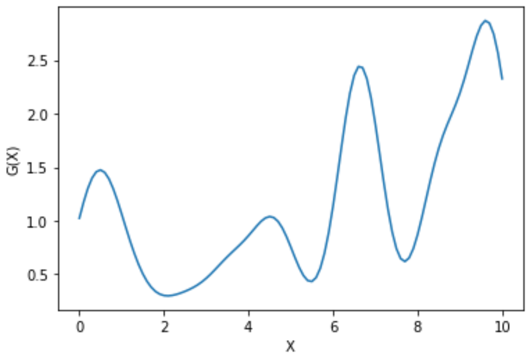
Background

In the field of artificial intelligence, it's an essential capability to be able to maximize or minimize the value of a function. This project explored two methods of doing so: simulated annealing and gradient ascent. The function in question that was optimized by these methods was a Sum of Gaussians function. Gaussian functions in their most primitive form are basically bell curves but when these are summed together, they create more interesting surfaces and curves- especially when extended into multiple dimensions. Examining figure 1, one can see a randomly generated two-dimensional Gaussian with 50 centers. An example of a 3d surface created from a Sum of Gaussians is shown below in figure 2.



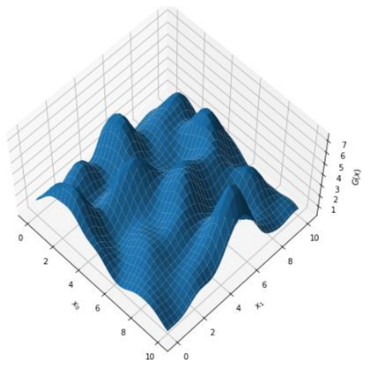


Figure 1 and 2: 2d and 3d Sum of Gaussian plots

Gradient Ascent

The first code developed was greedy.py. It is named greedy.py because it uses a *greedy* scheme for its gradient ascent algorithm. Generally, the algorithm attempts to maximize a function by "walking" up gradient of a function, pursuing the direction that provides the *steepest ascent* at any given time. (Russell 236) A common metaphor often used to explain this action is to imagine that one is trying to go up on a hill in a foggy valley where the line of sight is only a few feet. The best course of action here is to just follow the grade of the hill up as best as possible, where eventually the top of *some* hill will be reached. (Russell 236) At the beginning of a run, a seed, a sum of gaussians dimension constant and a number representing the quantity of gaussians will be set from command line positional arguments. A randomized S.O.G. function is generated using all three of these values and a d-dimensional start location vector x in the [0,10] d-cube is generated using the seed. Once the code starts, it will step in the direction of the increasing gradient until a threshold is met. To prevent "overstepping", the step size at any given time in the algorithm is only one percent of the gradient at a given point:

$$\text{Increment} = 0.01*\nabla xG \text{ , where } \nabla xG = [\frac{\delta G}{\delta x0}, \frac{\delta G}{\delta x1}, \cdots, \frac{\delta G}{\delta x_{d-1}}] \text{ [Ref 2]}$$

A run is considered complete upon one of two conditions: one, the code ran for 100,000 iterations, or two, the code's calculated increment is within the tolerance 1e-8. This method does have a few key shortcomings. It won't always pick the global maxima, nor the highest local maxima. Sometimes it won't even find a maxima at all. If one were to look back at figure one, numerous local maxima can be located at x points 0.75, 4.25, and 6.5. The global maxima can be found around 9.5. Due to its greedy nature, if the gradient ascent algorithm's starting location were to be at a starting location such as x=5, the gradient ascent algorithm would find the maxima to the left first because the gradient travels that direction-it would effectively miss the much higher maxima to its right. If the algorithm were to have a starting location at a flat region such as a ridge or plane, the increment may be so small that the code essentially dies from being within the tolerance [no substantial gradient up or down, so it doesn't move]. There is another method that seeks to address these problems.

Simulated Annealing

Sa.py implements an alternative approach to optimization called simulated annealing. All the general details of the sum of gaussians function, random seed, and random start locations are the exact same as greedy.py—the primary difference is the method of finding a maximum. In gradient ascent, the "good" move would be the one that moved in the direction of the gradient. A "bad" move would be one that moves opposite of the gradient. In a greedy approach, only "good" moves are taken which leads to the possibility of getting stuck in a local minima or ridge. What simulated annealing aims to do is provide a mechanism for escaping these via a "temperature" value that starts off very hot, and slowly cools down [decreasing in value]. If a move improves the current situation, it is accepted, however if the move is considered worse, simulated annealing will accept the move based on a random probability derived from the metropolis criterion:

$e^{\left(\frac{G(y)-G(x)}{t}\right)}$, where $G(x)$ and $G(y)$ are the old and new point and t is the temperature.

