Computational Physics – Lecture 10

Andrei Alexandru

April 22, 2019

1 Introduction

This week we will discuss how to solve Laplace's equation, a partial differential equation frequently encountered in physical problems, using the Gauss-Seidel algorithm. The problem we will tackle during the lecture is to find the potential in a two dimensional region bounded by two conductors held at different electric potentials. In general, for static potentials we can write the electric field in terms of an electric potential:

$$\boldsymbol{E} = -\nabla\phi\,. \tag{1}$$

Gauss's law, in differential form $\nabla \cdot E = \frac{1}{\epsilon_0}\rho$, then connects the electric potential to the charge distribution ρ via Poisson's equation

$$\nabla^2\phi = -\frac{1}{\epsilon_0}\rho\,. \tag{2}$$

For regions where the charge distribution is zero, like the case in our problem, Poisson's equation reduces to the Laplace equation

$$\nabla^2\phi = \frac{\partial^2\phi}{\partial x^2} + \frac{\partial^2\phi}{\partial y^2} + \frac{\partial^2\phi}{\partial z^2} = 0\,. \tag{3}$$

Solutions of the Laplace equations are called *harmonic functions* and they have a number of interesting properties. In particular, a harmonic functions has the property that the if we take a sphere centered at x, $S(x,r) = \{y|\,\|x-y\| = r\}$, the average value of the function on the surface of the sphere is equal to the value at the center:

$$\frac{1}{4\pi r^2}\int_{S_{x,r}} d^2y\,\phi(y) = \phi(x)\,. \tag{4}$$

This is the property that inspires the Jacobi relaxation method we will use to solve numerically Laplace's equation: we will iteratively replace the value of the field with the mean of its neighbors until convergence. The solution of this iteration will have the harmonic property discussed above.

Before we move on to discuss the Jacobi method in more detail, I would like to point out that Laplace's equation alone does not uniquely determine the field ϕ. We need to supplement this with *boundary conditions*. To be more specific, assume we try to find out a solution of Laplace's equation in the region Ω. The solution is then unique when we specify the behavior of the function on the boundary of the region, often denoted with $\delta\Omega$. Two types of boundary conditions are commonly used: Dirichlet conditions when the value of the field on $\delta\Omega$ is known, and Neumann conditions where $\boldsymbol{n}\cdot\boldsymbol{\nabla}\phi$, the normal component of the gradient on the boundary, is known on $\delta\Omega$. In this lecture we will use Dirichlet boundary conditions.

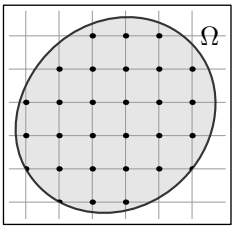
2 Jacobi relaxation method and Gauss-Seidel algorithm

Consider the Laplace equation in two dimensions:

$$\frac{\partial^2\phi}{\partial x^2} + \frac{\partial^2\phi}{\partial y^2} = 0\,. \tag{5}$$

We assume that we want to find the solution to this equation in a domain Ω, given the values of the field ϕ on the boundary of the domain.

We will solve this by discretizing the equation: we sample the field ϕ on a regular grid $(x, y) \in \Gamma = \{n_x a\hat{\boldsymbol{x}} + n_y a\hat{\boldsymbol{y}} | n_x, n_y \in \mathbb{Z}\} \subset \Omega$. In the figure on the right we sketch the process: the domain Ω is the grayed area and the sampled points Γ is indicated with the black dots. The equation is expressed in terms of discretized derivatives:



$$\begin{aligned} \frac{\partial^2 \phi(x,y)}{\partial x^2} &\approx \frac{\phi(x+a,y) + \phi(x-a,y) - 2\phi(x,y)}{a^2}\,, \\ \frac{\partial^2 \phi(x,y)}{\partial y^2} &\approx \frac{\phi(x,y+a) + \phi(x,y-a) - 2\phi(x,y)}{a^2}\,. \end{aligned} \tag{6}$$

The discrete derivative above involves only values of the field ϕ at the sampled points in the set Γ. The approximation can be easily verified by performing a Taylor expansion around (x, y) to the second order. You can also check this in Mathematica easily using `Series`; for example `Series[(ϕ[x+a]+ϕ[x-a]-2ϕ[x])/a^2,{a,0,3}]` will check the second derivative expression and also tell you what the leading order correction for the approximation is. The correction term will control how fast the discretized result converges to the exact one when making the gird finer and finer (the $a \to 0$ limit.)

