PHYS358: Midterm 2 (due Nov 13, 9:30am)

Solving Potential Problems: Multigrid Solvers

Note: Please read the complete set of instructions before you start.

Goal: The goal of this task is to develop an efficient method for solving elliptic partial differential equations (PDEs), or potential problems. These are abundant in electro-(or magneto-)statics (think electric fields in accelerators affecting the trajectories of particles), incompressible hydrodynamics (pressure field of air flow around wings), or fluid dynamics (self-gravity of an interstellar gas cloud, for example). As always with elliptic PDEs, the assumption is that we can treat the system as being in some sort of equilibrium such that there is no time dependence.

Discretizing Poisson's equation (here in two dimensions) leads to

$$\partial_x^2 u + \partial_y^2 u = -q \quad \rightarrow \quad \delta_x^2 u_{ij} + \delta_y^2 u_{ij} = \Delta x^2 q_{ij}, \tag{1}$$

under the assumption that we have a square domain with support points $x_i = i\Delta x$, $\Delta x = (x_{max} - x_{min})/J$, and similarly for y, i.e. $\Delta x = \Delta y$. Also, we'll use short-hand notation for the finite difference formulation of the second derivative:

$$\delta_x^2 u_{ij} \equiv u_{i+1,j} - 2u_{i,j} + u_{i-1,j}. \tag{2}$$

The question is how to solve eq. 1 efficiently. When you're finished this midterm, you should have understood:

1. how to use the Jacobi and Gauss-Seidel method to solve eq. 1.
2. the concept of *residuals* and *errors* in linear systems, and how to use them to improve the solution.
3. how to use those concepts to develop a "multigrid" method to solve Poisson's equation.

We will deal with the first item in Part (I), and the second couple of items will be discussed in Part (II).

Part I: Iterative Methods for Linear Systems

This is a review of topics discussed in PHYS331, albeit possibly cast in a different environment. We wish to solve the linear system 1 iteratively. Remember how this worked in the most general case: Given a linear operator $\boldsymbol{A}$ and a RHS vector $\boldsymbol{f}$, we wish to determine $\boldsymbol{u}$:

$$\boldsymbol{Au} = \boldsymbol{f} \tag{3}$$

The trick was to *assume* that for each row i of $\boldsymbol{A}$, we know every component $u_{j\neq i}$, so that we can bring it to the RHS:

$$a_{ii}u_i = f_i - \sum_{j\neq i} a_{ij}u_j. \tag{4}$$

Dividing by a_{ii} then yields a (new) guess for u_i. In matrix notation, this is just

$$\boldsymbol{u}^{n+1} = \boldsymbol{D}^{-1}\boldsymbol{f} - \boldsymbol{D}^{-1}(\boldsymbol{L}+\boldsymbol{U})\boldsymbol{u}^n, \tag{5}$$

where $\boldsymbol{A} \equiv \boldsymbol{L} + \boldsymbol{D} + \boldsymbol{U}$, i.e. the lower triangular, diagonal, and upper triangular of $\boldsymbol{A}$. Applying eq. 5 repeatedly yields the *Jacobi method*.

MT2(a) The Jacobi method [15pts]: Implement the Jacobi method in `pde_multigrid.py`. A few comments seem in order.

- I recommend using the function definition of `jacobi` in `pde_multigrid.py`. It may look somewhat cumbersome, but will make life easier when we'll talk about the multigrid solver.
- Standard textbooks (see e.g. Numerical Recipes) implement this problem using *face-centered* positions (see Zingale 2013, Sec 2.1 and 3.1). While conceptually simpler at first glance, multigrid solvers are more of a pain to implement with this, than with the alternative, namely *cell-centered* positions. For cell-centered grids, the grid points are located in the (as the name says) *centers* of cells. This means that e.g. the lower x-boundary is located at $x_0 - \Delta x/2$, and the upper boundary sits at $x_{J-1} + \Delta x/2$, where $\Delta x \equiv (x_{max} - x_{min})/J$. This has repercussions on how to set the boundaries. Rather than setting e.g. $u_{-1} = 0$, we have to make sure that the function value at x_{min} has the desired value $u(x_{min})$, i.e.

