Daniel Votaw

Computational Physics

Final Project

**Introduction:**

This project demonstrates *simulated annealing* (SA) as an optimization technique. SA is a stochastic method for locating global extrema of functions. The stochasticity is used to guarantee that the algorithm will eventually converge to the **global** extremum, whereas deterministic optimizers often face the problem that they can get stuck in local extrema. If you have some function which potentially has many local extrema, but you need to find the global extremum, deterministic methods are not suitable. For purposes of this project, I’ll focus on cases where the goal is to find a global minimum, but the algorithm can be modified in the obvious way to look for maxima instead (to maximize f(x), just minimize –f(x)).

The function to be minimized will be referred to as the “cost function” f, and it takes some D-dimensional vector **x** as an input. The possible values of the components of **x** live in some predefined parameter space Ω. Most of the time, we are not as interested in the minimum value of the cost function, but rather the values of its inputs such that it is minimized. Therefore mathematically, the algorithm is designed to find:

The idea behind simulated annealing is basically an extension of the Metropolis algorithm (discussed in class), but where the temperature of the Boltzmann acceptance distribution is “cooled” as a function of Monte Carlo time in order to drive the system to its “ground state” (i.e., find the minimum of the cost function). The ability of the Metropolis accept/reject routine to accept a move to a point with a higher cost, which may seem counterintuitive, is what allows SA to evade local minima, and what will eventually guarantee its convergence to the global minimum once the parameters and “cooling schedule” have been set adequately. As is often the case with Monte Carlo methods, we will see that this algorithm is often sub-optimal, in low-dimensional spaces, but extremely efficient in high-dimensional spaces.

We will see that for a given set of algorithm parameters, at low dimensions, SA will often iterate over the volume of the entire parameter space multiple times, making it more efficient to simply use a brute force search. However for the same set of algorithm parameters, at high dimensions, the SA algorithm may only need to iterate over a very small fraction of the space in order to find the global minimum. For a particular implementation shown in this project, the dimensionality where SA begins to outperform brute force in terms of volume is somewhere between 4 and 5. And the dimension-dependence of the volume performance is very strong; in this particular case, the scaling seems to obey a power law of the form (Fraction of space iterated over) ∝ Dp, where p ≈ -10.7 (see volume\_analysis.plt). Of course SA is a stochastic method, so the exact values of the runtime and fraction of the space iterated over vary with each run. They will depend on the initial conditions, the shape of the cost function, and the algorithm parameters, but to analyze the “average behavior”, we can simply take multiple runs and average the results.

# To demonstrate SA, I have chosen to use it to find the global minimum of the Rosenbrock function in 3 ≤ D ≤ 7 dimensions. In D = 3, this function has exactly one minimum at x = (1,1,1). In dimensions 4-7, there is a global minimum at x = (1,1,…,1), and a local minimum at x = (-1,1,…,1). I tried to explore dimensions higher than 7 as well, but the algorithm doesn’t seem to converge to any particular minimum, and I’m not aware if any exist. So go beyond 7 dimensions at your own risk. In this case, knowing whether the algorithm converged to the correct minimum is easy, because the answer is known a priori. But I will also discuss how to check convergence if the answer is not known a priori. The parameter space on which I will minimize the Rosenbrock function is Ω = [-5,5]D.

# The Rosenbrock function is real-valued (as opposed to only integer-valued), but there must be some kind of discretization in any computational approach. In the code, there is a parameter called “LATTICE\_SPACING” which specifies the discretization of Ω. It is nominally set to 1.0, meaning that the annealer jumps between integer values only. I chose this for simplicity, and because I happen to know that the exact minima lie at integers. In a case where the location of the minimum is not known, one can start with a coarse discretization and iteratively refine the discretization until a desired precision is reached.

**How to run main program:**

1. Compile with “make –f make\_annealer”.
2. Run with “./annealer.x”.
3. See the output in the terminal (also appended to file “results.txt”).

