Background And Results

January 8, 2022

1 1D Poisson Multigrid Solver: Background and Results

1.1 By Collin Kofroth

2 Mathematical Preliminaries

The 1D Poisson problem with Dirichlet boundaries seeks to find solutions u to the ordinary differential equation

$$\begin{cases} -u''(x) = f(x) & x \in (0,1) \\ u(0) = a \\ u(1) = b \end{cases}$$

where f is a prescribed function and a, b are the prescribed boundary data. This equation has numerous applications, such as giving the steady state solution to the 1D heat equation. One of the more simplistic ways to generate a numerical approximation of the solution to this equation is through the method of finite differences. First, we break the domain $[0,1]$ into n sub-intervals of equal length using grid points $x_j = j\,\Delta x$, where $0 \leq j \leq N$ and $\Delta x = 1/N$. In the interior of the grid (i.e. x_j for $1 \leq j \leq N-1$), we approximate the second derivative of u via the second-order central difference method

$$\frac{u(x_{j+1}) - 2u(x_j) + u(x_{j-1})}{(\Delta x)^2} \approx u''(x_j) = -f(x_j), \qquad 1 \leq j \leq N-1.$$

To this end, we seek a vector v with components v_j which approximates $u(x_j)$ such that

$$\frac{-v_{j+1} + 2v_j - v_{j-1}}{(\Delta x)^2} = f(x_j), \qquad 1 \leq j \leq N-1,$$

and $v_0 = a, v_N = b$. We will write $f_j = f(x_j)$, and $g = (g_0, g_1) = (a, b)$.

The central difference condition generates a system of $N-1$ equations with $N-1$ unknowns. We can write this as a tridiagonal linear system

$$\frac{1}{(\Delta x)^2}\begin{bmatrix} 2 & -1 & & & \\ -1 & 2 & -1 & & \\ & -1 & 2 & -1 & \\ & & \ddots & \ddots & \ddots \\ & & & -1 & 2 \end{bmatrix}\begin{bmatrix} v_1 \\ \vdots \\ \vdots \\ \vdots \\ v_{N-1} \end{bmatrix} = \begin{bmatrix} f_1 + g_0/(\Delta x)^2 \\ f_2 \\ \vdots \\ f_{N-2} \\ f_{N-1} + g_1/(\Delta x)^2 \end{bmatrix}.$$

We remark that

$$\begin{bmatrix} 2 & -1 & & & \\ -1 & 2 & -1 & & \\ & -1 & 2 & -1 & \\ & & \ddots & \ddots & \ddots \\ & & & -1 & 2 \end{bmatrix}$$

is a symmetric, positive-definite matrix of dimension $(N-1) \times (N-1)$.

There are two flavors of numerical methods to solve such a system $Ax = b$. One is to *directly* solve the system, i.e. invert the matrix. The number of flops (floating point operations per second) to perform this inversion ranges from $\mathcal{O}(N^3)$ (using e.g. Gaussian elimination, singular value decomposition) to $\mathcal{O}(N^2)$ (if we exploit the tridiagonal structure and use e.g. banded Cholesky). For such methods, the spatial complexity ranges from $\mathcal{O}(N^2)$ to $\mathcal{O}(N^{3/2})$ for our problem. The second method is to utilize *iterative* solvers. In this case, we generate a sequence of approximate solutions (x_n) such that $x_n \to x$ and x_{n+1} is cost-effective to compute after having x_n. For standard iterations schemes like Jacobi and Gauss-Seidel, we get flop counts of $\mathcal{O}(N^2)$ and spatial complexities of $\mathcal{O}(N)$ (SSOR reduces the former to $\mathcal{O}(N^{3/2})$. So, the iterative solvers are cheaper and at least as fast (and faster without special matrix structure), although they only give an approximate solution to the linear system. They are guaranteed to converge for any b and initial guess x_0 under certain conditions on the matrix, such as symmetric positive-definiteness. These are more commonly used for large N.

We will focus on the Jacobi method. For our problem, this generates the iteration scheme

$$v_j^{(n+1)} = \frac{1}{2}\left(v_{j+1}^{(n)} - v_{j-1}^{(n)} + (\Delta x)^2 f_j\right), \qquad 1 \le j \le N-1.$$

