0 to 0.6 with steps of 0.01.

After each solution is finished, the residuals list is passed to a new array to store the values along with the current θ and pre- and post-iterations.

```
# THIS TESTS OVER 1500 SETTINGS COMBINATIONS, TAKES A WHILE TO RUN#

theta_experiments = []
hm_data = []
hm_data_all = []
seednum= 5
A, B, diag = generate_problem(seednum, size)

for preIts in numpy.arange(1, 6, 1):                    # Pre iteration loop
  for postIts in numpy.arange(1, 6, 1):                 # Post iteration loop

    temp_resid_len = []
    temp_theta_store = []

    for theta in numpy.linspace(0.01, .60, 60):         # Theta value loop
      ml, b, x0 = build_theta_amg(A, B, theta, preIts, postIts)

      residuals = []
      x = ml.solve(b=b, x0=x0, tol=tol, residuals=residuals)

      # Array stores ALL data for display later
      theta_experiments.append((theta, preIts, postIts, residuals))

      # Temporary storage for each combination of pre and post its
      temp_theta_store.append((theta, preIts, postIts, residuals/residuals[0]))

      #  Temporary storage of length for all thetas in current combination of pre and
      temp_resid_len.append(len(residuals))
```



```
    # These two lines find the theta with the fewest iterations for each pre and post
    minpos = temp_resid_len.index(min(temp_resid_len))
    the, pre, post, resid = temp_theta_store[minpos]

    # Storage for the 25 optimal thetas and their respective results for use in displa
    hm_data_all.append((the, pre, post, resid[-1]/resid[0], len(resid)))
    hm_data.append((the, pre, post))
```

2.5 Continuous Optimization Problem θ

This code block once again defines an objective function: Θiter(Θ). Which is a function of θ that returns the iteration count (to hit desired tolerance). This function is then minimized using scipy.optimize.minimize_scalar(). The produced theta solution always tends towards the upper bound and differs from the estimated theta value discovered by utilizing the "brute-force" approach above.

```
from scipy.optimize import minimize_scalar
A, B, diag = generate_problem(seednum,size)
def Θiter(Θ):             #Objective function definition

  m1, b, x0 = build_theta_amg(A,B,Θ,1,1) #Varying pre/post iteration essentially has
  residuals = []
  x = ml.solve(b=b,x0=x0,tol=tol,residuals=residuals)
  iter_count = numpy.nonzero([r < tol for r in residuals])[0]
  if iter_count.size > 0 :
    return float(iter_count)
  else :
    return float(len(residuals))

res_omega = minimize_scalar(Θiter, bounds = (0.01, 0.6), method = 'Bounded', tol=0) #
res_omega
```

2.6 Pre- and Post- iterations with a Fixed θ

The section above tests all values for theta against every combination of pre- and post-iterations, but that also takes a long time to run. This section allows the user to run an individually chosen theta value on all combinations of pre- and post-iteration values. The main reason we left this in is to test any value that might be of interest, or to test fewer values by hand, which will result in much faster response times for ease of use.

```
theta_iteration_experiments = []


theta=0.10
```

```
theta=0.10
A, B, diag= generate_problem(seednum, size)
for preIts in numpy.arange(1, 6, 1):
  for postIts in numpy.arange(1, 6, 1):
    ml, b, x0 = build_theta_amg(A, B, theta, preIts, postIts)
    residuals = []
    x = ml.solve(b=b, x0=x0, tol=tol, residuals=residuals)

    theta_iteration_experiments.append((preIts, postIts, residuals))
```

3 Results

Seperated into different categories below are the results, and their graphs, for each experiment run above.

3.1 Testing Omega with Default Pre and Post iterations Values: Results

Once the AMG solver has run for all of the desired values of ω, all of the resulting residuals will be graphed according to their total number of iterations for a solution.

