Simulated Annealing

Basis of my Algorithm

I based my algorithm on the fundamental similarities between simulated annealing and metropolis sampling. The essential idea is to form a probability distribution p(x) using the function f(x) that we wish to minimise. This should be done in such a way that the minima of f(x) correspond to the modes of the probability distribution p(x), for example take p(x) $\propto exp(-f(x))$. This is interesting because if we can sample from p(x), then the highest probability samples will be those closest to the global optimum.

Metropolis sampling seeks to generate dependent samples from p(x) in the following manner: take a current sample x and then select a candidate sample x' using the proposal distribution q(x'|x). We then accept this proposed sample with probability f(x,x'). By setting $f(x, x') = \min(1, \frac{p(x')}{p(x)})$ (assuming symmetric proposal distributions) we can ensure that in the limit, samples are generated from the distribution p(x). The key difference between metropolis sampling and simulated annealing is that in the latter, we do not sample from a fixed distribution p(x), but rather allow this distribution become more peaked (concentrated around the minima) as time progresses. This is typically performed by exponentiating $p(x)$ by 1/T where T is the 'temperature', which decreases with time as we cool the probabilities.

With this framework in place, there are some obvious modifications that could be made to the basic simulated annealing algorithm I wrote. I decided to implement a range of **cooling schedules**, which determine the rate of decrease in the temperature T. My implementation can be found in a nested function called 'cool_t( )' within 'anneal( )' in the appendix. I also implemented a range of un-normalised **probability functions** p(x), within the function 'energy( )' in 'anneal( )'. NB: in further explanations I may use the term p(x) synonymously with energy. After this, I implemented a range of proposal distributions within the function 'propose( )' that is also nested within 'anneal( )'. If we consider all of these factors together, then we have a great deal of room to make changes to the basic algorithm proposed, the pseudocode for which I have written below for clarity.

Simulated Annealing

```
0.T=T0 % Initial Tempeature
1.x=x % select a random point as an initial sample
2.Compute f(x)
3.use f(x) to compute un-normalised probability p*(x)

4.while function evaluations < 1000
5.  Generate x' according to proposal distribution q(x'|x)
6.  Compute f(x')
7.  use f(x') to compute unnormalised probability p*(x')
8.  select x' with probability min(1,[p*(x')/p(x)]^(1/T))
9.  Cool(T) according to the cooling schedule
```

Experiments with Energy Functions and Initial Temperature

The different energy functions I implemented take the forms as given below:

normal/linear: p*(x) =exp[−f(x)]

cubic: p*(x) =exp[$-f(x)^3$]

squared: p*(x) =exp[$-\text{sign}(f(x)) * f(x)^2$]

arctan: p*(x) =exp[−arctan(f(x))].

The main difference between these different energy functions is in their rate of increase. When f(x) is outside of [1,-1], the cubic function has the fastest rate of increase, followed by the squared, followed by the normal/linear, followed lastly by the arctan. However, inside [1,-1] the rates of increase will typically be reversed. The energy functions can be thought of as determining the relative probabilities of two candidate solutions, the ratio of which will be used when we decide to accept or reject a sample. Faster growing energy functions will lead to more peaked distributions and an increased rejection rate for worse proposed candidates. The acceptance ratio will also be heavily modulated by the initial temperature. As the initial temperature is increased, the value $[p * (x')/p(x)]^{(1/T)}$ tends towards $[p * (x')/p(x)]^0 = 1$, which means we will accept almost any candidate point proposed.

As an example of how the behaviour changes with different initial temperatures for the squared energy function, consider the diagram below. The left hand side shows the exploration path of my simulated annealing algorithm for a very high initial temperature, T_0=1000. The right hand side shows the exploration path with a very low initial temperature, T_0= 0.01. Note the different emphasis each of the algorithms place on either exploration or exploitation.

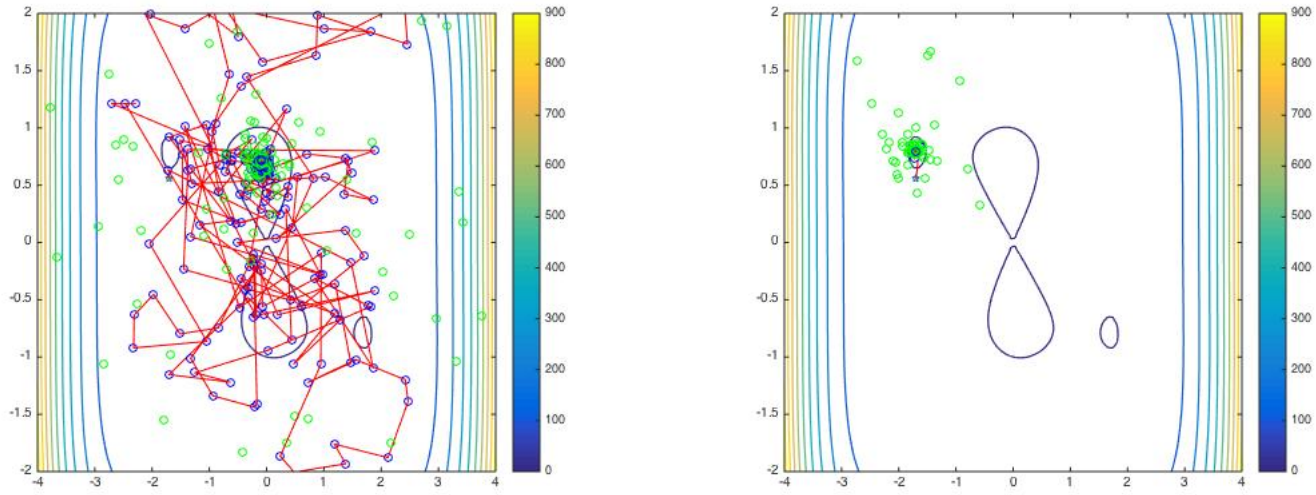


Figure 1: Right: T=1000 and Left: T=0.1 for function call: anneal(1,'gauss', 'squared','geom',0.95,T, 1.05, 0.9,'none').