The probability is dependent on the temperature's value. If it is higher, the move is more likely to be accepted regardless of if it is good or bad. As the temperature decreases, the bad moves are less likely to be accepted. (1 Phillips) The general idea is to have the temperature on a schedule so that it is high in the beginning to find a *close enough* answer to the global maximum but then it decreases slowly as it continues so that by the end it won't randomly veer away by accepting bad moves. Sa.py uses a linear temperature schedule with a coefficient of one, mainly so that it wasn't too fast and not too slow. Exponential seemed like it would be a bit too fast, and it would end up resulting in losing efficiency. Two more design considerations were made to improve performance: addition of a simulation max, and also bounds checking. Simulated annealing can possibly scrap a good position with random moves, but there's no reason to throw this value out. The maximum value reached during the simulation is kept up with and is the final value printed, although it may not actually be the point simulated annealing stopped on. In addition, if a random point is generated that exists outside of the d-cube, the program replaces the point with a point that is on the boundary e.g. [3,11,-1] becomes [3,10, 0]

Results

Both scripts had to be run all combinations of dimensions 1,2,3, 5 and number of centers 5, 10, 50, 100 with 100 different seeds. This brought the total amount of runs for each script to 1600, totaling 3200 runs. In terms of wall clock time, this took around two hours. A bash script was written to mass run these cases with multiple cores and to dump them to files for each seed and type of run. After these finished, a final post-processing data script was run to tally the number of times that sa.py outperformed greedy.py. The results are tabulated in figure 3.

In almost all cases, simulated annealing beat the greedy approach, which is honestly not that unexpected. Looking at figure, one can see that on average it performed best on all dimensional variants of the N=100, and the worst on the N=5-presumably as the lower number of centers made the surface much more smooth and therefore subject to having to take less random moves. It is also clear that on the N=100 cases, simulated annealing was very powerful as it only dropped below 80 wins once the fifth-dimension cases occurred. This makes sense as more complex surfaces [higher N's] will have higher amounts of local maxima that the greedy approach can get stuck in. Greedy and simulated annealing were considered tied when they reached 1e-8 of each other. When viewing figure 4, one can see that simulated annealing tied with greedy hill climbing for a substantial amount of N=5 cases, more than likely because greedy is not a bad choice for a simpler surface. Moving on to individual values, the lowest and highest values are highlighted in red and green respectively. Simulated annealing performed the worst on the fifth-dimensional N=5 case and the best on the five-dimensional N=50 case. When viewing smaller subtrends in each row, it seems to be apparent that as the number of dimensions increase, simulated annealing performance tends to drop. This is more than likely because the temperature schedule doesn't cool down fast enough.

	D=1	D=2	D=3	D=5
N=5	78	73	69	68
N=10	82	79	87	92
N=50	87	84	93	95
N=100	94	94	82	77

Figure 3 Table showing number of times Simulated Annealing beat Greedy

	D=1	D=2	D=3	D=5
N=5	19	22	27	31
N=10	0	0	0	0
N=50	0	0	0	0
N=100	4	0	0	0

Figure 4 Table showing the number of times simulated annealing tied with greedy

	D=1	D=2	D=3	D=5
N=5	3	5	4	1
N=10	18	21	13	8
N=50	13	16	7	5
N=100	2	6	18	23

Figure 5 Number of times Greedy beat simulated annealing

Conclusion/Further Work

For further learning, it would be beneficial to also implement a genetic algorithm search as well as a local beam search. Currently, the project technically supports greedy ascent, simulated annealing, and a random walk search because the temperature schedule can be set to zero to emulate picking only random moves. An improved schedule, and an alternate bound checking procedure could also be worthwhile investigating. The two main limitations of this project are the fact that the starting points are random and the results past 3 dimensions are difficult or impossible to visualize. It's hard to exactly pin down if the code itself is not working as intended, or if it's having just *bad luck.* After a run, a plotting tool exists that can accurately plot the sum of gaussians function along with the points evaluated by sa.py and greedy.py, but once the dimensions past the third are reached, it cannot be used anymore. This makes it hard to validate. After doing this project however, the inner workings of gradient ascent and simulated annealing are much more understood.

References

Phillips, Joshua Dr. (2022, September 14). *Local Search Strategies*. [13]. Department of Computer Science. Middle Tennessee State University. https://jupyterhub.cs.mtsu.edu/azuread/services/csci4350-materials/private/2022-09-14_Chapter_4.pdf

Phillips, Joshua Dr. (2022 October 3). *Open Lab 2: Solving Problems by Local Search*. Department of Computer Science. Middle Tennessee State University. https://jupyterhub.cs.mtsu.edu/azuread/services/csci4350-materials/private/OLA2-4350.pdf

Russell, Stuart J. (Stuart Jonathan). (2020). Artificial intelligence: a modern approach. Upper Saddle River, N.J. :Prentice Hall