Due on Oct 11, 2016 in Lab

Algorithms II - Lab 3

Random Processes

Goals: In the first part of this lab we want to see how we can use Monte Carlo to solve a difficult problem in scientific computing, locating the global extrema of a function. We will only consider functions of one independent variable. In the second part of this lab we implement a routine to do a random walk in 2D and see how we can use this to estimate the solution of Laplace's equation $-\Delta u = -(u_{xx} + u_{yy}) = 0$.

Part I - Using Monte Carlo for Optimization

We consider the 1D optimization problem of finding a point x^* in a given domain $D \subset R^1$ where the function $f(x)$ attains its *global* minimum. If we can solve this problem, then we can also find its global maximum by finding the global minimum of $-f(x)$.

The difficulty with optimization is that the function may have several local minima. Most minimization algorithms can be "trapped" by a local minimum. We are going to see how we can develop an algorithm which uses Monte Carlo sampling to estimate the location of the global minimum.

It would be fair to ask why we bother randomizing this algorithm. Why not just divide up the region into say 500 equally spaced points; then you will guarantee that you sample all the space just as well (even better, because you know there can't be any large gaps). There are several reasons to use pseudorandom sequences to sample the domain. One of the reasons is that in two or three dimensions we may have a very complicated geometry. Another reason is that when we go to higher dimensions, i.e., a function of several variables, then our 500 points in one dimension becomes 500^d in d dimensions. It should be noted that there are much better sampling methods than Monte Carlo and those should be used in practice.

Assume that we want to find the global minimum of $f(x)$ on $[a, b]$. We want to do this in as few function evaluations (i.e., evaluation of f at a point) as possible. Our first strategy will be to sample $[a, b]$ with n points $\{x_i^0\}_{i=1}^n$ using Monte Carlo and evaluate the function at each of those points. As we sample each point we keep track of which x value has the smallest function value. From all the sampling points we choose the location where $f(x)$ takes on its minimum value and that will be our answer. This algorithm will converge as $n \to \infty$. We will consider modifications to this algorithm in the exercises.
We consider three functions, each more difficult than the previous for locating its minimum.

$$f_1(x) = |\cos \pi x| \quad \text{on } [0,1]$$

$$f_2(x) = \cos(x) + 5 * \cos(1.6 * x) - 2 * \cos(2.0 * x) + 5 * \cos(4.5 * x) + 7 * \cos(9.0 * x) \quad \text{on } [2, 7]$$

$$f_3(x) = -\Big[\text{sech}^2(10.0 * (x - 0.2)) + \text{sech}^2(100.0 * (x - 0.4)) + \text{sech}^2(1000.0 * (x - 0.6))\Big] \quad \text{on } [0, 1]$$

To compare codes, please use a seed of 56789 for easier grading.

---

1. Plot the three functions $f_i(x)$, $i = 1, 3$ on their domains. From the graphs estimate the location of the *global minimum* and all *local minimum* of the functions.

---

2. Write a code to implement our sampling algorithm for locating a global minimum of a function. Your code should have the following structure:

- Input: `n` the number of sampling points;
- `xl,xr` the endpoints of the interval where the function is defined (so you can map the random point to that interval)
- Output: the location and value of the minimum that the algorithm found.

Make sure your code is working properly by finding the local minimum of our "easy" function $f_1(x)$. For $f_1(x)$ we know the exact location of its global minimum. Make a table of n and the error in the location of the minimum as n increases. For example, $n = 25, 50, 100, 200, 400, 800, 1600$.

---

3. If we have a good approximation to the location of the global minimum, it probably doesn't make sense to keep sampling in the entire domain; you saw in #2 that sometimes doubling the number of points didn't improve the accuracy. A better strategy might be that once we have an approximation then we only sample around that approximation to zoom in on a more accurate approximation to the location of the minimum. Of course, if we have too few random points in our initial sweep and have an erroneous location, then zooming in is not going to help us; that is why this approach is not guaranteed to converge unlike the brute force approach in #2. After we have located our approximation, say x_1^* to the local minimum by generating n random points, then we will sample in the interval $[x_1^* - \delta, x_1^* + \delta]$ using say $n_1 = n/2$ points (of course you can change this) to get points $\{x_i^1\}_{i=1}^{n_1}$ and we can determine our second approximation to the global minimum x_2^* by evaluating $f(x)$ at each of these random points. The procedure can be repeated. At the last step, we choose the location where $f(x)$ takes on its smallest value; this is our approximation to x^*. Remember that the algorithm is not guaranteed to converge to a *global* minimum but if our initial sampling is good enough, we should be able to capture the minimum for most functions.

Modify your code to input `nzoom`, the number of zooms we will use; if `nzoom=0` you should have the algorithm from #2. We want to run the code for f_2, f_3 and compare the number of function evaluations required as a function of `nzoom` to find the global minimum and its location to a desired accuracy. You should choose the calculations which best support your conclusion.

Part II - Solving Laplace's equation using Random Walks

In this lab we want to see how we can use Monte Carlo simulation to approximate the solution to the second order PDE $\Delta u(x,y) = u_{xx} + u_{yy} = 0$ on a given domain such as $[0,1] \times [0,1]$. Of course we need to also specify boundary conditions.

Typically, one would use a technique such as finite difference methods to approximate the solution of a PDE. Recall that to discretize this time-independent PDE we first discretize our spatial domain. To this end, we overlay the domain with a grid; for simplicity we choose a uniform rectangular grid defined by

$$x_0 = 0, x_{n+1} = 1, \quad x_i = x_{i-1} + \Delta x, i = 1, \ldots, n \quad \text{where } \Delta x = \frac{1}{n+1}$$

$$y_0 = 0, y_{n+1} = 1, \quad y_i = y_{i-1} + \Delta y, i = 1, \ldots, n \quad \text{where } \Delta y = \frac{1}{n+1}$$

For simplicity we choose $\Delta x = \Delta y$.

If we set u on the boundary of our domain then our goal is to determine an approximation to u at each interior point (x_i, y_j), $i, j = 1, \ldots n$ (because on the boundary we simply set u to be the given boundary value). If one uses finite difference techniques, then a difference equation is written at each interior node where the difference equation is obtained by replacing the derivatives in the original PDE with difference quotients.

To use Random Walks to approximate the solution we take a different approach. At each interior node where we want an approximate we take M random walks. Each random walk is stopped when the walk reaches the boundary. We then record the value of the boundary condition at the point where the random walk terminates. We repeat this process M times and then average the values to find the approximation at that node. We then repeat the process at the next node.

---

1. In this problem we want to write a routine to perform a single random walk in two dimensions. If this was all we were going to use the code for, we might assume that the object always starts at $(0,0)$ and have a tiled square of $(2n+1) \times (2n+1)$ "tiles" of length one and then we could move one tile to the north, south, east or west; the random walk would continue until it reaches the boundary, i.e., when x or y is n. Because we will use this routine to approximate the solution to Laplace's equation we will write the code in a slightly different way. As described above, we think of overlaying our domain which we assume is a square region (say $[xl, xr] \times [yb, yt]$) with a uniform grid with spacing $\Delta x = (xr - xl)/(n+1)$ where $n+1$ is the number of cells in each direction. Then we can take a step of length Δx to the north, south, east or west. We also want the capability to start the random walk at each cell point, i.e., $(xl + i\Delta x, yb + j\Delta x)$. Thus we should input the boundaries of our rectangle xl, xr, yb, yt; and the number of cells in each direction, and the starting point (i, j) whose coordinates are $(xl + i\delta, yb + j\delta)$. For this problem you need to output the number of steps, and the path so you can plot it.

Write your code and plot the path of a random walk starting from 4 different interior locations in your grid. Use the box $[0,1] \times [0,1]$ with 10 cells in each direction.

2. We now want to use our code from # 1 to approximate the solution to our PDE

$$-\Delta u = 0 \quad \text{on } \Omega = (0,1) \times (0,1)$$

$$u = e^y \cos x \quad \text{on the boundary of } \Omega$$

a. Verify (on paper) that $u(x,y) = e^y \cos x$ is a solution to Laplace's equation; thus you know the exact solution to the PDE so you can use it to compute an error.

b. Write a code to use your routine in #1 to approximate the solution to this PDE. Make a table of M (the number of random walks from each interior node) and the ℓ_2 norm of the error vector (i.e., the error at each interior node). Be sure to normalize your standard Euclidean length by the length of the vector. Use $\Delta x = 0.1$ and choose $M = 10, 20, 50, 100$.

c. Now we want to hold M fixed at 100 and decrease Δx. Approximate your solution for $\Delta x = 1/4$, $1/8$, $1/16$ and $1/32$ and tabulate your errors. What do you conclude?