Below I investigate how the energy functions and the initial temperatures affect the algorithm. To do this I tested my algorithm with initial temperatures from the range {1000,100,50,10,1,0.5,0.1,0.001} against the different energy functions discussed. For each curve produced below, I ran my algorithm 50 times with different random seed values and then averaged over their performance.

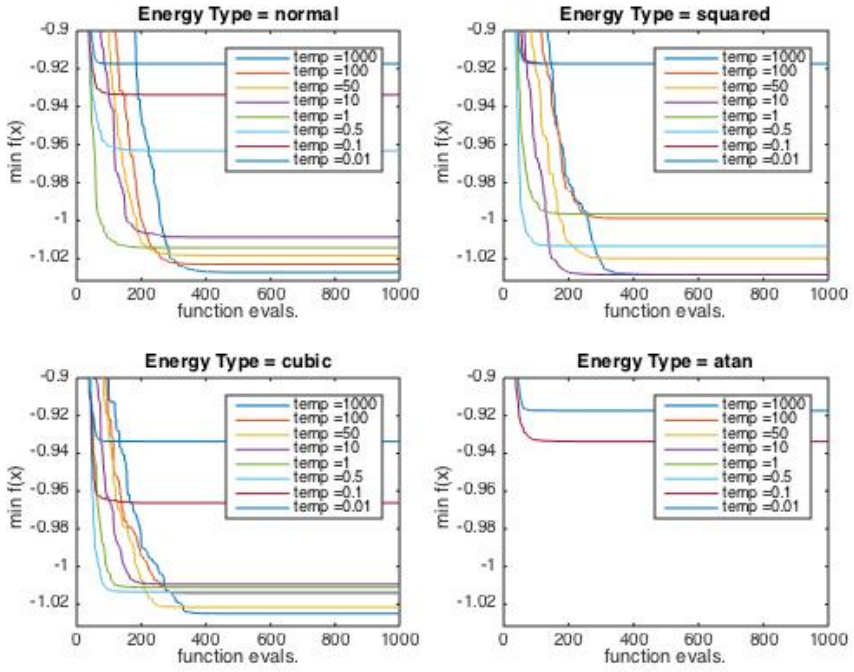


Figure 2: Best f-value found for different energy and temp values against function evals, averaged over 50 runs.

There seems to be a fundamental trade-off: higher temperatures enable more frequent convergence to the global minimum, however their convergence rates tend to be slower. For instance, T=1000 has the best/ joint best convergence for all functions (except for arctan which is insufficiently 'peaked'; see below).

It seems to me that on the whole that the different energy functions are more similar than they are distinct, except that the arctan is not a very good energy function. It does not typically converge to the global minima, probably because arctan grows too slowly and does not allow the probability distribution of candidate points to become sufficiently concentrated. As a consequence this function tends to favour parameter settings with high initial temperatures.

The best parameter settings seem to be squared energy with an initial temperature of 10. These settings reach the lowest average function values, and do so more quickly than other parameter settings that also reach low function values, e.g. (energy = 'squared', T_0= 1000). It also is offers faster convergence than all of the initial temperatures for the normal energy type.

The parameter settings: (energy = 'squared', T_0= 10) will be added to the default parameter settings for my algorithm in future.

I thought it would also be interesting to occasionally plot the average length of the archived samples on termination, as another measure for how well the algorithms explore the sample space. The settings for the archive (d_sim=0.1 and d_min=0.5) were chosen to ensure that an algorithm with good coverage of the search space would gather at least 25 or more samples. As we can see, higher initial temperatures typically mean fuller archives (more coverage), whereas lower initial temperatures mean that less exploration is undertaken and so the archives are smaller in length. I created the archive using variables to store the locations and function values of sample points. The archive is updated using the function 'replace( )' which I implemented and listed in the appendix.

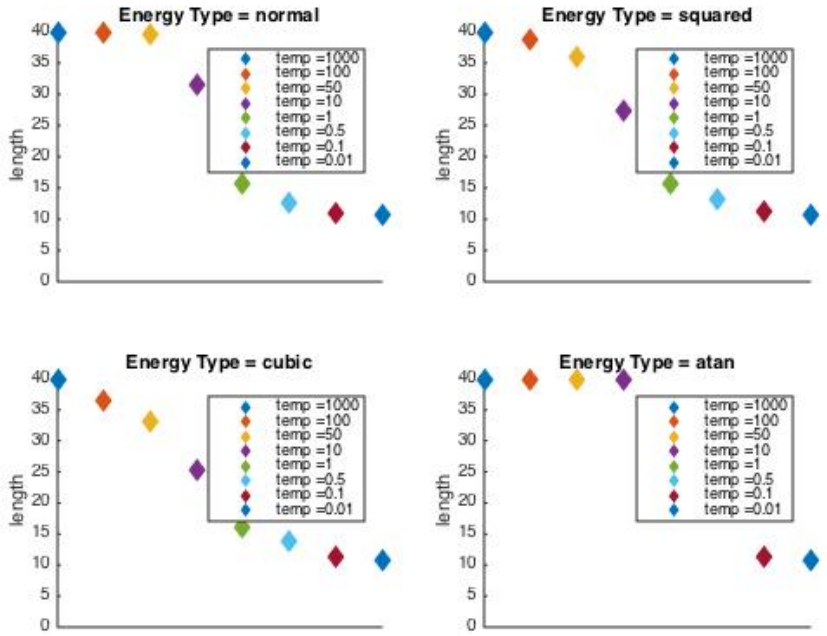


Figure 3: Archive length on termination for different energy and temp settings. Averaged over 50 runs.