Below the graph, the optimal omega value from all of the tested values is printed. In this case, the optimal ω was determined by sorting the list of residual arrays by the length of the residual arrays. This gives the ω value that corresponds to the fewest number of residuals, or fewest iterations.

```
residuals_len =[]
figure(figsize=(18, 12))
plt.xlim(0,20)

# Loop to graph each residuals list from the experiments
for experiment in omega_experiments:
  omega, residuals = experiment
  #Fournier added the edits below, the color is sorted, fewest iterations are light gr
  pylab.semilogy(residuals/residuals[0], '.-', label=f"ω: {omega:.2f}",
                 c=cm.rainbow((omega - omega_experiments[0][0])/(omega_experiments[-1]
  residuals_len.append(len(residuals/residuals[0]))

# Graph Labels
pylab.xlabel('iterations')
pylab.ylabel('normalized residual')
pylab.legend(loc='upper right',ncol=4)
pylab.show()

# Just some useful information as well as the optimal choice for Omega in regards to i
minpos = residuals_len.index(min(residuals_len))
print(f'Lowest iteration count = {min(residuals_len)} \n')
```



```
print(f'All iteration counts = {residuals_len} \n')
print(f'Optimal Omega index = {minpos} \n')
print(f'Optimal Omega value = {omega_experiments[minpos]}')
```

3.2 Using Theoretical Optimal ω to find Iteration Settings: Results

The graph below suggests that there is a connection between the pre- and post-smoother iterations, and how many overall iterations it takes for the AMG solver to finish. It seems that as the pre- and post-iterations both increase, the number of overall iterations decreases. Unfortunately, a problem was identified in the code block below just prior to the end of the term and the group was unable to correct it in time. Changing the direction of the for-loop, that is, looping through iteration_experiments in the reverse order produces a different result. This is troubling as the answer produced should be the same regardless of the order of the loop and must be investigated further.

```
residuals_len =[]
figure(figsize=(20, 16))
plt.xlim(0,20)

for experiment in iteration_experiments:
  preIts, postIts, residuals = experiment
  residuals_len.append(len(residuals))
  pylab.semilogy(residuals/residuals[0], '.-', label=f"Pre Its: {preIts} Post Its: {po
  c=cm.rainbow((preIts-iteration_experiments[0][0])/(iteration_experiments[-1][0]-iter
  marker='${}$'.format(postIts))

pylab.xlabel('iterations')
pylab.ylabel('normalized residual')
pylab.legend(loc='best',ncol=2)
pylab.show()

minpos = residuals_len.index(min(residuals_len))
print(f'Lowest iteration count = {min(residuals_len)} \n')
print(f'All iteration counts = {residuals_len} \n')
print(f'Optimal Iteration Index = {minpos} \n')
print(f'Optimal Iteration Values = {iteration_experiments[minpos]}')
```

3.3 All θ Values Tested for all Pre and Post Iteration Combinations: Results

The graph below shows the results from the experiments in the prior cell. It is graphed using the same method described previously, with one caveat. Despite best efforts, removing the repeating final residual from many of these experiments would also remove many data points that are

important. Any method attempted to remove these values also removes residual values that are below the desired tolerance level, sometimes leaving only values that are above the tolerance level.

The values of θ and pre and post-iterations that led to the repeating of iterations, despite having already hit the tolerance threshold, are quite troublesome. It seems that some part of the AMG package does not terminate the algorithm sometimes, even though it seems to hit the desired tolerance level prior to repeating the last residual a hundred times before stopping. This leads to another problem that came up, there is no actual guarantee in these experiments that the final residual achieved is below the tolerance level. The PyAMG package is designed to solve systems with this tolerance constraint, but there was never a check implemented to make sure this is the case. That is an oversight of the project and something to keep in mind when viewing the below results.

While the method for finding the best θ using the residuals list with the shortest length still functions, some part of the PyAMG package causes the solver to repeat iterations of the last residual. So for the sake of accuracy with the existing values, the residuals list is kept the same.

```
residuals_len = []
figure(figsize=(20, 16))
plt.xlim(0,20)

# Loop to graph each residuals list from the experiments
for experiment in theta_experiments:
  theta, preIts, postIts, residuals = experiment

  # Commented out the legend because it was too long
  pylab.semilogy(
                residuals, '.-', label=f"θ: {theta:.2f}",c=cm.rainbow((theta-theta_ex
                            (theta_experiments[-1][0]-theta_experiments[0][0])))

  residuals_len.append(len(residuals))

pylab.xlabel('Iterations')
pylab.ylabel('Normalized Residual')
#pylab.legend(loc='best')
pylab.show()

# Prints the Theta and Pre/Post iterations combo that used the least iterations to sol
minpos = residuals_len.index(min(residuals_len))
print(f'Optimal Theta index = {minpos} \n')
print(f'Optimal Theta value = {theta_experiments[minpos]}')
print(min(residuals_len))
```


3.4 Heatmap View of the Optimal θ for Each Combination of Pre- and Post-Iterations

First, a small note about the heatmaps, the axes are indexed from 0 to 4 even though the pre and post iteration values are from 1 to 5, so keep that in mind when viewing the results. For example, if you want 3 pre-iterations and 4 post-iterations, look at row 2 column 3.

The heatmap on the left shows the θ value that resulted in the fewest number of iterations for each combination of pre- and post-iterations. Most of these were 0.01, but as the map shows there were a few values that differed.

The second heatmap shows the final residual when the solver is run with the θ settings from the first heatmap.

The final heatmap shows how many iterations it took to reach the final residuals shown in the second heatmap. The problem with these heatmaps lies in the third and final one because of the combinations that led to over 100 iterations. These are not indicative of a proper solution by the solver and so they skew the results shown in the heatmap, since they are not a reliable indicator of how that combination of θ and pre- and post-iterations truly affect the solution. There is a possibility

```
theta_map = numpy.zeros((5,5))
resid_map = numpy.zeros((5,5))
its_map = numpy.zeros((5,5))
k = 0

#for x in hm_data_all:
  #print(x)

for i in numpy.arange(0,5,1):
  for j in numpy.arange(0,5,1):
    the, pre, post, resid, len_resid = hm_data_all[k]

    #print(i, j)

    theta_map[i][j]= the
    resid_map[i][j]= resid
    its_map[i][j]= len_resid

    k += 1

fig, ax = plt.subplots(ncols=3, sharey=False)
fig.set_size_inches(24,6)


the_heat = sb.heatmap(theta_map, annot= True, ax=ax[0], linewidths= 0.5, cmap="YlGnBu'
resid_heat = sb.heatmap(resid_map, annot=True, ax=ax[1], linewidths= 0.5, cmap="YlGnBu
its_heat = sb.heatmap(its_map, annot=True, ax=ax[2], linewidths= 0.5, cmap="YlGnBu")
```

```
fig.suptitle('Optimal Theta Results for All Combinations of Pre- and Post-Iterations')
ax[0].set_title('Optimal Theta')
ax[1].set_title('Last Residual')
ax[2].set_title('Total Iterations')

for ax in ax.flat:
    ax.set(xlabel='Post-Iterations', ylabel='Pre-Iterations')
```

3.5 Pre and Post iterations with a Fixed θ: Results

This code block below graphs the results of fixing a θ value and varying the pre- and post-iterations, in the same manner as the previous graphing cells. Excluding the results cell for section *Testing to Find the Optimal θ*.

The results once again show that there is a relationship between the solver iterations and the number of iterations the pre- and post-smoothers have to run. If the pre- and post-smoothers are allowed more iterations each, then the overall iteration count of the solver goes down.

This is the case if the repeating residuals are excluded, as mentioned above, the solver sometimes continues running even after the desired tolerance is reached. The group was unable to identify the cause for this troubling finding.

```
residuals_len =[]
figure(figsize=(20, 16))
plt.xlim(0,20)


for experiment in theta_iteration_experiments:
  preIts, postIts, residuals = experiment
  residuals_len.append(len(residuals))
  pylab.semilogy(residuals/residuals[0], '.-', label=f"Pre Its: {preIts} Post Its: {po
  c=cm.rainbow((preIts-theta_iteration_experiments[0][0])/(theta_iteration_experiments
  marker='${}$'.format(postIts))


pylab.xlabel('iterations')
pylab.ylabel('normalized residual')
pylab.legend(loc='best', ncol=2)
pylab.show()


minpos = residuals_len.index(min(residuals_len))
print(f'Lowest iteration count = {min(residuals_len)} \n')
print(f'All iteration counts = {residuals_len} \n')
print(f'Optimal Iteration Index = {minpos} \n')
print(f'Optimal Iteration Values = {iteration_experiments[minpos]}')
print(theta)
```


3.6 Discussion

The difference between the tested optimal ω and the theoretical optimal ω is the most interesting result. This result was unexpected, and raised a lot questions as to why the tested ω and the theoretical ω were always different. Looking a bit more into the mathematics of iterative methods, it seems that the optimal theoretical ω is considered optimal in terms of the rate of convergence of the iterative method. The experiments regarding ω revealed the value that reduces the number of iterations for the solver. This does not imply that this value also acts as the optimal ω for the rate of convergence.

The tests involving the strength of connection paramter, θ, also had some interesting results, each experiment showed that there was not a heavy dependence on θ for the solver to run to completion. Any value used seemed to have the same effect on the sparse, diagonally dominant matrices. This might not be the case if given matrices with different properties, which is something to look into for further testing. Generally, increasing both the pre- and post- iterations decreases the solver iterations; this result was achieved regardless of the values of the other parameters (ω, θ).

The strength of connection parameter (θ) was also optimized by minimizing a function that takes in θ and returns the iteration counts required to reach a desired tolerance level. While this approach worked wonderfully for optimizing the relaxation parameter, ω, used in the SOR solver, using it to compute an optimal theta yielded unclear results. The supposed optimal theta value always tends to the upper bound. So for example if the upper bound is set to 0.6, the optimal theta will supposedly be 0.5999999, if the upper bound is set to 0.3 it will 0.299999. This occured regardless of what post/pre-iteration settings were used. While the reason for this was never unequivocally determined it may be due to the construction of the GMRES solver in the PyAMG source code and should be investigated in the future. Since this process accurately computed the optimal relaxation parameter, the group is inclined to believe that the issues lies with the GMRES solver and not the minimize_scalar() function.

4 Conclusions

Due to restrictions present in Google Colaboratory, the group was not able to work directly with the data produced by NREL. This was simply because the data produced by NREL was too large for Colab to handle. After collaborating with both Dr. Aimé Fournier and Dr. Stephen Thomas, the group successfully generated data (a diagonally dominant A matrix and **b** vector for the solver to ingest) and identified which parameters appeared to influence the number of iterations as well as the generated residuals. After identifying these parameters, the group needed to develop a method for

optimizing these data. The group began by using the SOR algorithm to solve a large, sparse system of equations and were able to successfully identify an optimal parameter for the relaxation parameter, ω, mathematically and then proceeded to find the optimal value by fixing a particular tolerance level and varying the relaxation parameter in order to see which ω value minimized the number of iterations. Graphs of the residuals were generated and the group was able to visualize how varying the relaxation parameter changes the associated residual. The group employed a similar method to find the optimal strength of connection parameter (which cannot be easily found mathematically) and proceeded to investigate how other parameters influence the strength of connection parameter as well as the residual and computation times. Specifically, the group investigated how the strength of connection parameter influenced the post and pre-iterations. The group's approach yielded unclear results regarding the optimal settings for the GMRES solver and different methods should be investigated in the future. The group also computed the optimal parameters by minimizing a function which takes in a desired parameter value (either strength of connection or relaxation) and returns the number of iterations needed to reach the desired tolerance. This was done in an attempt to accelerate and increase the accuracy of the group's optimization and yielded somewhat unclear results (with the θ parameter specifically).

There are a few main conclusions to draw from the results of the experiments above. First, it is best to note again that all of these experiments were done using sparse, diagonally dominant matrices, these results might vary if given matrices with different properties.

The first conclusion deals with the pre- and post-smoother iterations. The group initially was unsure of how the pre- and post-smoother iterations would impact the total number of iterations. According to the group's testing, increasing the pre- and post-smoothers to their maximum values appears to decrease the amount of overall iterations needed to reach a solution; increasing the pre- and post-smoothers to their maximum values (or close to their maximum values) resulted in the fewest iterations. This result was seemingly independent of the other settings and were consistent between all experiments. In general, the group's findings suggest that when the pre- and post-iterations increase, the solver iterations decrease. Unfortunately, an error was identified which could not be corrected in the alloted time. The answer for the optimal values differed if the order of the elements in iteration_experiments changed. This insinuates that the group's method is not locating the correct residual and must be addressed.

The second conclusion deals with the SOR relaxation parameter (ω). Specifically, the difference between the optimal ω for reducing solver iterations and the theoretical optimal ω. In all experiments, the tests to find the ω that reduces the solver iterations to the fewest possible produced multiple possible ω values, the program returns the first one it encounters. This means that, the search for the relaxation parameter was not optimal. Despite this, the unique optimal parameter can be reproduced by decreasing the tolerance level. Generally, the group's approach did not initially follow the standard form of a continuous optimization problem but this was addressed towards the end of the project. Defining an objective function with appropriate bounds (which

correspond to the parameter being optimized) and minimizing the function using scipy.optimize.minimize_scalar led to a more rigorous and accurate optimization of the relaxation parameter (SOR), but yielded unclear results when applied to the strength of connection parameter (GMRES). Specifically, the value of the optimal parameter would always tend towards the upper bound of the given range which raises doubts regarding the accuracy of this value.

In regards to the optimal θ value, these data do not seem overly-dependent on the θ value. Varying the θ parameter did not result in large variations in the number of iterations. This is likely due to the type of matrices the solver is using (sparse, diagonally dominant matrices). The data generated by the group suggests that almost all the values of θ seem to result in a solution being found in the same number of iterations. It is important to note that varying the post and pre-iteration settings does lead to variations in the optimal strength of connection parameter.

It is critical to address the discrepancy between the optimal theta calculated using the "brute-force" approach (looping over various post/pre-iteration values and θ) and the optimal θ value generated by minimizing an objective function which takes in a value for θ and returns the iterations necessary to reach the desired residual. While the latter approach is much quicker, the tendency for the optimal θ to consistently be the upper bound is concerning and may have occured for several reasons including: a limitation in the PyAmg source code when building the solver, a quirk in the minimize_scalar() which leads to problems when computing the optimal θ. Perhaps the optimal parameter actually tends to be the upper bound, however the group couldn't identify a reason for this to be the case. The fact that this approach worked seamlessly for the relaxation parameter (in the SOR solver) suggests that this problem is specific to θ which may insinuate an issue in the PyAmg source code and needs to be investigated in the future.

Towards the very end of the semester, it was discovered that the group's program does not locate the correct residual and iteration count (the ones associated with the optimal parameter) effectively. Essentially, the students' method for locating the aforementioned residual would return the first residual in a set of equal-length residuals. This first residual is generally not the one associated with the optimal parameter. Due to time constraints, the group was unable to rework the necessary code. Unfortunately, theta cannot readily be computed mathematically (as is the case for ω) so the group is uncertain of the accuracy of their calculated optimal θ parameter. If future research is conducted, changing the method described about and comparing the value produced with the optimal θ parameter calculated using scipy.optimize.minimize_scalar() may prove to be a useful starting point.

5 Continued Research: Machine Learning

The next step in AMG optimization which couldn't be implemented this semester comes from the field of AI and machine learning. Much of our work here is formulaic enough that a machine can be trained to find the optimal parameters for the optimal smoothing methods with enough training

data. Using the solutions from either NREL's data or our toy data, an algorithm can close in on the optimal θ or ω, removing the need for the time-consuming brute force or exact method.

A method for this supervised learning would resemble a standard linear regression model:

1. Define the input matrix the model will iterate on, the pre-calculated output matrix it will train towards, and whatever parameters the model will vary to get from the input to the output.
2. Have the model run our functions while varying the parameters, narrowing the error between the input and output data with each iteration
3. When the model has passed a certain error threshold, output the optimal parameters it found.
4. Once the model is trained, it can do this without needing to be given the output matrix beforehand.

Keras for TensorFlow and scikit-learn are the two most accessible frameworks for implementing machine learning in Colab/Jupyter. A problem we ran into was getting a TensorFlow instance to accept both the data matrix and the solution matrix to train against. The former is a Scipy matrix and the latter is a Numpy array, and they will both need to be converted into tensors (or the Scikit equivalent) for the machine learning instance to operate on them.

A subset of machine learning is *deep learning*, which uses multiple inference layers to reach its desired outcome, mimicking a human brain. Each additional layer requires lots of computing power, so while machine learning has been around for decades, deep learning is relatively new. This further accelerates the task by using a neural network to have the model learn either supervised or unsupervised. Several kinds of neural networks have come about, including artificial (ANN), convolutional (CNN), recurrent (RNN), recursive (RvNN), and the more novel graph (GNN). A team at the Weizmann Institute in Israel has recently researched using a single GNN to find the optimal prolongation operator for any AMG. Their approach differs from ours by using unsupervised learning to accelerate the prolongation/interpolation step, while ours focuses accelerating the smoothing and iterative method, and would have involved supervised learning for those parameters. Although these approaches are very different, they both accomplish the same goal of solving large lineary systems quickly. Future research may involve replicating or improving this approach, or using it in conjunction with our approach.