Notes:

* No command-line arguments.
* Unique PRNG seed is generated based on the time.
* Most of the tunable parameters are in “annealer\_main.cpp”, however the cooling function and adaptation function “annealer.cpp” can be played with as well.
* Can turn on verbosity in “annealer\_main.cpp” for additional information to print to the terminal.

Additional files:

* results\_i.txt: Previous results which have been saved for analysis.
* results\_avg.txt: The averaged results of the results\_i.txt files, already logarithm-ized for plotting convenience. This is the input to the “.plt” files.
* converter.cpp/converter.x: A quick-and-dirty program to combine the information in all of the results\_i.txt files, writing the final result to “results\_avg.txt” for plotting. If you want to use this for some reason, it can just be compiled in the command line with “g++ -o converter.x converter.cpp”.
* time\_analysis.plt: GNUPlot file to show the dependence of the time to convergence on the dimension of the parameter space.
* volume\_analysis.plt: GNUPlot file to show the dependence of the fraction of space iterated over on the dimension of the parameter space.

**Analysis:**

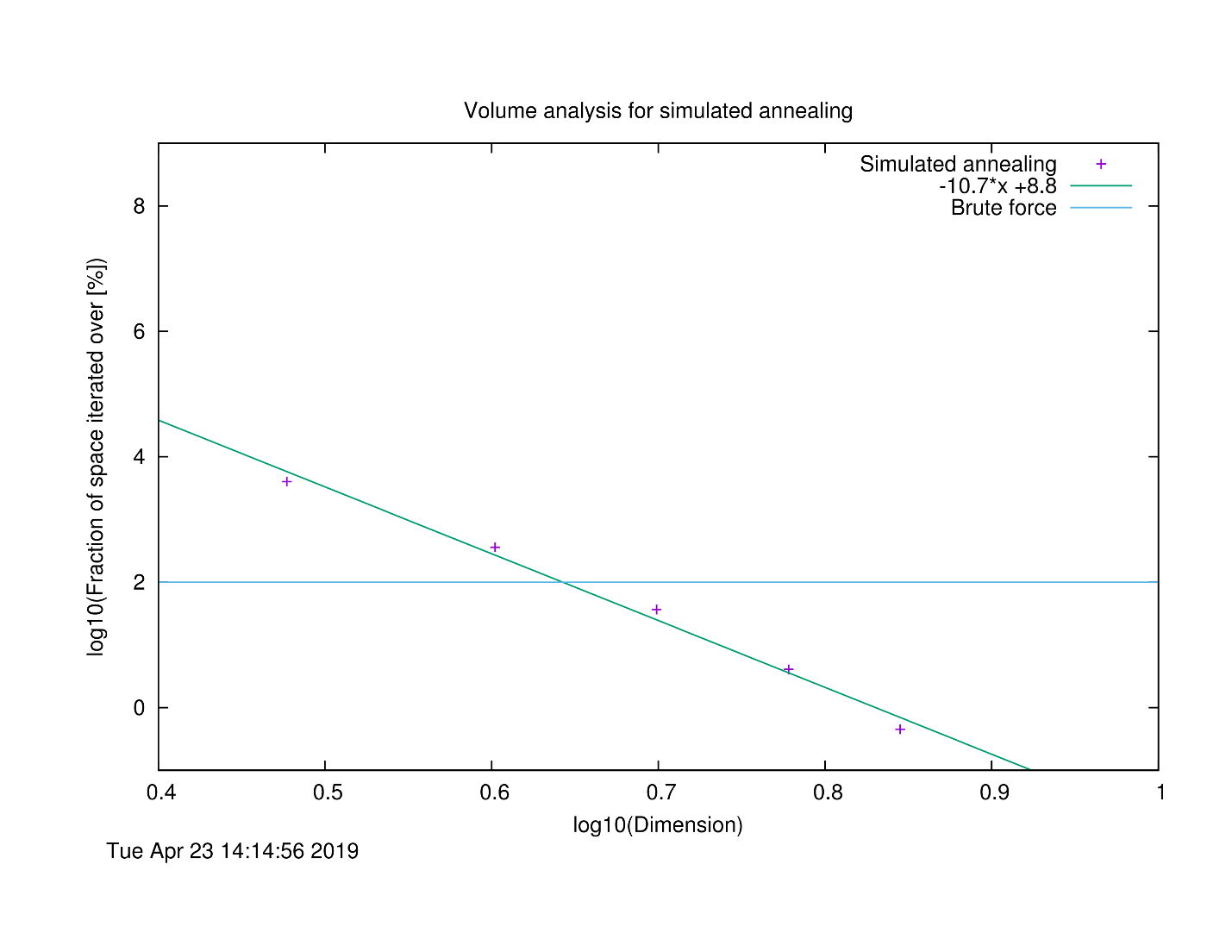
The main points of this analysis are that simulated annealing is a very nice way to find global minima even if there are arbitrarily many local minima, and that it tends to perform very well (after tuning the algorithm parameters suitably) in high dimensions. So I argue that if you need to perform a high-dimensional minimization where there are potentially multiple minima, that the overhead involved with tuning an SA algorithm for your particular problem is worthwhile, rather than performing a brute force search. Individual aspects of the analysis will be discussed in the following sections.

Convergence testing:

If the location of the global minimum is not known a priori, convergence can be tested using “random restarts”. This simply involves running the code multiple times, with different initial conditions (which will happen automatically because of the unique PRNG seed), and verifying that it converges to the same minimum each time. The parameters can be tuned until the same minimum is reached an arbitrarily large fraction of the time. With a “good” set of parameters, it can be essentially guaranteed to find the right minimum (fidelity).

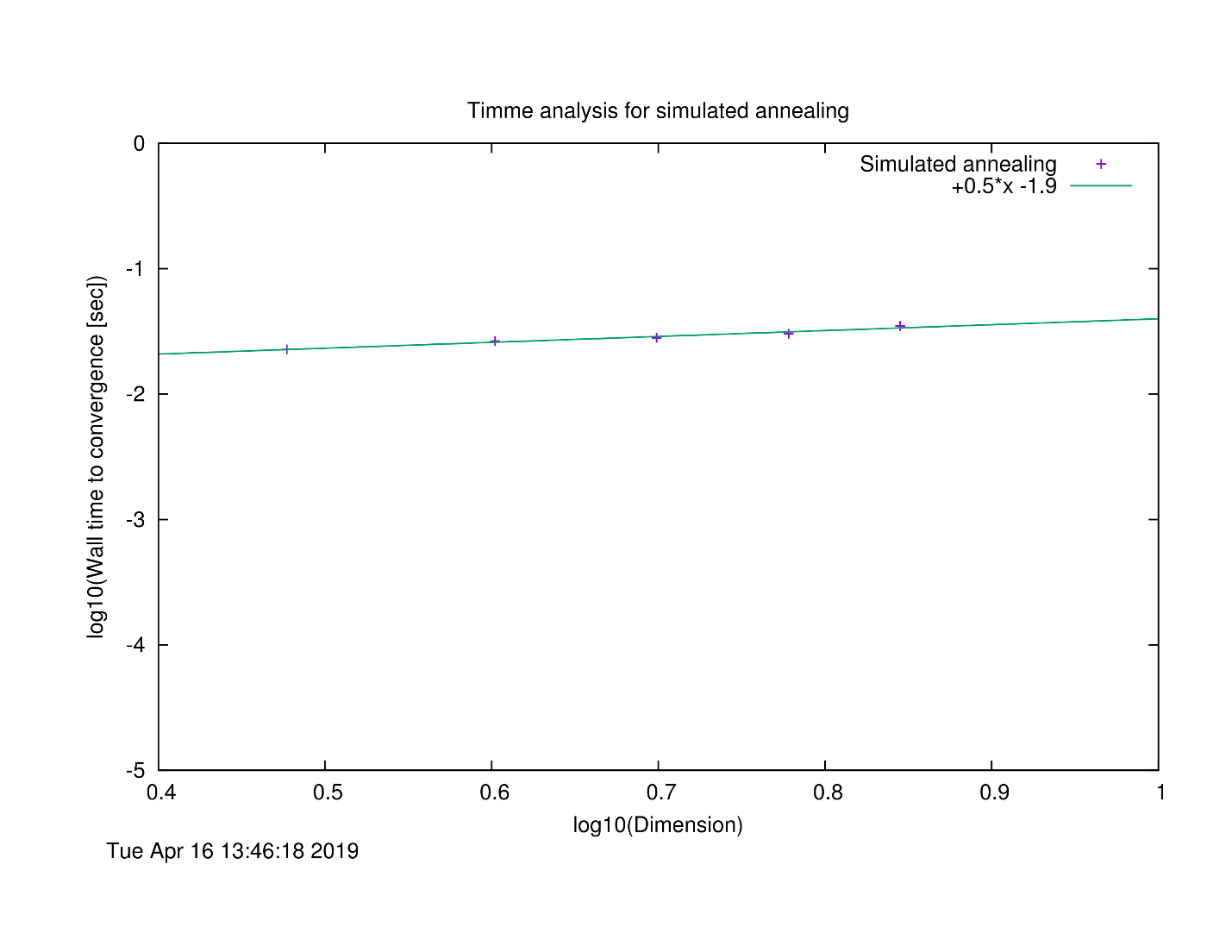
However there is a tradeoff in performance. If the parameters are not optimized for efficiency, it may take a long time for the code to converge. So the user should take care to find the right balance of fidelity and efficiency. Relatively small deviations from the optimum parameter set can lead to dramatic worsening of either the fidelity or the efficiency. In general, the following changes to the algorithm parameters tend to increase fidelity and decrease efficiency: increasing the initial temperature, decreasing the minimum temperature, slowing down the cooling (bringing the “cooling rate” parameter closer to 1 from below), and harshening the cooling criterion. With these ideas in mind, the developer of the algorithm should walk on a tightrope trying to optimize these quantities until the desired level of fidelity and efficiency is achieved.

Volume performance: (volume\_analysis.plt)



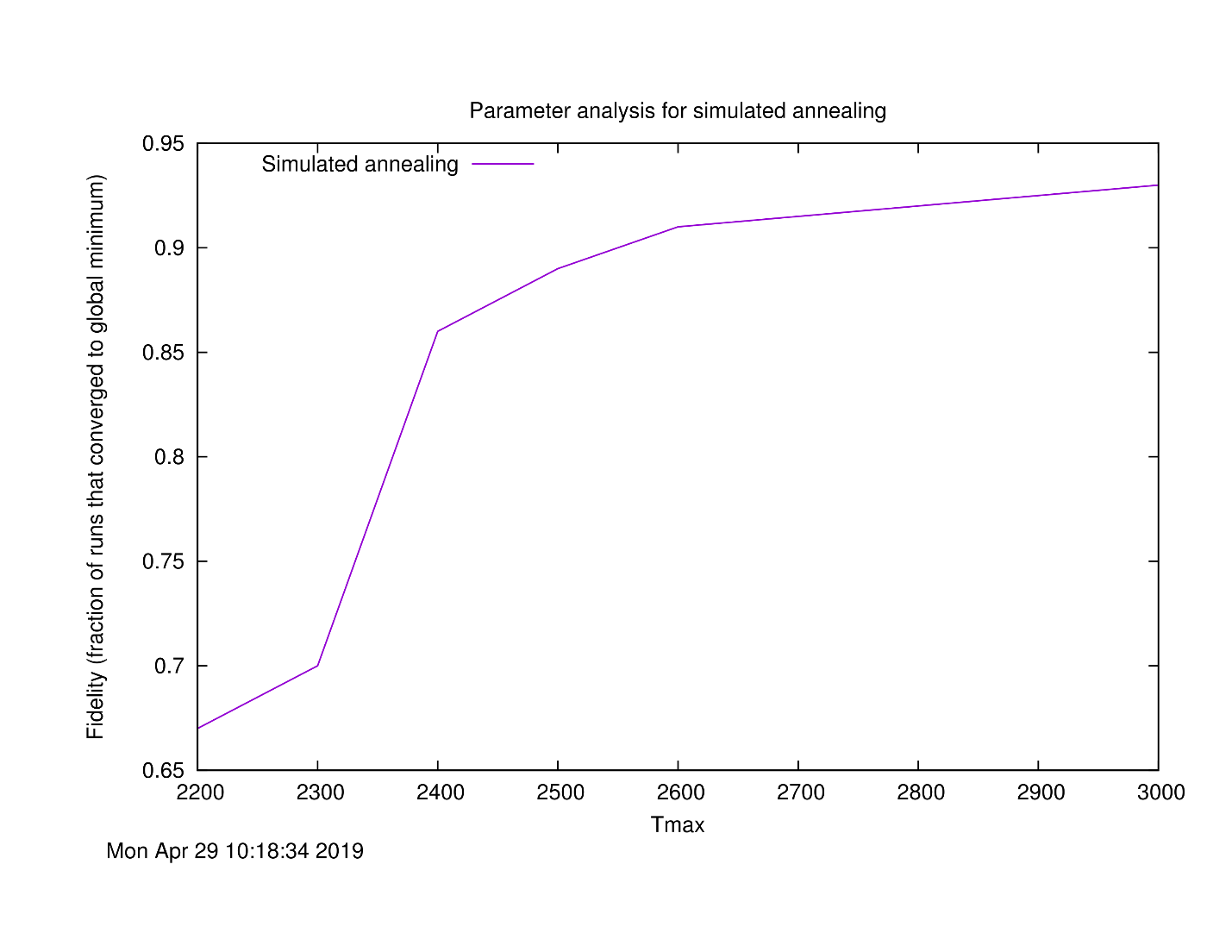
This plot shows how the SA algorithm performs in volume as a function of the dimension of the parameter space (with all algorithm parameters fixed). The “volume” of the discretized space is just equal to the total number of points, and the fractional space iterated over is the ratio of the number of total iterations to the volume. The parameters were set to values that gave a high degree of fidelity in 7 dimensions, and the same parameters were used for the lower dimensions. The plot shows a very strong power law dependence, and in this case the SA outperforms brute force for D > 4.

Time performance: (time\_analysis.plt)



This plot shows the wall time elapsed during the annealing as a function of the dimension of the space. These are from the exact same runs as the data in the previous section, so all parameters are the same. For this implementation of SA, it appears that the wall time increases as the square root of the dimension. So while it takes longer to find the minimum in higher dimensions, the growth is slow. This means that even a very high-dimensional optimization can be done in a reasonable amount of time.

Dependence of fidelity on algorithm parameters: (Tmax\_analysis.plt)



This plot shows the fraction of the times that the SA algorithm converges to the true global minimum in D = 7 as a function of the initial temperature parameter (all other parameters held constant). Increasing the initial temperature gives the algorithm more time to search for the global minimum as it cools, and so one would expect that at higher initial temperatures, the likelihood of finding the global minimum should increase. That is what I observe for this implementation. So to increase fidelity, the initial temperature should be set to a high value (compared with the typical differences in the cost function between points). However increasing the initial temperature also affects the time that it takes for the algorithm to converge. So unless you want to be waiting a few minutes for it to finish, the initial temperature shouldn’t be set *too* high. And if the initial temperature drops below a certain value, the fidelity tends to worsen very rapidly. The algorithm cools too quickly to settle into the correct “ground state”.