Cooling Schedules

The next feature of my algorithm that I shall explore are the cooling schedules, which cool temperatures through time t:

Geometric: $\mathrm{T}_0 \cdot \beta_{\mathrm{c}}^{\mathrm{t}}$

Logarithmic: $\mathrm{T}_0/(2 + \mathrm{t} \cdot \beta_{\mathrm{c}})$

Inverse-linear: $\mathrm{T}_0/\log(\beta_{\mathrm{c}} \cdot \mathrm{t} + \mathrm{e})$

Inverse-exponential: $\mathrm{T}_0 \cdot \exp(1 - \beta_{\mathrm{c}} \cdot \mathrm{t})$

These schedules will be explored in tandem with β_c, which is a parameter that controls the rate of cooling within each function. I varied this parameter β_c with values from the range {0.99,0.97,0.95,0.9,0.7,0.5,0.2,}. Below, I plot the results of my experimentation.

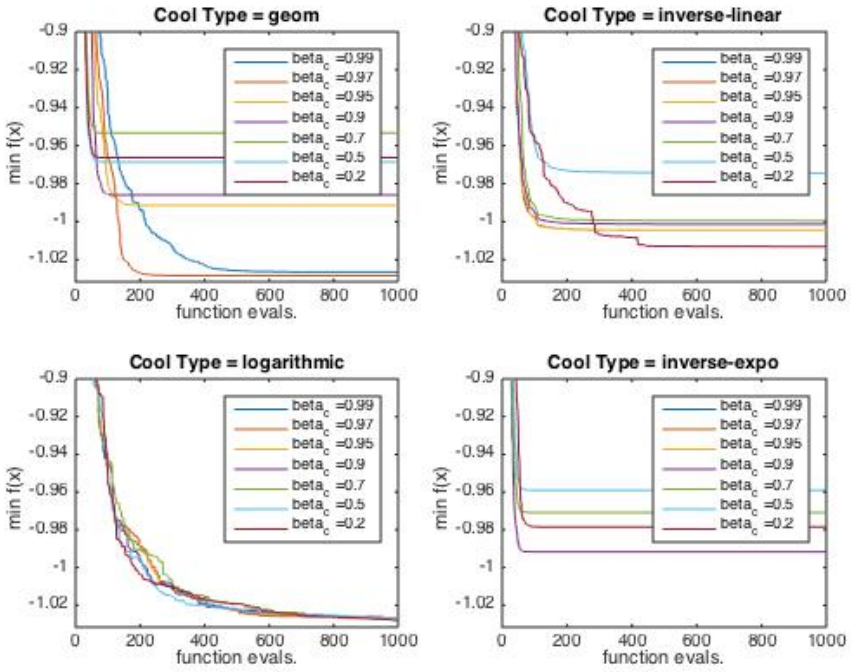


Figure 4: Best f-value found for different cooling and temp. against func. evals. Averaged over 50 runs.

From the graphs above, it is clear to see that the best rates of convergence are offered by the geometric cooling function, with parameter β_c= 0.97. On average this approaches global optimum after just 200 function evaluations.

In general logarithmic cooling is too slow, and is still converging after 1000 iterations. However, having said that: both the logarithmic (and also the geometric cooling with β_c=0.99) have healthier archives, which are completely full with archive lengths of 40 on average, as displayed in the graph below. However, perhaps it could be said that the aforementioned parameter settings do a little too much for the scope of this problem.

The inverse-exponential and inverse-linear functions generally cool too quickly. This can be seen from the sharp rate of descent that their curves initially take. They typically converge between 50 and 100 function evaluations. However, they haven't sufficiently explored the sample space, and on average don't reach the global optimum. This is a fact that is confirmed by the paucity of their archives, which on average have fewer than 20 values.

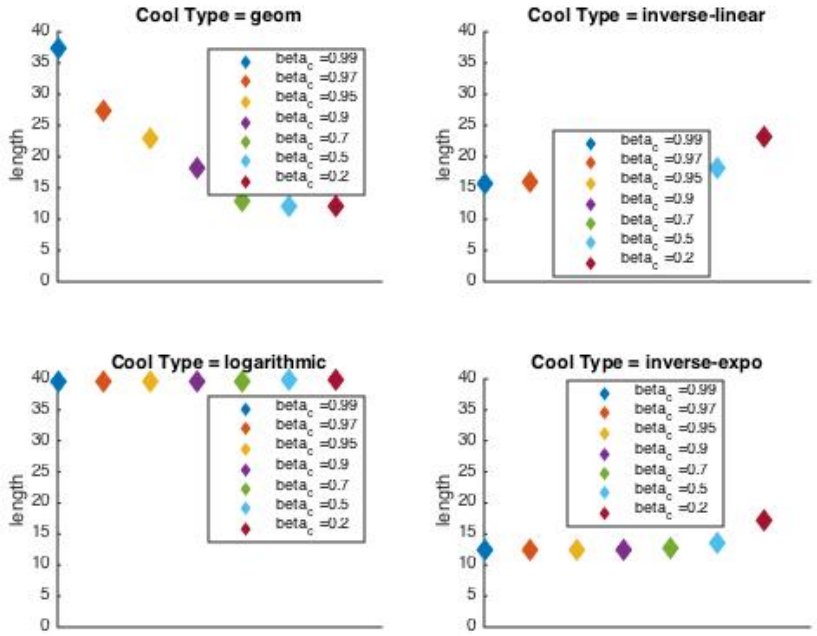


Figure 5: Archive length found for different cooling funcs and params, after 1000 function calls. Averaged over 50 runs

Another interesting feature that could have been explored in connection with cooling schedules is reannealing: raising the temperature after a fixed number of steps. A similar method will be investigated in a later section.

Proposal Distributions

There are a number of ways in which we can vary the proposal distributions. We could draw samples from a Gaussian, uniform or logistic distribution. I implemented each of these methods for my algorithm in the nested function 'propose()' , which is contained in 'anneal()'.