A variant of this, called the weighted or damped Jacobi method, compute the above as an intermediate value then weights it with $v_j^{(n)}$. Explicitly, the scheme is

$$\begin{aligned} v_j^* &= \frac{1}{2}\left(v_{j+1}^{(n)} - v_{j-1}^{(n)} + (\Delta x)^2 f_j\right), \qquad 1 \le j \le N-1 \\ v_j^{(n+1)} &= (1-\omega)v_j^{(n)} + \omega v_j^*, \qquad 1 \le j \le N-1, \end{aligned}$$

where $\omega \in \mathbb{R}$. It can be shown that $0 < \omega \le 1$ will give convergence (one can go a little higher, but it is approaching 1 asymptotically). A key feature of this method (for ω is the aforementioned range) is that it is very effective at damping high frequency (highly oscillatory) components of the

vector that it iterates on. This means that it smooths rough features out very well. However, it does not deal with smooth/low frequency components well (already smooth!). In fact, it can be shown that no ω will do this effectively. For this reason, we instead choose ω so that it deals with high frequencies optimally, and it is provable that $\omega = 2/3$ is optimal for our problem.

3 Multigrid

Recall that the damped Jacobi method is effective at dealing with high frequencies, but it does not fare well with low frequency modes. If we take our grid and coarsen it (say we remove every other grid point, being careful that we choose N such that we keep the boundary points), then a smooth wave will look sharper. That is, it now possesses more high frequency features. So, a reasonable idea is to

1. Do some Jacobi sweeps
2. Move what we get to a coarser grid (where our low frequencies become high frequencies)
3. Do some relaxation on the coarse grid (where we can effectively deal with high frequencies)
4. Come back up to the full grid via an interpolation process.

This isn't that far from what we actually do with multigrid. To move more in that direction, we make the following key observation:

If $Au = f$ and v is an approximation for u, then $Ae = r$, where $e = u - v$ is the error and $r = f - Av$ is the residual. We call $Ae = r$ the residual equation. The above tells us that relaxing on $Au = f$ with an arbitrary initial guess v is equivalent to relaxing $Ae = r$ with initial guess 0.

So, we modify our prior method as follows:

1. Do some Jacobi sweeps
2. Compute the residual
3. Move to the residual a coarser grid
4. Do some relaxation on the coarsened residual equation (this will give us an approximation to the error)
5. Bring our error back up to the full grid via an interpolation process
6. Correct our original Jacobi sweep with our refined error
7. Do some more relaxation on this quantity.

This is the building block of multigrid. However, once we go down to the coarse grid, why do stop there? We could continue this process recursively until we get to a grid where we can do a cheap direct solve. This is how we get our multigrid method, which can be done by following these steps:

1. We start with our problem on a given grid.
2. We perform a few iterations of weighted Jacobi (damping the high frequencies).
3. We compute the residual $f - Av$, where v is our approximate solution from step 2.
4. We move the residual to a coarser grid (where the problematic low frequencies now possess increased high frequency behavior and can be dealt with more effectively via weighted Jacobi). We can do this transition to the coarse grid either by restriction/projection or using averaging methods.
5. If the grid is coarse enough, we do a direct solve on the residual equation (which is cheap due to the size of the problem). If not, then we move back to step 2 on our current grid (now looking at the residual equation) and proceed recursively until the grid is coarse enough to do our direct solve. Then, we move onto step 6.

6. We move back to the refined grid (one step up) via interpolation between data points.
7. We correct what we got in step two with our computation in step 6.
8. We do a few more iterations of weighted Jacobi.

This is called a V-cycle. We can use them to generate our multigrid scheme by taking the returned vector from one V-cycle and making it the initial guess of the next V-cycle. The amazing thing about multigrid is that both the flop count and spatial complexity are $\mathcal{O}(n)$. This order is asymptotically optimal, since it indicates that we do $\mathcal{O}(1)$ operations to each component of the vector.

4 Results

The code in `1DPoissonMG.py`, along with containing the core multigrid solver using the function `V-cycle`, possesses various other functions with utility:

1. `solver_comp` computes the execution times for our multigrid method and NumPy direct solver, as well as the absolute 2-norm error for an array of grid sizes. Then, it prints this data in two tables (one table for execution times, one table for errors). One can use this function to verify the numerical accuracy of multigrid while featuring a much lower execution time than the Numpy direct solver (direct solver takes 49 seconds on a 16385 grid (N=14); I have not pushed it further, but it would likely take at least an order of magnitude longer).
2. `relax_test` computes the absolute error for an array of relaxation parameters, then them prints a table of these errors and outputs a plot of this data. From our prior discussion, we know that $\omega = 2/3$ is theoretically optimal, but this is not so noticable for small perturbations.
3. `mg_test` computes the execution time for multigrid for an array of grid sizes and prints this data in a table. It is similar to `solver_comp`, but it allows one to solve much larger problems (since we are not using the direct solver at the top level). This tells us that multigrid method can run on up to a 524289 (N=19) grid in under 1 minute.
4. `ana_comp` compares the multigrid solution to the analytical solution to -u''=sin(x) with arbitrary Dirichlet boundaries. One obtains an error of ~1e-6 after just 4 V-cycles when N=2^6