The strategy is to solve the discretized version of the equation, evaluate the quantity of interest in terms of the discretized solution, and then take the limit where the grid step a goes to zero. In practice we perform the calculation on a set of finer and finer grids until the solution converges to the accuracy required.

To solve Laplace's equations in this approximation we have to find a set of ϕ values that satisfy

$$\frac{1}{a^2}\left[\phi(x+a,y) + \phi(x-a,y) + \phi(x,y+a) + \phi(x,y-a) - 4\phi(x,y)\right] = 0\,. \tag{7}$$

This is equivalent with requiring that $\phi(x, y)$ is equal with the average of the nearest neighbors $\phi(x \pm a, y)$ and $\phi(x, y \pm a)$, a property similar to the sphere average for the harmonic functions. To find the field ϕ that satisfies this condition we start with an approximation of the solution, $\phi^{(0)}(x, y)$, and then improve the solution using the *Jacobi iteration*:

$$\text{For each } (x,y) \in \Gamma\text{:} \quad \phi^{(i+1)}(x,y) = \frac{1}{4}\left[\phi^{(i)}(x+a,y) + \phi^{(i)}(x-a,y) + \phi^{(i)}(x,y+a) + \phi^{(i)}(x,y-a)\right]\,. \tag{8}$$

Obviously, the solution of Eq. 7 is a fixed point of this iteration. As it turns out, if we start from any guess $\phi^{(0)}(x, y)$ we are guaranteed to converge to the solution. Of course, if we start with a good approximation the iteration will converge faster. For our implementation we will actually start with a very rough guess: $\phi^{(0)}(x, y) = 0$.

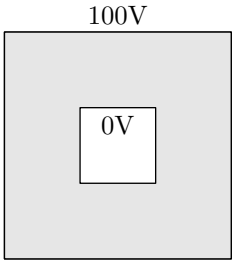
In the iteration above, all values of the field for the new iteration $\phi^{(i+1)}(x, y)$ are computed in terms of the old values. However, if we sweep the set Γ in a sequential manner, it is often the case that some of the neighbors of (x, y) have been visited by the iteration already and the values of field were updated and replaced with better estimates. It makes sense then to use these new values in our iteration, rather than the older ones. For example, if we cycle through Γ in increasing x and y order, the iteration above is replaced with:

$$\text{For each } (x,y) \in \Gamma\text{:} \quad \phi^{(i+1)}(x,y) = \frac{1}{4}\left[\phi^{(i)}(x+a,y) + \phi^{(i+1)}(x-a,y) + \phi^{(i)}(x,y+a) + \phi^{(i+1)}(x,y-a)\right]\,, \tag{9}$$

since the values $\phi^{(i+1)}(x - a, y)$ and $\phi^{(i+1)}(x, y - a)$ are already available. This improvement to the Jacobi iterations is called the *Gauss-Seidel method*. Note that in our implementation, this is done by performing an *in-place* update of the field values ϕ. Jacobi iteration usually requires a separate array to hold the values of $\phi^{(i)}$ while computing $\phi^{(i+1)}$, so Gauss-Seidel method is more efficient both in terms of convergence speed and memory use.

A note about the boundaries: the ϕ values involved in the iteration above are mostly in at positions included in the set Γ. However, for points in Γ located next to the boundary $\partial\Omega$, some of the values of field will be sampled from outside Ω. Those values are controlled by the boundary conditions: for our implementation we will assume that they have the value of the field at the corresponding boundary point. The values of these points will not be modified by the iteration.

3 Implementation details



We will now describe in detail the implementation of the algorithm for an electrostatics problem. Assume we have two conductors shaped as squares: one 1 cm × 1 cm and the other 3 cm × 3 cm. They are arranged as in the figure on the right. The outer conductor has a potential of 100V and the inner one 0V. Our task is to find the value of the field in the area between the conductors, the grey area in the figure. Using the value of the field we should be able to find out the electric field and using Gauss's theorem to determine the total charge on the inner conductor.

Our implementation will be split in three sections:

- the main routine that handles the input, output, and post-calculations like the gradient and the flux,
- the gauss-seidel routine that computes the solution to the Laplace equation,
- the setup routine that prepares the data structures required for the gauss-seidel routine based on the geometry of our problem.

