Introduction to Multigrid Methods Computer Exercise #2

Goals. This is the second computer exercise. The goal is to solve the Poisson equation in 2D, using a geometric multigrid method. After this exercise is completed, you should be able to solve very large-scale problems in reasonable time.

Specific goals include:

1. Solving the Poisson and Helmholtz equations in 2D on the unit square
2. Implementing a V-cycle multigrid iteration recursively
3. Testing the code and verifying that it works correctly. Here it is sufficient to work with the Poisson equation, with a simple right-hand side, and simple Dirichlet conditions (homogeneous conditions are sufficient to start with, but you should try to also implement inhomogeneous Dirichlet conditions)
4. Encountering, and deciding on data structures. In 2D or higher, it becomes important to represent the solution in such a way that efficient computations are facilitated. Note that this is not a question of “complex geometry” but a question that must be considered for problems that are large enough to put extraordinary requirements on organization, in order to be solved efficiently. It is also a matter of visualization
5. Full-scale “production runs”. Here we are talking about running systems as large as possible, using up all available RAM memory on your computer, while making sure that you still have reasonable solution times

The equations. The Poisson equation and Helmholtz equation, respectively, are

\[-\Delta u = f\]
\[-\Delta u - \beta u = f.\]

The task is to create a solver that is able to solve both equations for every right-hand side \( f \), and with a Dirichlet boundary condition, \( u = 0 \) on the boundary of the computational domain. Note the sign of the differential
operator; the minus sign is chosen to agree with the common variational formulation when using finite elements.

The parameter in the Helmholtz equation, $\beta$, is positive, and you should note that you will not be able to solve this equation for arbitrary values of $\beta$. When $\beta$ becomes large enough to coincide with the first eigenvalue of $-\Delta$, the problem will become singular and cease to be elliptic with a unique solution. It is sufficient that you are able to demonstrate that you are able to solve the Helmholtz equation for some positive $\beta$. You are not required to solve problems that are no longer elliptic, although you should give it a try. If you work on the unit square, the first eigenvalue is $2\pi^2$, and the problem ceases to be elliptic when $\beta > 2\pi^2$. You can read an excellent introduction to the Helmholtz equation on Wikipedia.

**The methods.** Discretize the equations using a finite difference method (five-point operator) on an equidistant mesh (a uniform grid). Choose $\Delta x = \Delta y$.

The code should be able to handle any right-hand side $f$. Only use grids with a mesh width corresponding to negative powers of two, so that you can work with an arbitrary regress of grids by halving/doubling the mesh width.

Before you set out to build the MG code, write a “standard” solver for $-\Delta u = f$ using a block Toeplitz matrix $T$ and solving the system using MATLAB’s backslash operator, as well as the `sparse` command. The reason for doing this is that in the MG code, you will need to solve such a system on the coarsest grid. You can also run this code as a reference, e.g. timing the code using the commands `tic` and `toc`. Unlike in the 1D case, the standard solver will have to use a fairly large number of grid points, say a $31 \times 31$ grid. This means that the standard solver must solve on the order of $1,000$ equations. This is quite straightforward, though.

**Implementing a V-cycle iteration recursively.** Start your code design from the code templates you have developed in Exercise 1. Rethink the 1D code, and find out where you have to modify it to convert it into a 2D code. Otherwise, the code’s basic outline will be the same.

Use one damped Jacobi sweep as pre-smoothing and one as post-smoothing, but be prepared to modify the damping parameter $\omega$ (our analysis so far is only valid in the 1D case).

Note that while it would be straightforward to construct the restriction operator (if you use plain “injection”), it takes more work to construct the linear interpolation prolongation correctly. Because you have to do this work anyway, you should implement the restriction as the smoothing restriction,
which is a 2nd order lowpass filter of convolution (Toeplitz) type. This restriction is directly linked to the prolongation, as $P = 2^d R^T$, see lecture notes. Thus your implementation should use the so-called “Galerkin condition” in its construction. This guarantees that you do not lose ellipticity (positive definiteness) on the regress of grids.

**Data structures.** If you work in 2D it is convenient to represent the solution (which is a function of $x$ and $y$) as a matrix, with matrix elements $u_{i,j}$ representing an approximation to $u(x_i, y_j)$. When plotting the solution, make sure that your coordinate system is correctly oriented and follows the right-hand rule. (Note that in matrices, the first index runs ‘vertically’ and the second ‘horizontally’; in a function $u(x, y)$ the first variable runs ‘horizontally’ and the second ‘vertically’. Your matrix indices conveniently correspond to $x$ and $y$ coordinates if you draw your $x$-axis vertically downwards, and your $y$-axis horizontally, to the right.)

You may not use any matrix to represent the discrete differential operator (see below). This means that your code must work in such a way that it evaluates the residual, $Tu - f$, without using a representation of the matrix $T$ at all. Instead you have to work with the computational stencil, implemented in a way that supports working with convolutions, see the MATLAB command `conv2`. Read up on how to use this command, and make sure that you get the right dimensions when using it (check out the option `same`). Test this command on some small matrix, so that you find out how to insert boundary conditions as needed. It’s generally a good idea to try out all important steps or functions in your code on small sets of data so that you know exactly what happens, e.g. on the boundary or with the boundary conditions.

Note that there is a great difference between storing the solution $u$ as a matrix and representing the discrete differential operator $T$ as a matrix. If the grid has $N$ internal points in each independent variable, then (in 2D) you have $N^2$ unknowns, meaning that the “solution matrix” is a dense $N \times N$ matrix, while the discrete differential operator is an enormously large and sparse $N^2 \times N^2$ matrix. Work out why this is the case so that you realize why memory management is an issue when you solve elliptic equations in 2D or 3D.

It may also be a good idea to start only with a two-grid code (avoiding the recursive structure, calling the “standard solver” at the center of the MG iteration) and make sure that this works before you employ the recursive call to your multigrid code. If you try 3D, you will have to consider what would be the best data structure and representation of the solution. This is closely related to the numbering of the equations. However, you are not required to do this; it is enough to make the code work in 2D.
Testing the code and verifying that it works correctly. Test the code and verify that it works correctly in some simple case with only a few equations. Work in an incremental fashion, i.e., make sure that the code works one step at a time and one function at a time.

Compute the “exact solution” (using the backslash operation) e.g. when the number of equations does not exceed $31 \times 31$ equations in the 2D case, or $15 \times 15 \times 15$ equations if you try to work in 3D.

Note that matrices may only be used to represent the differential operator when you compute the solution at the coarsest grid level.

Production runs. Choose a right hand side $f$ and suitable boundary conditions so that you have an analytical solution (not a polynomial solution in $x$ and $y$). This can be done by choosing the solution, plugging it into the equation (whether Poisson or Helmholtz) and finding the corresponding $f$. Compute the error in the discrete $L^2$ norm for the final solution. (NOTE: what is the discrete $L^2$ norm in a 2D problem? Be careful not to make a mistake, as convergence will only have the right behavior in the correct norm.) Make sure that you can solve the Helmholtz equation for moderate values of $\beta$.

You should be able to run your code for at least 4M equations but preferably higher, such as 16M or 64M. The higher you are able to go, the better, but don’t stretch yourself too far in case your computer runs out of RAM memory. That is, try to utilize your available RAM as well as possible, but don’t go into particular schemes swapping data to the hard drive. It’s enough that you make sure that you don’t store any unnecessary vectors or matrices (you can’t afford storing copies!). Check how many nested grids are used in your code. Remember that the objective is to demonstrate your ability to work with and implement a multigrid iteration, not to break the record in the number of equations used to solve the Poisson/Helmholtz equation. Even so, the winner is the one who can solve the largest system...)

Visualization. Visualization is relatively straightforward if you work in 2D. (Already in 3D this presents a major problem, even if you solve the problem only on the unit cube.) Use the built-in primitives in MATLAB such as mesh or surf as the case may be. Plot the solution, making sure that the coordinate system complies with the right-hand rule. It’s a good idea to choose $f$ such that the solution is not entirely symmetric, meaning meaning that you should be able to see in the plot whether the graph has the correct orientation.

Note. The final hand-in project will build on the solver you develop here.