All of these functions are contained in the `Multigrid()` class and have precise documentation in `1DPoissonMG.py`. Here is some sample input and output:

Input to `solver_comp`:

```
drive   = Multigrid()                 # instantiate multigrid class
N_array = [2**n for n in range(6,14)] # array of grid sizes
drive.solver_comp(N_array)
```

Output of `solver_comp`:

```
+-----------+------------------------------+----------------------------------------+
| Grid Size | Multigrid Execution Time (s) | NumPy Direct Solver Execution Time (s) |
+-----------+------------------------------+----------------------------------------+
|        65 |         0.008763785001065116 |                0.00036002200067741796 |
|       129 |         0.012847697998950025 |                 0.0004634220022126101 |
|       257 |         0.023098724999726983 |                  0.001266939998458838 |
|       513 |          0.04389215799892554 |                  0.011966152000240982 |
|      1025 |          0.08147994899991318 |                   0.01582442700237152 |
|      2049 |          0.16037838199918042 |                   0.09368109899878618 |
|      4097 |           0.3648365989974991 |                    0.6735369830021227 |
```

```
|      8193 |           0.6315701820021786 |                      2.6383576019979955 |
+-----------+------------------------------+-----------------------------------------+
```

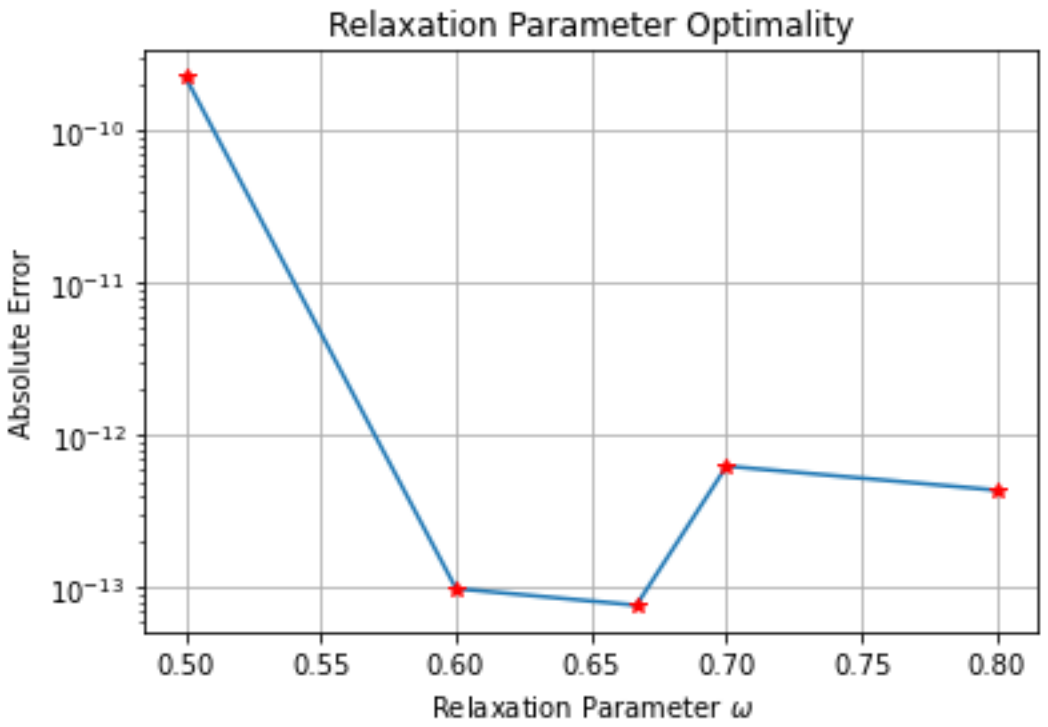
```
+-----------+--------------------------+
| Grid Size | Absolute Euclidean Error |
+-----------+--------------------------+
|        65 |    9.556254761269408e-14 |
|       129 |   3.6794935403782707e-13 |
|       257 |    1.840396983851491e-12 |
|       513 |    7.767679988043338e-12 |
|      1025 |    4.857682116518334e-11 |
|      2049 |   1.0490398063334259e-10 |
|      4097 |    3.743260606153312e-10 |
|      8193 |    5.097226123983849e-10 |
+-----------+--------------------------+
```

Input to `relax_test`:

```
drive       = Multigrid()               # instantiate multigrid class
omega_array = [0.5, 0.6, 2/3, 0.7, 0.8] # array of relaxation parameters
N           = 2**8                      # grid size
drive.relax_test(omega_array, N)
```