To keep the implementation general, we note that the iteration in Eq. 9 updates the value of $\phi(x, y)$ based on the values of the field at the nearest neighbor points on the grid. The important point is that this routine does not require knowledge of the *geometry* of the problem, that is the coordinates of the grid points, but only *topology*, that is only relative placement of the points. We will hold all the values of the field ϕ at the interior points in an array `double phi[]`. The length of the array is `int nphi`, indicating the number of interior points in the grid. Each values in this array corresponds to one particular point on the grid, but we make no assumption on how these point map to the grid points. To indicate the topology, we fill an array of integers `int ne[]`, that will hold 4 offsets for each interior points, with `ne[4*i+0]` holding the offset of the neighbor in the $+x$ direction for point `i`, `ne[4*i+1]` in the $-x$ direction, `ne[4*i+2]` the $+y$ neighbor, and `ne[4*i+3]` the $-y$ one. Thus if `i` is the offset for $\phi(x, y)$ (that is `phi[i]`$=\phi(x, y)$), then `phi[ne[4*i+0]]` will hold the value of $\phi(x + a, y)$.

For most interior points the neighbors are also interior points, so they will have an offset in the `phi` array. For the points close to the boundary some of the offsets will indicate boundary points, points that are not logically part of the `phi` array. However, to make the logic of the routine simple, the boundary points will be indicated by negative offsets and we will hold the boundary values in the same `phi` array. Note that we only update in the `gauss-seidel` routine the positive offset points in the `phi` array, the ones corresponding to the interior points. The implementation for this routine is given below.

```
1  int gauss_seidel(double* phi, int* ne, int nphi, double del)
2  {
3    int iter = 0;
4    while(1)
5    {
6      double err=0, nrm=0;
7      for(int i=0; i<nphi; ++i)
8      {
9        double res = 0;
10       for(int n=0; n<4; ++n) res += phi[ne[4*i+n]];
11       res /= 4;
12       err += (res-phi[i])*(res-phi[i]);
13       nrm += res*res;
14       phi[i] = res;
15     }
16     iter++;
17     if(sqrt(err/nrm) < del) break;
18   }
19   return iter;
20 }
```

Note that the iteration over the `nphi` values of the array implements the logic in Eq. 9. As we iterate, the field approaches the solution of the Laplace equation. To decide when to stop, we compute the relative difference between new estimate and previous one:

$$\texttt{rel} = \sqrt{\frac{\texttt{err}}{\texttt{nrm}}} = \sqrt{\frac{\sum_{(x,y)\in\Gamma}(\phi^{(i+1)}(x,y) - \phi^{(i)}(x,y))^2}{\sum_{(x,y)\in\Gamma}(\phi^{(i+1)}(x,y))^2}}\,. \tag{10}$$

When the relative error is smaller than the required precision `del` we stop iterating and return. Note that since we used the array `phi` to also hold the boundary values, the logic of the Gauss-Seidel iteration is implemented in a straightforward way in just few lines of code.

The particular geometry of the problem will dictate how the `phi` array maps onto the grid and determines the list of neighbors `ne`. This is implemented in the `setup` subroutine. The general strategy is to first overlay a grid that covers the interior of the outer square and then remove the points that lay inside the smaller square. In the first step the index is easy to compute: the grid is $N \times N$ and the point at position (i, j) on the grid has index $i \times N + j$. This allows us to also easily compute the index of the neighbors. For this first pass, we will setup an $N \times N$ array of `struct point`s that holds all the relevant geometrical and topological structure. The code implementing the first sweep is listed below (note that this is just a fragment of the `setup` routine.)

```
1  typedef struct {
2    int index;
3    int ne[4];
4    double x,y;
5  } point;
6
7  void setup(int N, int* ne, int *nphi, int *nb, double *pos)
8  {
9    double dx = 3.0/(N-1);
10
11   point *pts = malloc(N*N*sizeof(point));
12
13   for(int i=0; i<N; ++i)
14   for(int j=0; j<N; ++j)
15   {
16     pts[i*N+j].index = i*N+j;
17     pts[i*N+j].ne[0] = (i+1)*N+j;
18     pts[i*N+j].ne[1] = (i-1)*N+j;
19     pts[i*N+j].ne[2] = i*N+j+1;
20     pts[i*N+j].ne[3] = i*N+j-1;
21     pts[i*N+j].x = i*dx-1.5;
22     pts[i*N+j].y = j*dx-1.5;
23   }
```