  $$u(x_{min}) = \frac{1}{2}(u_0 + u_{-1}), \tag{6}$$

  where u_0 is the *first active* cell, and u_{-1} is the *ghost cell*. Eq. 6 can be solved for u_{-1}, thus providing a prescription how to fill the ghost cells. Similarly, you can determine the value of the ghost cell u_J (remember that the last active cell is u_{J-1}).
- I strongly recommend to make use of numpy's array operations when implementing the Jacobi method, instead of coding a double for loop. One way to do this efficiently is to define a temporary array u_1 with the dimensions $(J+2, J+2)$, and filling the first and last indices with the boundary condition values (eq. 6). Then, finite differences can be written as (in 1D) `u1[2:J+2]-2*u1[1:J+1]+u1[0:J]`. Thus, there is no need to rewrite the 2D arrays in 1D form - many texts still recommend this.
- The problem should be defined on a grid of extent $(x_{min}, x_{max}) \times (y_{min}, y_{max}) = (-1, 1) \times (-1, 1)$, and the source vector should be given by $\boldsymbol{f} = -(\Delta x)^2$. Why should this be negative?
- For testing, just remember our discussion of the ADI method in class. You should see some parallels.

Test your implementation by running `pde_multigrid.py 64 1e-6 JC plate`. Check what these options mean.

1. Your code should return four plots: the RHS field $\boldsymbol{f}$, the solution of your integration, the analytic result (already implemented in `get_analytic`), and the residual (i.e. the difference between the integration result and the analytic solution). Fig. 1 gives an example how this could look like. [6 pts]
2. Take note of the number of iterations to achieve the desired accuracy. [3 pts]

3. Explain the minus sign in front of the RHS vector $\boldsymbol{f}$, comparing to the solution of the heat equation. [3 pts]

4. Briefly explain for what matrices $\boldsymbol{A}$ eq. 5 will converge (you may have to dig up your notes on PHYS331). [3 pts]

Solution: See `pde_multigrid.py`. The code should take ~ 4300 iterations. The minus sign comes from the fact that $\boldsymbol{f}$ is the source term in the heat equation. Setting the time derivative to zero, i.e. assuming a steady state and bringing $\boldsymbol{f}$ to the other side...

MT2(b) The Gauss-Seidel method [15pts]: As you will remember, the Gauss-Seidel method is an improvement on the Jacobi method. Here, we use the already updated values as soon as they become available,

$$a_{ii}u_i^{n+1} = f_i - \sum_{j<i} a_{ij}u_j^{n+1} - \sum_{j>i} a_{ij}u_j^n. \tag{7}$$

In matrix notation, this is

$$\boldsymbol{u}^{n+1} = (\boldsymbol{D}+\boldsymbol{L})^{-1}\boldsymbol{f} - (\boldsymbol{D}+\boldsymbol{L})^{-1}\boldsymbol{U}\boldsymbol{u}^n. \tag{8}$$

1. Implement the Gauss-Seidel method in `pde_multigrid.py` (again, I recommend using the function definition `gauss_seidel` as provided). [6 pts]

2. Your code should produce the same four plots as discussed above.

3. Test your code with `pde_multigrid.py 64 1e-6 GS plate`. [3 pts]

4. Take note of the number of iterations needed. [3 pts]

5. Comparing eqs. 5 and 8, why does the Gauss-Seidel method converge faster than the Jacobi method? [3 pts]

Solution: The code should take ~ 2500 iterations. Eigenvalues of $\boldsymbol{D}+\boldsymbol{L}$ are generally larger than those of just $\boldsymbol{D}$, therefore, the iteration matrix leads to faster convergence.

MT2(c) The Multigrid solver: Preliminaries [10pts]: For the following, I recommend reading chapters 2 and 3 of the multigrid tutorial by Briggs *et al.* before proceeding. To summarize: The construction of the Jacobi and Gauss-Seidel methods should remind you of solving the heat equation $\partial_t u = (\partial_x^2 + \partial_y^2)u + q$ as discussed in homework 07. The only thing that changes is that we replace the time-update by an iteration. If we let $t \to \infty$ in the heat equation, we'll recover the solution to Poisson's equation $(\partial_x^2 + p_y^2)u = -q$. The key point for our next step is to remember how differently the diffusion operator affects perturbations on large scales and on small scales.

1. Discuss briefly why small-scale perturbations ("errors") are decaying faster than large-scale perturbations. Here "small-scale" means perturbations at large wavenumbers (or at short wavelengths). [4 pts]

2. The first key element in multigrid methods is to solve the underlying equations on a range of coarser grids (usually reducing the number of support points by a factor of 2 each time). Why can this approach accelerate the convergence to the correct solution of eq. 1? *Hint: See chapter 3 of Briggs et al, specifically page 32.* [3 pts]

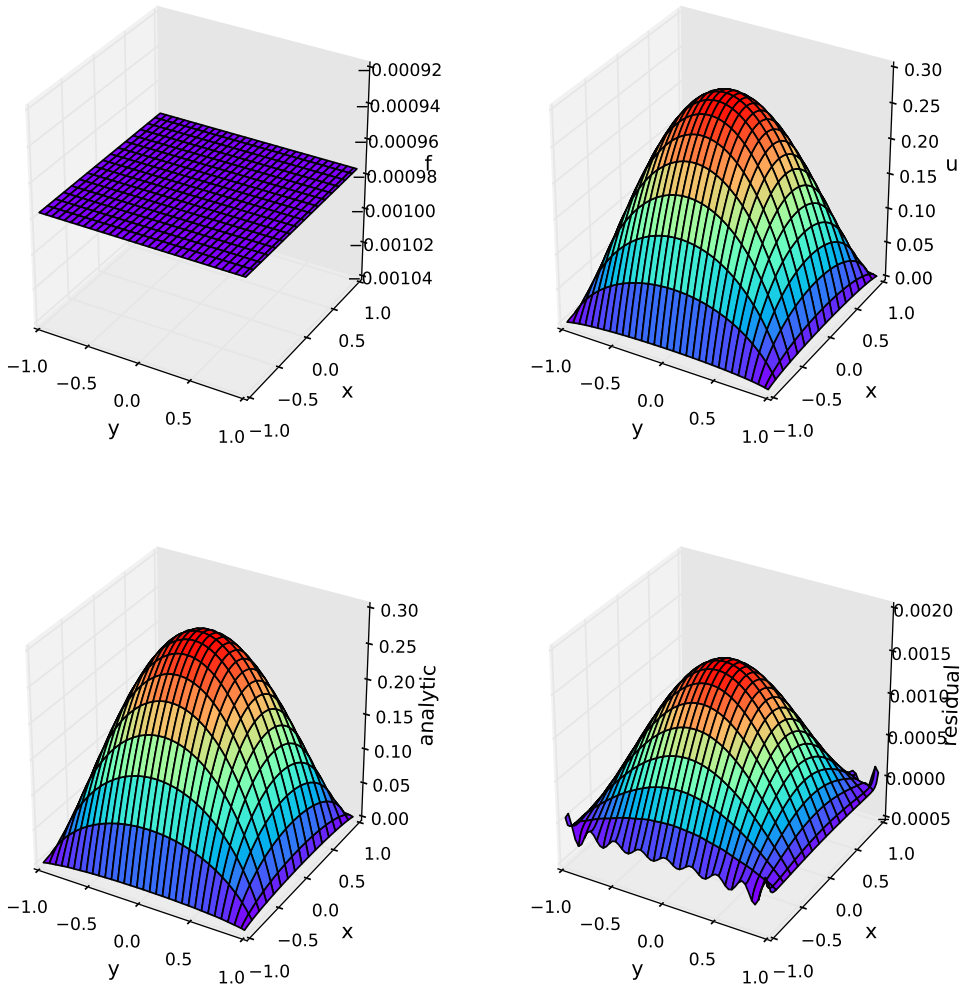

Figure 1: The "heated plate" problem solved with the Jacobi method.

3. The second key element is to realize that solving the linear system $\boldsymbol{Au} = \boldsymbol{f}$ is equivalent to solving the associated residual system $\boldsymbol{Ae} = \boldsymbol{r}$. Follow the steps in Briggs et al., exercise 2 (page 43) to prove that equivalence. [3 pts]

Solution: The diffusion operator is (mathematically) the curvature of the underlying function. Smaller scale perturbations have larger curvature, thus, they decay faster. Physically, steeper gradients lead to higher heat transfer rates (Fourier's law of heat transport). Large-scale errors on fine grids turn to small-scale errors on coarser grids - successively solving the equations on coarser grids will remove the (formerly) large-scale errors. $\boldsymbol{Au} = \boldsymbol{f}$ is equivalent to $\boldsymbol{Ae}_0 = \boldsymbol{f} - \boldsymbol{Av}_0$, with $\boldsymbol{e}_0 \equiv \boldsymbol{u} - \boldsymbol{v}_0$, since the RHS should approach $\boldsymbol{0}$.

MT2(d) The Multigrid solver: Implementation of a recursive V-cycle [20pts]: The idea of the V-cycle (for 2 levels) is this:

a Use an iterative method (e.g. Gauss-Seidel) to get an approximate solution $\boldsymbol{v}_h$ for the system $\boldsymbol{A}_h\boldsymbol{u}_h = \boldsymbol{f}_h$. The subscript h indicates that the vectors have support point spacings of h, and are acted upon by a correspondingly discretized linear operator $\boldsymbol{A}_h$.

b Calculate the residual $\boldsymbol{r}_h = \boldsymbol{f}_h - \boldsymbol{A}_h\boldsymbol{v}_h$ and *restrict* it to the coarser grid (with spacings $2h$), i.e. $\boldsymbol{r}_{2h} = \mathcal{R}_h^{2h}(\boldsymbol{r}_h)$. The *restriction operator* $\mathcal{R}_h^{2h}$ can be as simple as an averaging procedure (more below).

c Because of the equivalence between $\boldsymbol{Ae} = \boldsymbol{r}$ and $\boldsymbol{Au} = \boldsymbol{f}$, we determine $\boldsymbol{e}_{2h}$ on the coarse grid and *prolong* it to the fine grid, i.e. $\boldsymbol{e}_h = \mathcal{P}_{2h}^h(\boldsymbol{e}_{2h})$. The *prolongation operator* $\mathcal{P}_{2h}^h$ is just a prescription to interpolate the data from the coarse grid to the fine grid (see Zingale 2013, Fig. 4 and associated explanations).

d Now that we have the error on the fine grid, we can correct our approximate solution $\boldsymbol{v}_h \leftarrow \boldsymbol{v}_h + \boldsymbol{e}_h$.

Step [c] above involves calculating $\boldsymbol{e}_{2h}$ from $\boldsymbol{A}_{2h}\boldsymbol{e}_{2h} = \boldsymbol{r}_{2h}$. The trick is now that instead of trying to solve for $\boldsymbol{e}_{2h}$ right then and now, we calculate *its* residual, restrict it to the next coarser grid ($4h$), and start with step [a] again. Thus, we will get a *recursive* prescription to calculate $\boldsymbol{v}_h$. A few points of interest:

- Since we keep dividing the number of support points J by 2, it is convenient to set the number of support points J_h at the finest grid to a power of two.
- Then, after $\log_2(J)$ levels, we will have $J_{Jh} = 1$, i.e. our operator and vectors are scalars (the exact form will depend on the boundary conditions). Thus, we can solve exactly for $\boldsymbol{e}_{Jh}$.
- Cell-centered quantities: This choice seems unnecessarily complicated at first, but it allows for an easy implementation of the restriction and prolongation operators, rendering them *conservative*.
- Restriction: For cell-centered quantities (see Zingale 2013, Fig. 4), this just means averaging (in two dimensions) the four cells on the finer grid to their counterpart on the coarser grid (Zingale 2013, eq. 28).
- Prolongation: This will require the slopes between support points on the coarser grid (Zingale 2013, eq. 29-35).

- A single V-cycle call will only give an approximate resolution. To converge to the "true" solution, typically 10–20 repeated V-cycles are necessary. Each repeated call (except the first) uses the result of the previous call as initial guess, thus improving on the solution.

To implement the full V-cycle, I recommend the following steps.

1. Identify the function names in `pde_multigrid.py` with the steps for the V-cycle described above. [2 pts]
2. Implement the V-cycle in `mg_vcycle`, using the provided function definitions. Use `npre=10, npst=10` for the pre- and post-smoothing Gauss-Seidel iteration numbers. Make sure that your `gauss_seidel` accepts an argument `maxit` limiting the number of iterations. [10 pts]
3. Implement the driver for V-cycle in `multigrid`. Make sure all calls to `mg_vcycle` after the first use the result of the previous iteration as an initial guess (otherwise, you won't see any improvement). [8 pts]
4. Functions for polongation, restriction and "no-charge" boundary conditions are already provided.
5. You can use `get_analytic` to compare your results to the analytic solution. That function also provides an example how to use `get_mesh`.