An important point to note that as the temperature is cooled and we approach the minimum, with a fixed variance distribution it becomes increasingly likely that most samples will be rejected. This happens because: (1) far off points are less likely to be better than the current sample as we approach the minimum; (2) as the temperature is cooled the probability of accepting worse solutions decreases. It would be better implementation would be to sample from distributions that had increasingly small variances. I used a 'step-size' called alpha to effectively increase or decreased the variance of the distributions adaptively.

The strategy that I used to adjust the step size was based on the "bold-driver" approach. In that method, every time a mistake is made (a proposal sample rejected), the step size is reduced by a certain factor; and each time a sampled point is accepted, we increase the step size. Typically, the decrease factor, dec, weighs stronger than the increase factor, inc, so that: dec*inc <1. This respects the notion that as we cool temperatures through time, step sizes should decrease. The other reason is that we want to accept as many points as possible to make progress during convergence, and a smaller increase after a successful proposal is likely to converge faster than a large increase.

Varying these parameters, I produced the graphs below. As can be seen, if the decrease factor is too strong i.e. dec $= 0.85$, then often we converge too soon and land in a local minimum. The same effects can be observed if the increase factor is too weak at inc $= 1$. The ideal parameter settings were inc $= 1.1$ and dec $= 0.95$, which for all distributions reach the best function value on average. There is little to no difference between the logistic and the Gaussian distributions, though both marginally outperform a uniform distribution.

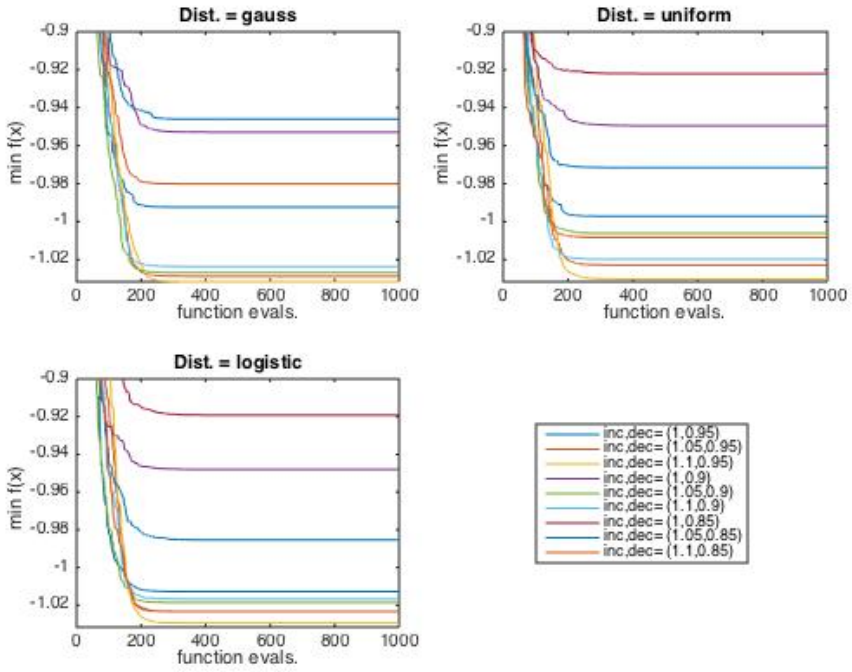


Figure 6: Best f-value found for different proposal functions against func. evals. Averaged over 50 runs.

Below I have plotted a run of the algorithm with the optimal parameter settings. The green circles represent proposed samples that have been rejected. The blue points represent accepted samples. The red lines plot the trajectory of the algorithm. As one can see, the algorithm explores the sample space well- even exploring points close to other local minima. However it does not choose to move to these points, preferring instead to remain closer to the global optimum. For this particular run, it took 228 samples to converge to a global minimum.

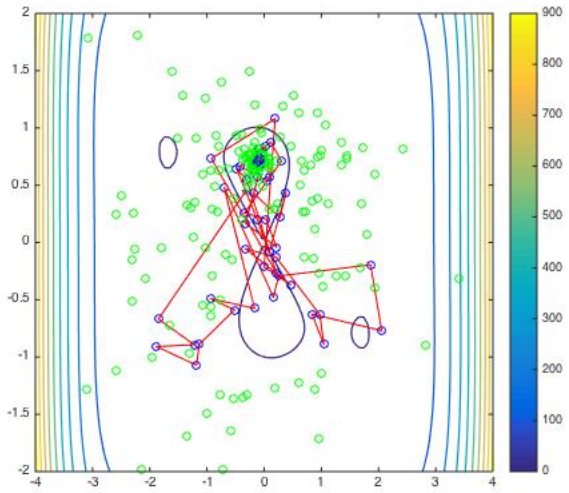


Figure 7: Path of algorithm anneal(1,'gauss', 'squared','geom',0.97,10, 1.1, 0.95,'none'); 228 sampled points to find the optimum.

The parameter settings for the run of the algorithm above represent the best parameter settings found taking the line of experimentation engaged with so far. Having obtained a parameter setting that yields a respectable performance, this is the point where I feel we can take a radically different approach with the algorithms developed so far. In the next section, I will explore such changes.

Adding Restarts, Cooling Quicker

So far we have used a steady cooling temperature to converge to the global minima in reasonable time: typically around 200 points are sampled. Another option is to use a rapid rate of cooling to converge quickly to a local optimum. However, with such a rapid rate of cooling we cannot guarantee that we will converge to a global minima. As such, after a sufficient degree of convergence has been reached, we must restart the algorithm to continue our exploration. There are essentially 2 different kinds of restarts that I have implemented: restarts from minima and restarts from a random location in the feasible space. These methods are implemented within the nested function 'restart( )' in 'anneal( )', which is listed in the appendix.

Random Restarts

The method is as follows: when the algorithm has converged to a minimum with an accuracy of roughly 0.0001 (this is checked for by measuring the distance between successive sampled points), we then force the algorithm to move to a new sampled point randomly located within the feasible space. We also figuratively turn back the clock, setting the time to 0, so that cooling can proceed from the initial temperature. The factor that controls for the variance of the proposal distributions, the step size alpha, is also reinitialised to 1. This means that the algorithm is free to make large jumps through the sample space. Using a faster rate of cooling, the trajectory of the sampled points begins to look like a hill climbing algorithm (with restarts).

Below, I plot the path the algorithm takes through the solution space. On the left hand side, we have the path the algorithm takes until it reaches the global minimum. On the right hand side, we have the path the algorithm takes over 1000 function evaluations. The starred points denote the new restart locations. Just looking at the diagram on the left, it would seem the algorithm has not undertaken much exploration. It moves quickly to one local minima, then after converging, restarts at a random location and thereafter converges to the global minimum. However, this is not the entire picture. On the right, we can clearly see that the algorithm explores the search space well, spending most of its time at 5 of the 6 minima. The algorithm reaches the global optimum after just 121 sampled points.

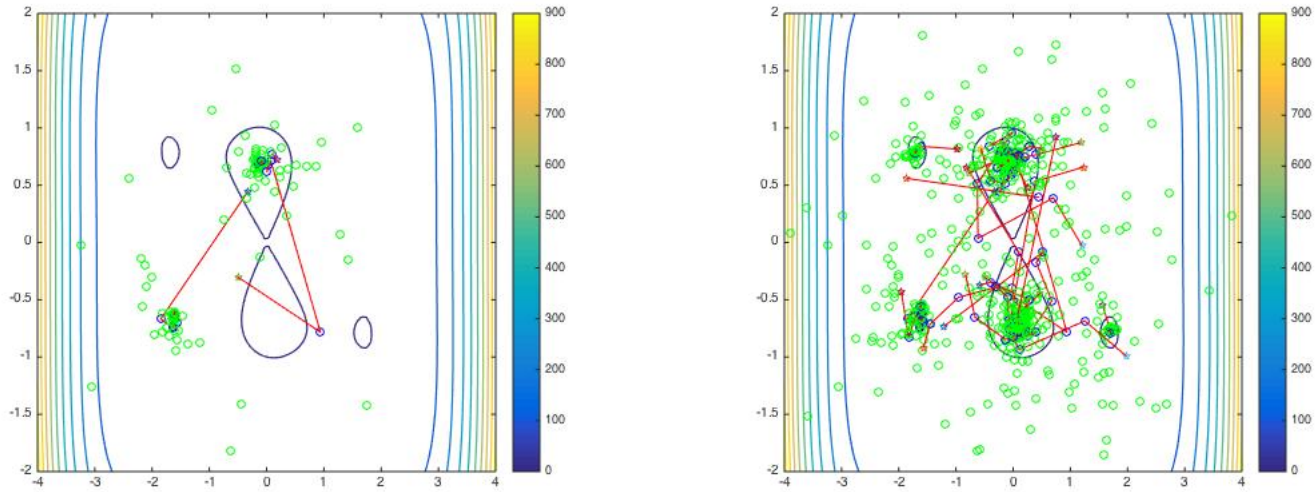
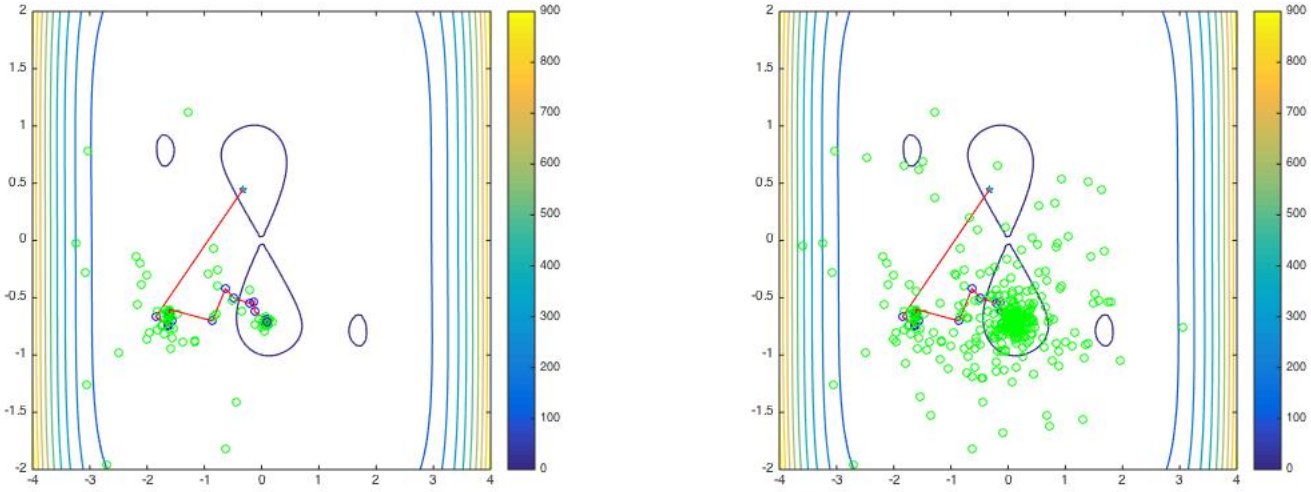


Figure 8: The path of the algorithm with parameters: anneal(1,'gauss', 'squared','geom',0.2,10, 1.05, 0.85,'random'); Left: on reaching the global optimum; Right: after 1000 evaluations.

Restarts from Minima

The second kind of restart that I implemented is closely related to the 'reannealing' process that was mentioned earlier, which involves raising the initial temperatures periodically, but by a smaller factor each time. The difference with my method is that I only raise the temperatures when there is convergence to the minima, and not periodically. Each time we reach a successive minima, we raise the temperature to the initial temperature multiplied by a decrease factor: $T_0 \cdot \text{decrease}^n$, where n is the number of restarts. The time is also reset to 0 and the step size alpha is reset in the same way as for the 'random restarts' described previously. On the left, I plot the path the algorithm takes up until is reaches the global minimum. On the right, I plot the algorithm for the full duration of the run (1000 function evaluations). The algorithm does rather less exploration than the previous random restart method, presumably because escaping from a minimum can be challenging once one has ended up there. The algorithm took 164 sampled points to reach the global minimum and then remain there.

The advantage that this method has over 'random restarts' is that once the global minimum has been found, the algorithm is much more likely to remain there, since escaping the global minimum is difficult and becomes ever more so with consecutive restarts. (This is due to the decreased initial temperature as discussed, selected in my code to be 0.7).



If let run around

Figure 9: The path of the algorithm with parameters: anneal(1,'gauss', 'squared','geom',0.2,10, 1.05, 0.85,'minima'); Left: on reaching the global optimum; Right: after 1000 evaluations.

A more detailed picture of how the algorithms perform is given by the graphs below. Both algorithms always reach the global minimum, with random restarts tending to do so more quickly. The algorithms favour a temperature cooling factor β_c of 0.2 rather than 0.4- for very rapid convergence. Both algorithms perform well with a wide variety of increase and decrease factors for the step size alpha, which controls the variance of the proposed samples.

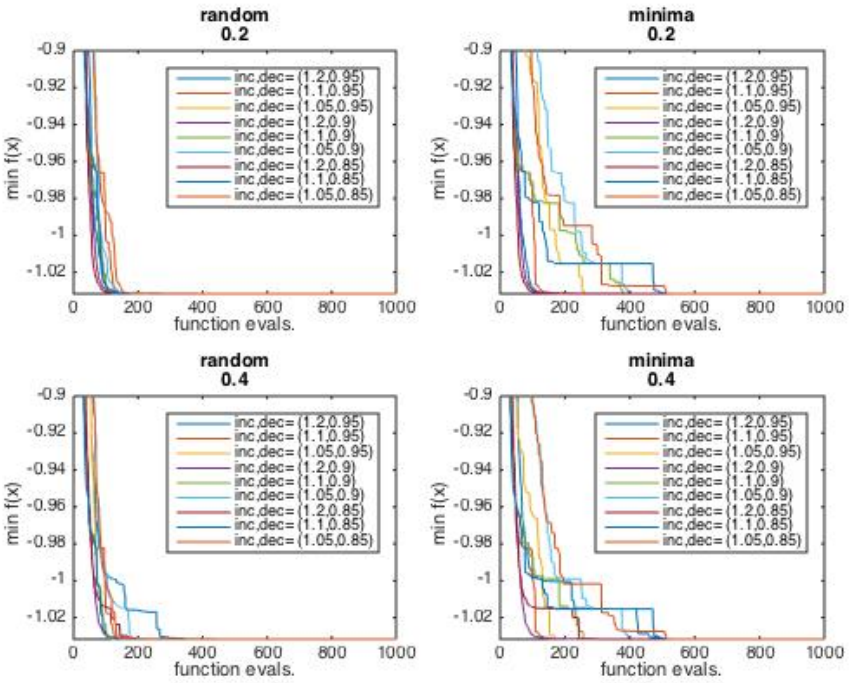


Figure 10: Best f-value found for random/minima restarts with cooling rate 0.2/0.4. against func. evals. Averaged over 50 runs.

Conclusion

In this report, I explored a number of implementation methods for simulated annealing, based on the links with metropolis sampling. After writing a basic simulated annealing algorithm, I implemented methods to vary the proposal distributions, cooling schedules and energy functions. I then implemented an adaptive step size to control the variance of the proposal distributions. After conducting these initial investigations an optimal parameter setting of (proposal='gauss', energy= 'squared',cooling ='geom', β_c=0.97, T_0 = 10,inc=1.05, dec=0.95) was found, with convergence obtained after 200 function evaluations. Having taken one line of investigation, an alternative path was considered. I implemented methods to enable restarts, both from minima and to a random location. These methods allowed for faster rates of cooling and convergence, but only guaranteed convergence to within a fixed limit (0.0001). The optimal parameter settings discovered for this section were: (proposal='gauss', energy= 'squared', cooling ='geom', β_c=0.2,T_0=10, inc=1.05, dec=0.85, restart='random'), with convergence obtained just after 100 function evaluation.

Evolution Strategy

The Algorithm

The initial algorithm I implemented for evolution strategies was based on the procedure described in the 4M17 lecture notes. In the main body of this algorithm the following procedures were carried out.

1. Select mu parents
2. Mutate parents' strategy parameters
3. Mutate parents
4. Recombine Mutated parents to create children
5. Assess new population
6. Update archive

I made a number of changes to the procedures outlined above while investigating the different parameter settings. As a result, the algorithm I have listed in the appendix is slightly different from the above, since the code in the appendix displays the final outcome. However, the above was the procedure I used at the start of the experimentation before modifications were made. It would be useful to describe how some of the elements of the procedures above were computed within my program 'es2( )', before discussing how I adapted the algorithm.

(1) Selection is performed by simply selecting the best μ parents from λ offspring. This is computed within section 1. of the main body of 'es2'. (2) To assess the fitness of the population, I created a function 'fit( )' which is nested within 'es2'. 'Fit( )' sums the 6 hump camelback function 'camel( )' with a penalty term. The penalty term penalises out of bounds variables to ensure convergence. It penalises out of bounds y values twice as much as it penalises the x values since the bound for x is twice that of y. As time goes by the penalty for out of bounds variables becomes harsher and harsher, as a result of exponentiating to the power of t. (3) Initially, I mutated parents according to the method described in the lecture notes, where there is seems to be a single set of strategy parameters for the entire population. These strategy parameters comprised a single rotation angle and variances in x and y coordinates. the strategy parameters were mutated using parameters tau, taup and beta as time passes. However, I later altered this method, and the changes will be described in a section below. The mutations are handled in section 4. of the main body of 'es2( )'. (4)

To perform recombination I implemented a function 'recombine( )', listed in the appendix, which can recombine according to global, bivariate, intermediate, or discrete methods. Recombination is performed within section 3. of the main loop. (5) Assessment is performed at the end of the loop, and uses the penalised method of determining fitness as described previously. (6) There are 2 variables within 'es2( )' xs_arch and fs_arch, which contain the [x,y] variables and the function evaluations at those variables for the members of the archive. The archive is updated using a function called replace( ), which I implemented. By default, the similarity distance (threshold for points to be considered as close) and d_min (threshold for points to be considered as far) are set to 0.1 and 0.5 respectively. I discovered that with such values, an archive size of 25 or higher would ensure that an algorithm had good coverage over the feasible space.

As the submission required writing over 500 lines of code and the two implementations were rather similar, I decided to omit the former implementation and just detail the changes made.

Experimentation: Mu-lambda Variation

The most important parameter settings for evolution strategies are the parent and child population sizes: μ and λ. These parameters determine the population size at any given time instant. And all of the other features of the algorithm involve recombining, mutating or selecting from these populations. As such, varying these parameters seems like a natural place to start. The population parameters also determine how selective the algorithms are. More selective parameter settings, for instance mu=1 and lambda =20 will do more exploitation than exploration, however less selective parameter settings, for instance 7:14 will do more exploration- and this should be visible in a graph.

Below, I plot the graphs showing my exploration of the different parameters. The graphs below were produced by averaging the best function value found by the algorithm against function evaluations over 50 seeded runs.

There are a number of facts that are immediately apparent when looking at the graph: single parent populations tend not to do well, presumably because there is no selective pressure improving generations. The algorithms also tend to do better with increased selectivity, at 1:10- whereas Schwefel recommends a selectivity ($\mu : \lambda$) of 1:7. However, another more distressing issue is also apparent. Many of these parameter settings are not converging to the global minima at f(x) = -1.03. To converge, the curves should lie on top of the horizontal black line at -1.03, however there is a good amount of white space between al the curves and this black line.

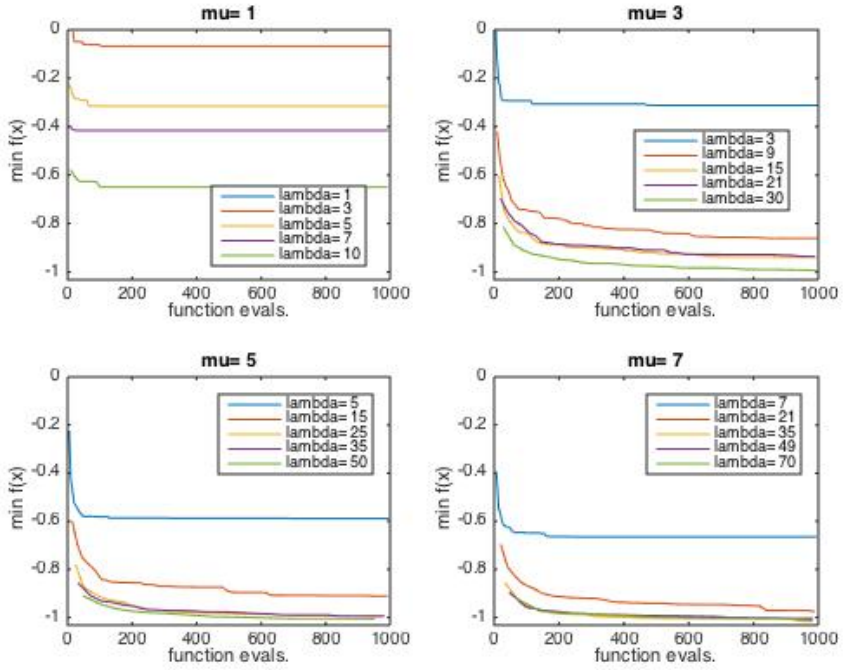


Figure 11: Best f-value found for μ and λ against func. evals. Averaged over 50 runs.

After some investigation, I discovered why the curves were not approaching the global minimum. Below, I have included a series of 4 graphs that show the points sampled by the algorithm through time, with population sizes $\mu = 5$, $\lambda = 15$. On the 200th function call, the sampled points are still widely distributed across the sample space, and only by the 950th function call do the points seem to have converged to a single region. However they have only done so momentarily and after just 5 more iterations they will soon be spread apart.

Before we continue, we should take a deeper look into how the variances are mutated. They are mutated according to the formula: $\sigma_i' = \sigma_i' exp(\tau' \cdot \aleph_0 + \tau \cdot \aleph_i)$, for each standard deviation σ_i. τ and τ' are mutation constants that are as set as recommended by Schwefel: $\tau = \frac{1}{\sqrt{2\sqrt{2}}}$ $\tau' = \frac{1}{\sqrt{2}}$; $\aleph_0$ and $\aleph_i$ are sampled from a standard normal. In our case, we only use 2 standard deviations σ_1 and σ_2, since the problem is defined in 2 dimensions.

The standard deviations σ_i of the population don't converge sufficiently well. The σ_i will have converged momentarily at the 950th iteration above, but due to mutations [of the form $\sigma_i' = \sigma_i exp(\tau' \cdot \aleph_0 + \tau \cdot \aleph_i)$] , the σ_i will grow larger again. Essentially all that happens is the standard deviations balloon upwards and downwards subject to the forces of random sampling. Mutations from these σ are ultimately added to x, and since the σ don't decrease sufficiently, x doesn't converge to the global optimum.