Output of `relax_test`:

```
+-----------+----------------------+-------------------------+
| Grid Size | Relaxation Parameter |                   Error |
+-----------+----------------------+-------------------------+
|       257 |                  0.5 | 2.2155681694916334e-10 |
|       257 |                  0.6 |  9.908908329069226e-14 |
|       257 |   0.6666666666666666 |  7.713323153041377e-14 |
|       257 |                  0.7 |  6.282443532386848e-13 |
|       257 |                  0.8 |  4.367908411594411e-13 |
+-----------+----------------------+-------------------------+
```



Input to `mg_test`:

```
drive   = Multigrid() # instantiate multigrid class
N_array = [2**n for n in range(6,16)]  # array of grid sizes
drive.mg_limit(N_array)
```

Output of `mg_test`:

```
+-----------+-------------------------------+
| Grid Size | Multigrid Execution Time (s) |
+-----------+-------------------------------+
|        65 |          0.009710047001135536 |
|       129 |          0.015764535000926116 |
|       257 |           0.02671550000013667 |
|       513 |           0.04875404700214858 |
|      1025 |           0.08642815699931816 |
|      2049 |           0.17044590799923753 |
|      4097 |           0.34982147100163274 |
|      8193 |            0.6816639559983741 |
|     16385 |            1.3509082869968552 |
|     32769 |            2.6054512939990673 |
+-----------+-------------------------------+
```

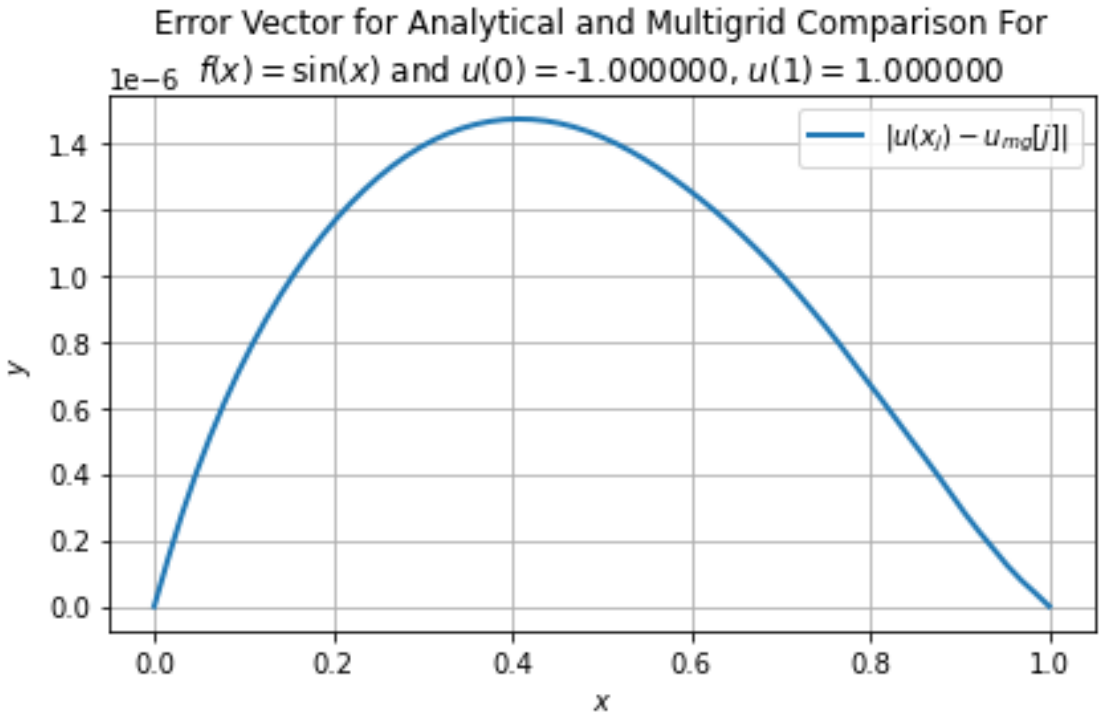
Input to `ana_comp`:

```
drive = Multigrid() # instantiate multigrid class
N     = 2**6        # grid size
g     = [-1,1]      # sample boundary data
```

```
drive.ana_comp(N, g)
```

Output of `ana_comp`:

```
Euclidean error is 8.431739824717852e-06
```



5 References

1. W. L. Briggs, V. E. Henson, S. F. McCormick, *A Multigrid Tutorial*, 2nd ed, SIAM, Philadelphia, 2000.
2. J. W. Demmel, *Applied Numerical Linear Algebra*, SIAM, Philadelphia, 1997.

[ ]: