Write an iterative real-space Poisson solver in Python/C

Ask Hjorth Larsen
asklarsen@gmail.com

October 10, 2018

The Poisson equation is

$$\nabla^2 \phi(\mathbf{r}) = \rho(\mathbf{r}). \tag{1}$$

This is a second-order linear differential equation which in physics relates an electrostatic potential $\phi(\mathbf{r})$ to a continuous charge distribution $\rho(\mathbf{r})$. The goal of this tutorial is to write a Python code which calls a C extension to solve the Poisson equation in two dimensions on a uniform real-space grid. To keep life simple, we will do this for zero boundary conditions[1], and only for rectangular domains with equal spacing in the x and y directions.

Grids and Laplacian

Define first a simple 1D grid (e.g. using `np.linspace`) and a function whose second derivative you know analytically. How can you calculate the second derivative using finite differences?

Next we represent a function $f(x, y)$ as a two-dimensional array $f_{ij} \equiv f(x_i, y_j)$, where $x_i = x_0 + ih$ and $y_j = y_0 + jh$ define a uniform rectangular grid with spacing h, and where $i = 0, \ldots, I$ and $j = 0, \ldots, J$ are integer indices.

Choose a 2D grid and a suitable charge distribution $\rho(x, y)$ whose potential you might want to calculate later on, then plot it with `imshow` from matplotlib.

It may be helpful to use numpy functions like

- `x = np.linspace(start, stop, npoints)`

[1] It is common to solve with zero boundary conditions and later correct the boundary conditions by pre/postprocessing. This is simple because the equation is linear.

- `X, Y = np.meshgrid(x, y)`
- Now `X` and `Y` are useful for defining 2D functions like `X**2 + Y**2`.

Interactive interpreters and Jupyter notebooks are good for playing around and trying things, but if some day we would like to make our solver available as a Python module, we should write it as a Python module.

Put the code into a Python file and make sure things work as a standalone code.

The Laplacian ∇^2 can be discretized onto our grid by locally approximating the second-order derivatives from the neighbouring grid points:

$$\begin{aligned}\nabla^2\phi(x,y) &= \left(\frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2}\right)\phi(x,y) \\ &\approx \frac{1}{h^2}\left[\phi_{i+1,j} + \phi_{i,j+1} + \phi_{i-1,j} + \phi_{i,j-1} - 4\phi_{ij}\right]. \end{aligned} \tag{2}$$

This is called the *5-point stencil*. Each value is evaluated from the 5 closest grid points. It approximates the Laplacian within an error of $\mathcal{O}(h^2)$, so we say it is a second-order stencil.[2]

I. Implement a function which calculates the Laplacian of a function f_{ij} on the grid.

Clearly the stencil only works on the *interiour* of the grid. Recall that we earlier endeavoured to fix the boundary conditions to zero.

Poisson solver

In our grid representation the Poisson equation reads

$$\phi_{i+1,j} + \phi_{i,j+1} + \phi_{i-1,j} + \phi_{i,j-1} - 4\phi_{ij} = \rho_{ij}h^2. \tag{3}$$

Isolating ϕ_{ij}, we define an iterative map in which the potential ϕ_{ij}^{k+1} at the $(k+1)$th step is calculated from that of the kth step according to

$$\phi_{ij}^{k+1} \leftarrow \frac{1}{4}(\phi_{i-1,j}^{k} + \phi_{i+1,j}^{k} + \phi_{i,j+1}^{k} + \phi_{i,j-1}^{k} - \rho_{ij}h^2). \tag{4}$$

Lo and behold, as we iterate this enough times, ϕ_{ij}^{k} approaches the solution no matter which starting potential we chose.

Implement a function which iterates (4), then solve the Poisson equation as described above for some chosen density $\rho(x,y)$.

[2] By using more neighbours, it is possible to formulate more accurate stencils with order $\mathcal{O}(h^{2n})$ where n is the number of nearest neighbours used.

If the Poisson solver works, it should be easy to verify that the residual $R[\phi^k] = \nabla^2\phi^k(x,y) - \rho(x,y)$ approaches zero. Also, the norm of the residual is a useful measure for whether the calculation has converged. Be sure to make some plots while considering this.

Package the code into a function `solve(rho, phi, h, epsilon)` so the user can provide different arrays, grid spacing h, and a tolerance ϵ which controls how well the calculation is converged.

This can be done using basic Python loops or using numpy. How much faster is the numpy version?

Gauss–Seidel solver

If we loop over i and j in increasing order, we will already have passed $(i-1, j)$ and $(i, j-1)$ by the time we reach (i, j). Hence we already have the new and improved $\phi^{k+1}_{i-1,j}$ and $\phi^{k+1}_{i,j-1}$ of which we can take immediate advantage:

$$\phi^{k+1}_{ij} \leftarrow \frac{1}{4}(\phi^{k+1}_{i-1,j} + \phi^{k}_{i+1,j} + \phi^{k}_{i,j+1} + \phi^{k+1}_{i,j-1} - \rho_{ij}h^2). \tag{5}$$

This iterative scheme is called the Gauss–Seidel method. The former is called the Jacobi method.

How many iterations does it take to solve the Poisson equation for a given tolerance with Gauss–Seidel versus Jacobi?

The Gauss–Seidel method will require fewer iterations. We easily wrote the Jacobi update using Numpy arrays, but it is not straightforward to write the Gauss–Seidel update likewise.

Sometimes there is a function that can solve the problem for us, but not always. Indeed there will always be some types of functions that cannot be done as efficiently as in C:

- Complex element-wise mathematical operations on the same array, like `np.sqrt(1 + A**2 * (1 + A))` lead to several loops over the same chunk of memory, and would be faster to do in C using a single loop.
- Any operations where the each element successively depend on each other in a tight loop.

In our case, both these things are a problem to some extent. For high-performance computing it is normally fine to write most code in Python, but a few parts will be very performance critical and hence nice to have in a low-level language like C. We now have an excuse to write the Gauss–Seidel update scheme in C.

Calling C from Python

There are many ways to call C from Python. The simplest is probably to use the module `ctypes`. First we need to know how to write a function in C. Here is some inspiration:

```
#include <stdio.h>

int multiply_vector(double a, double *v, int len)
{
    int i;
    for(i=0; i < len; i++) {
        v[i] *= a;
    }
}

int multiply(double a, double b)
{
    double c = a * b;
    printf("multiply %f %f --> %f\n", a, b, c);
    return c;
}

int main(int argc, char **argv)
{
    double x = multiply(2.0, 3.0);
    printf("got %f\n", x);

    double v[5] = {1.0, 2.0, 0., -1., 42.};
    multiply_vector(0.5, v, 5);
    int i;
    for(i=0; i < 5; i++) {
        printf("%f\n", v[i]);
    }
    return 0;
}
```

This file, which we shall call `example.c`, can be compiled by `gcc example.c`. This will produce a binary called `a.out` which will run and print some things.

Write a "hello world" program in C, then compile and run it.

Add a function which computes something simple (e.g., $a+b$) and verify that it works.

In C, arrays are represented as a shape (here two integers I, J) plus the data (here a `double*` pointer to the first element, with the $n \times m$ elements following in immediate succession).

Pass a numpy array to C. In Numpy you get elements like `array[i, j]`. But in C you just have a pointer to a chunk of data (and the shape). How do you get element i, j then?

Once we have figured out how the 2D index i, j is mapped to the single index $0 \leq n \leq IJ$, it will be possible to loop over arrays and write the solver.

Write a Gauss–Seidel solver in C.