MT2(e) The Multigrid solver: Testing [10pts]: To test your implementation, use the following steps:

1. Run `pde_multigrid.py 64 1e-6 MG plate`. Your code should produce the four plots as for the Gauss-Seidel and Jacobi method. With 20 calls to `mg_vcycle`, the plots should look like shown in Fig. 2. [3 pts]
2. Run `pde_multigrid.py 64 1e-6 GS plate` again. Compare the magnitude of the residual to that achieved with the multigrid method. Also, compare the time used between each method (times should be comparable). Note that the Gauss-Seidel method should stop when the desired *tolerance* is reached (here, 10^{-6}). [3 pts]
3. Now, run `pde_multigrid.py 64 1e-6 GS circle`. Check the initialization file to understand what the problem setup is. How many iterations do you need? (This will take a while...). [2 pts]
4. Repeat, but now with multigrid, i.e. `pde_multigrid.py 64 1e-6 MG circle`. Does the multigrid method take longer or shorter? [2 pts]

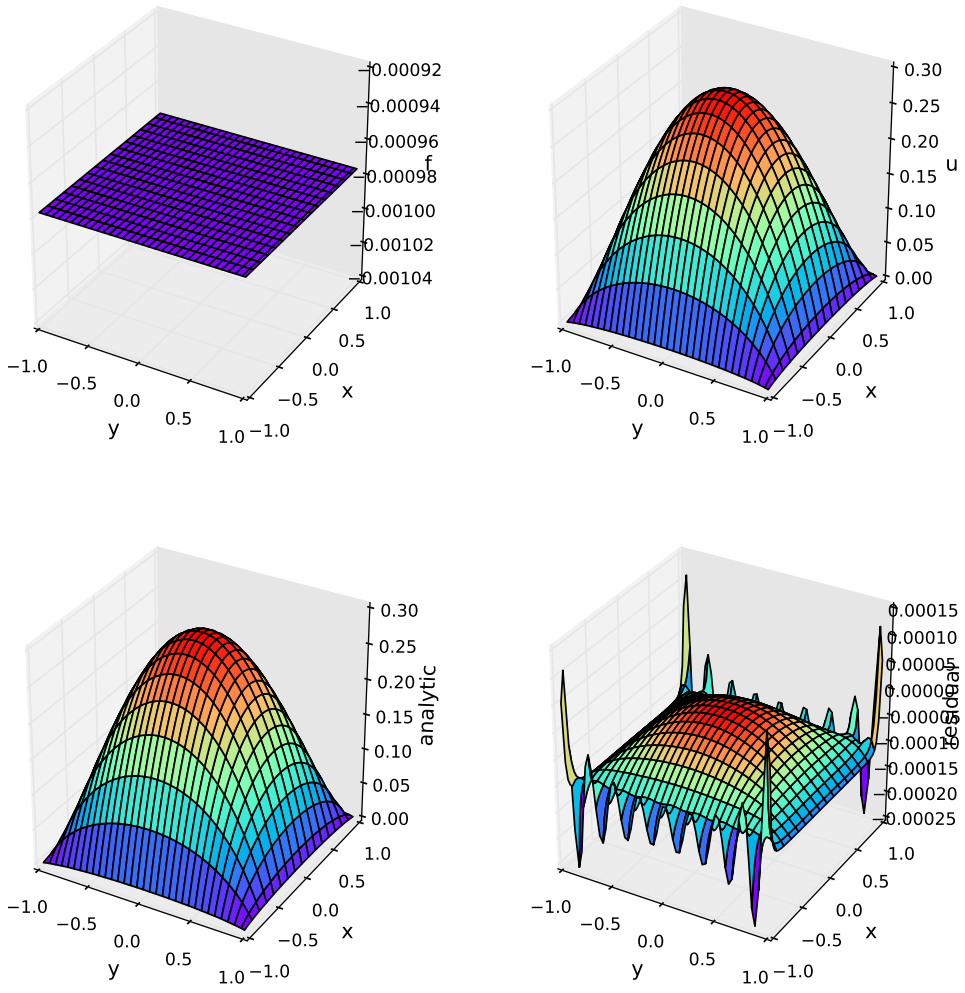

Figure 2: The "heated plate" problem using the V-cycle.