AM 129 HW6 Deliverable

Sarosh J. Sopariwalla
December 5, 2020

Abstract

We work on using numerical methods to quantify the stationary distribution of a particle moving randomly on a lattice. We do the primary calculations in the C programming language and plot the results using Matplotlib in a Python3 file called Visualization.py.

Sparse Matrix Data Structure

In general, structs are useful to work with since we can define a group of variables under one name in a specific block of memory. This allows us to to access all these variable via a single pointer; further, this is beneficial because we can store various datatypes under one address. A sparse matrix is one in which there are so many zeros that it becomes quicker to only store certain pieces of information from that matrix rather than that whole thing. In essence, we could express any $n \times n$ matrix in three one-dimensional arrays. The first array contains all the non-zero values in our $n \times n$ matrix. The second and third arrays contain the column and row index for each corresponding non-zero entry. Consider the matrix $\begin{pmatrix} 1 & 0 & 0 \\ 0 & 5 & 0 \\ 0 & 7 & 0 \end{pmatrix}$

We can put this into a sparse matrix structure by defining array one to be [1, 5, 7]. (Notice this contains all the non-zero elements of our matrix.) Then we define the second array to be [0, 1,1] (because the one is in the "0th" column since we index starting at 0. Likewise 5 and 7 are in the "first" column.) Likewise our third vector is [0, 1, 2] since our non-zero entries are in the 0th, 1st, and 2nd row respectively. Clearly for small matrices or ones with few zeros, this method is inefficient but it could be quite useful for large matrices with many zero entries.

This brings us to an important question: Is the sparse matrix data structure helping the efficiency of our program? When running our code with 100 grid points, we see our algorithm converges in 15,550 iterations which involves matrix multiplication each time. Even for modern computers so many dense matrix multiplications would take incredibly long; however, it seems that performing 15,550 matrix vector products happened pretty quickly in our case. Clearly, defining our sparse matrix data structure was helpful for less-intensive computations. The only question to answer now is why?

We know that a matrix-vector product with a dense matrix of size $N \times N$ requires on the order of N^2 operations. To show this, consider how we multiply a matrix $\mathbf{A}$ and a vector $\mathbf{x}$.

$$\mathbf{y}_i = \sum_{j=1}^{n} \mathbf{A}_{i,j}\mathbf{x}_j$$

Thus, if $\mathbf{A}$ has n rows we need to sum n terms n times. In our sparse matrix format, this multiplication is simplified like so:

$$\mathbf{y}_i = \sum_{k=0}^{nnz[i]} rowVals[i][k] * x[cols[i][k]]$$

Now instead of summing n terms n times we sum very few terms n times. In our case, our matrix only has a maximum of perhaps three or four non-zero terms per row so we are doing less than $4n$ operations.

Behavior for Various Grid Resolutions

Grid Resolution	Iterations to Converge
100	1,550
200	1,449
350	5,0139
478	99,776
479	N/A

Unsurprisingly, as the grid resolution increases, the number of iterations to converge increase. Let's take note that at 479 grid resolution our, algorithm fails to converge; this is because our maximum number of iterations is capped at 100,000 and our program will not end until we reach a threshold less than 1×10^{-6}. We see that after 100,000 iterations, the difference is still a little too high at 1.001347×10^{-6}.

PMF Plotted with Matplotlib