The `dx` spacing is computed such that the first and last points in the grid lay on the boundaries of the outside box $x = \pm 1.5$, $y = \pm 1.5$, since we take the boxes to be centered at $(x, y) = (0, 0)$. The array `pts` will be used to compute the neighbor structure `ne`, the number of interior points `nphi`, and the position of these points to be stored in the array `pos`. The routine will also set the number of inequivalent boundary points, which in this case is `*nb=2` since the field assumes only two distinct values on the boundary.

Note that the above code sets the `ne` values correctly only for interior points that are away from the boundaries. We will fix their index in the subsequent sweep through the lattice. The next sweep removes the points that are not interior points. We will index all points that are on the outer border of the box with -1 since all of them will be set to the same potential.[1] The points inside of the inner square (on the inner

[1] If the potential varies along the boundary, then each boundary point would require a separate negative index and their values have to be filled accordingly.

boundary or inside) will be indexed with -2 as they all have the same potential. The code implementing this sweep is listed below.

```
1   for(int i=0; i<N*N; ++i)
2   {
3     if(fabs(pts[i].x)>1.5-1e-10 || fabs(pts[i].y)>1.5-1e-10) pts[i].
          index = -1;
4     if(fabs(pts[i].x)<0.5+1e-10 && fabs(pts[i].y)<0.5+1e-10) pts[i].
          index = -2;
5   }
```

Note that we used small shifts of the order 10^{-10} to make sure that we capture the border points; remember never to compare for equality floating point numbers since this can lead to surprising results. In this pass we only change the value of the `index` member of the `pts` array to make it easy to distinguish the interior points from the ones on the boundary. We are now ready to compute how many interior points we have: this will be the total number of point in the `pts` array that have non-negative index.

```
1   *nphi = 0;
2   for(int i=0; i<N*N; ++i) if(pts[i].index>=0) pts[i].index = (*nphi)
        ++;
```

In this routine we also adjust the index of every entry in the `pts` array to point to an offset in the `phi` array. For the interior points this offset should be an unique number between `0` and `nphi-1`. The mapping is arbitrary and we just label this points sequentially as we encounter them in this sweep. Finally, with this information we can now fill in the `ne` and `pos` array.

```
1   for(int i=0; i<N*N; ++i) if(pts[i].index>=0)
2   {
3     int idx = pts[i].index;
4     for(int k=0; k<4; ++k) ne[4*idx+k] = pts[pts[i].ne[k]].index;
5     pos[2*idx+0] = pts[i].x;
6     pos[2*idx+1] = pts[i].y;
7   }
8
9   *nb = 2;
10
11  free(pts);
12 }
```

The most delicate step is on line 4 in the above listing: there we use the fact that `pts[i].ne[k]` has the offsets in the `pts` array of the nodes neighboring `i`. When we copy this information into the `ne` array, we first have to take into account that in the final `ne` array the offset for `i` is `idx=pts[i].index`. Since the `k` neighbor of `i` in `pts` is at offset `pts_neighbor_offset=pts[i].ne[k]`, the `phi` offset for this neighbor is `pts[pts_neighbor_offset].index`.

Putting it all together, the `main` routine calls the `setup`, sets the boundary values and the initial guess for the `phi` field, and then calls the `gauss-seidel` routine. The solution of the Laplace equation is then printed out, the gradient is computed at every point using the approximation

$$(\nabla\phi)_x \approx \frac{\phi(x+a,y)-\phi(x-a,y)}{2a}\,, \quad (\nabla\phi)_y \approx \frac{\phi(x,y+a)-\phi(x,y-a)}{2a}\,. \tag{11}$$

Finally, the flux is computed through two square surfaces: the smallest square that goes through the inner points in Γ and the largest one. The listing is given below

```
1  int main(int argc, char** argv)
2  {
3    int N = atoi(argv[1]);
4    double del = atof(argv[2]);
5
```

```
  int* ne = malloc(4*N*N*sizeof(int));
  double* pos = malloc(2*N*N*sizeof(double));

  int nphi, nb;
  setup(N, ne, &nphi, &nb, pos);

  double *phi = malloc((nphi+nb)*sizeof(double));

  phi[0] = 0;
  phi[1] = 100;
  for(int i=0; i<nphi; ++i) phi[nb+i] = 0.0;
  gauss_seidel(phi+nb, ne, nphi, del);

  for(int i=0; i<nphi; ++i)
  printf("res: %f %f %e\n", pos[2*i+0], pos[2*i+1], phi[nb+i]);

  double a = 3.0/(N-1);

  for(int i=0; i<nphi; ++i)
  {
    double gradx = (phi[nb+ne[4*i+0]] - phi[nb+ne[4*i+1]])/(2*a);
    double grady = (phi[nb+ne[4*i+2]] - phi[nb+ne[4*i+3]])/(2*a);
    printf("resg: %e %e %e %e\n", pos[2*i+0], pos[2*i+1], gradx, grady)
        ;
  }

  // determine the largest and smallest square contours
  // that fit in the area between the conductors.
  double maxx=0.0, minx=3.0;
  int imax, imin;
  for(int i=0; i<nphi; ++i) if(pos[2*i+0]==pos[2*i+1])
  {
    double x = pos[2*i+0];
    if(x>maxx) { maxx=x; imax = i; }
    if(x>0 && x<minx) { minx=x; imin = i; }
  }

  // Now compute the fluxes
  double fmax=0.0, fmin=0.0;
  // start at (x,x) and reduce x until you get to (-x,x)
  // the first and last point should contribute a/2
  // since they have the same contribution, by symmetry, we include
      only one
  for(int i=imax; pos[2*i+0] > -maxx+1e-10; i = ne[4*i+1])
  {
    double ey = -(phi[nb+ne[4*i+2]]-phi[nb+ne[4*i+3]])/(2*a);
    fmax += a*ey; // dl is in the x direction so l_\perp is in the y-
        dir
  }
  for(int i=imin; pos[2*i+0] > -minx+1e-10; i = ne[4*i+1])
  {
    double ey = -(phi[nb+ne[4*i+2]]-phi[nb+ne[4*i+3]])/(2*a);
    fmin += a*ey; // dl is in the x direction so l_\perp is in the y-
        dir
```

```
56      }
57
58      // there are 4 edges that each contribute equally to the flux.
59      // we only summed over one of them.
60      fmin *= 4; fmax *= 4;
61
62      printf("flux(min,max): %e %e\n", fmin, fmax);
63
64      free(ne);
65      free(pos);
66      free(phi);
67
68      return 0;
69  }
```

The code in lines 33–40 determines the offsets of the points on the diagonal $x = y$ that are the closest to the inner square and the outer one, `imin` and `imax`, and the positions of this points. The code below computes the flux going through one edge of the square, by from the point (x, x) to the left, first to $(x - a, x)$, then to $(x - 2a, x)$ and so on until we get to $(-x, x)$. For each of these points the flux is computed by taking the $(\nabla\phi)_y$ component perpendicular to the edge and multiplying with $dx = a$, the distance between points. For the corner points, the relevant segment is only $dx = a/2$.

As an example we plot here the contour plot for the solution of the Laplace equation for our geometry, the electric field, and the fluxes as a function of the step size $a = 3/(N - 1)$ of the lattice together with a fit to help us determine the limit $a \to 0$.

The flux is related to the charge enclosed on the inner conductor via Gauss' law: $\Phi = Q/\epsilon_0$. To relate to something physical we can assume that this geometry represents a section of two long square conductors of length $L \gg 1\,\mathtt{cm}$, such that neglecting the edge effects the total flux is $\Phi \approx -621.5\,\mathtt{V} \times L$ and the charge is $Q \approx -621.5\,\mathtt{V} \times L\epsilon_0$ (the flux result is taken from the right panel of the figure below.) The capacitance for this conductor arrangement is then $C = |Q/\Delta V| = 6.215\epsilon_0 \times L$, since $\Delta V = 100\,\mathtt{V}$ for our arrangement. To check whether this is a reasonable answer you can approximate this with 4 capacitors, one for each edge, connected in parallel. The distance between plates is $1\,\mathtt{cm}$ and the area of the plates is of the order of $1\,\mathtt{cm} \times L$. Using the standard results for the capacitance of parallel-plate capacitors, $C = \epsilon_0 A/d$, we get for this approximation a value of $C = 4\epsilon_0 \times L$, which is in line with our result.