Overview:

I have implemented both sieving by differences (SD) and modified sieving by averages (MSA) to solve the shortest vector problem (SVP). In my code you will find two main functions for the SVP: old_sieving() and sieving() which are described in the two subsections below.

old_sieving():

Design:

This function contains my initial approach at sieving as well as any redundant optimisations I made during experimentation. We first generate a list of random basis vectors which are stored as a tuple (u, x, norm) so we have easy access to each lattice point's data. We then fill a new list of lattice points, using either SD or MSA, by selecting random pairs of points from our original list. We add the new points generated to the new list subject to 3 restrictions:

1. The new vector has norm less than both parents
2. The new vector is not already in our new list (collision)
3. The new vector is not the zero vector

If a minimum is found, it is stored. We fill our new list with new lattice points until it matches the size of our old list and then repeat the process on this new list.

Justification

By iterating through lists of a set size, it prevents unnecessary memory growth caused by storing each new point generated. We know that taking the difference or average of smaller vectors is more likely to produce a vector that is smaller still. Therefore, this method also increases the rate at which our algorithm approaches the minimum vector since at each iteration, the average norm of each list decreases due to (1). Since the shortest vector exists within a small bounded region containing the origin, our algorithm is more likely to encounter the shortest vector at each iteration.

(2) prevents our algorithm from becoming less efficient since identical points reduces the number of distinct combinations of points to operate on. In the case of MSA, the list would very quickly fill with identical points since the average of a vector with itself returns itself. In the case of SD, the list would very quickly fill with 0 vectors since the difference of a vector with itself is the 0 vector. (3) prevents the zero vector being the solution to the SVP as it is not allowed in the definition of the problem.

Optimisation

By requiring our new list of lattice points to match the size of the old one, we encounter a problem: it is possible for our algorithm to come to a standstill by being unable to find enough lattice points shorter than both parents. To solve this, we define a function that fills our list with random lattice points if our algorithm stagnates i.e. if no new points are added to our new list. We use a simple counter for the time spent on each iteration which causes the function to activate if a threshold is reached.

This is not the most time efficient approach as the threshold is static so our algorithm may have stagnated long before the threshold is reached, particularly if the size of our list is small. A better approach would be to initialise a timer that resets after each new point is added. We do not choose to do this as this is only our naïve approach. By flooding our list with random points, we also have the added benefit of increasing our exploration of the lattice.

sieving():

Design/ Justification

This function follows a similar approach to above, but with a different structure. Instead of adding each new lattice point to a new list, we add it to the current list before removing the vector with the largest norm. This allows our list to contain the n shortest vectors we have found at each step, greatly increasing the speed of convergence to a minimal vector. By removing the longest vector each time, we also maintain our memory optimisation.

We also find that we can use both sieving techniques to increase performance: for each pair of points, we choose either SD or MSA with a probability of 50%. Instead of doing SD and MSA in terms of u and v, we do them in terms of x and y to reduce excessive computation using inverses which would result in messy computation with floats since the inverse of basis B does not contain integer elements.

Optimisation

Since our list is still finite, it is possible that our algorithm can come to a standstill as it is unable to find a shorter vector given all combinations of points in our set. To mitigate this, we can increase the size of our list. Comparing a list size of 10, 100, 1000, and 10000, we find that all but size 10 converge to the same shortest vector, regardless of sieving method. With a list of 10000 tuples, our algorithm is slow since sorting and checking 10000 elements for collisions is slow. As a result, we choose a list size of 100 as it is much quicker and still converges to the same minimum.