The main reason for this problem is that the standard deviations are generated for the whole population, and so there is no selective pressure placed on the σ to help improve their values (i.e. become more suited to convergence to the global minimum).

In the next section, I will explore changes that can be made to the algorithm to bring the forces of selective pressure to bear on these standard deviations.

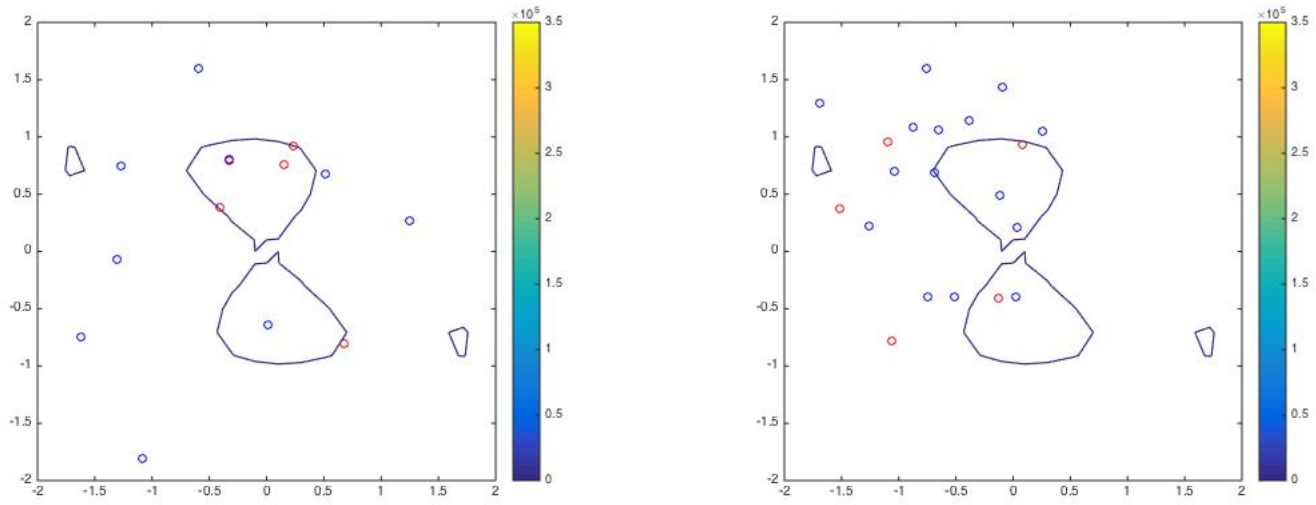


Figure 12: A (μ =5,λ =15) population at left; t=1, right: t=50; time measured by function evals.

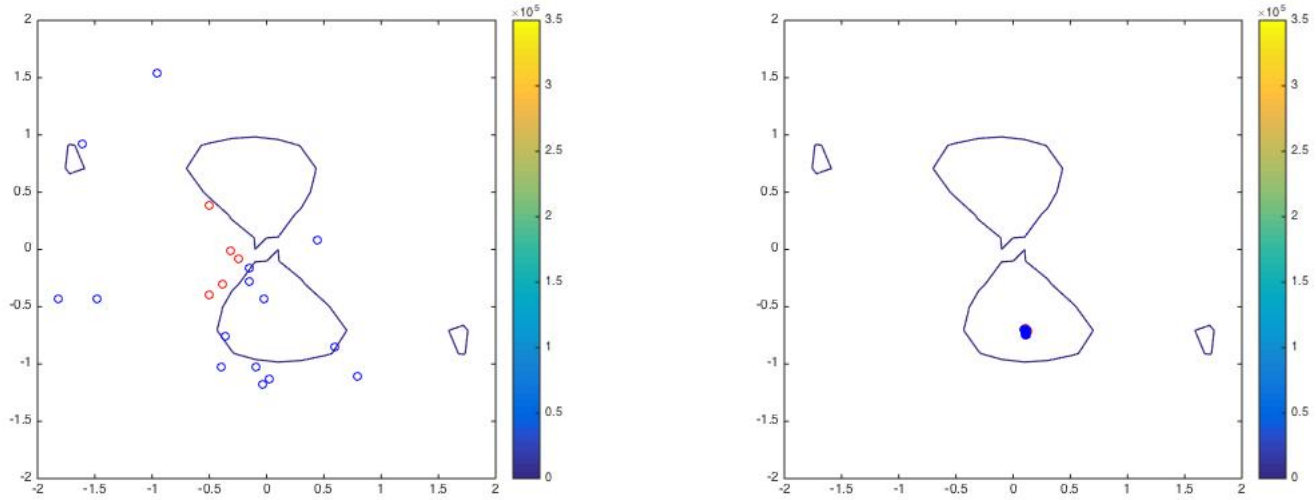


Figure 13: A (μ =5,λ =15) population; left: t=200, right: t=950; time measured by function evals.

Altering Mutation Parameters

The main changes I made to the original algorithm described in the lectures are as follows:

(1) I gave each population member its own variance. To have selective pressures we first need to select from a population of σ, and not a single σ.

(2) I simplified the standard deviation mutation to $\sigma_i' = \sigma_i exp(\tau \cdot \aleph_i)$, which mutates each dimensions of σ_i separately, rather than adding a bias. This change was intended to make the effects of a single mutation on a sampled x more disentangled from others, and thus to hopefully improve selective pressures on mutations to σ through simplicity.

(3) I mutated the children using σ rather than the parents. This would improve selective pressure by increasing selectivity. If parents passed down their mutations to their children, there would be fewer mutations to select from. The intuition is that by having the children induce their own mutations, we have lambda > mu mutations to 'select' from.

(4) I decided to mutate the children after recombination, rather than the other way around. To maximise the selective pressure on the σ we want to tie them as closely as possible to the selection process. This can only be done by tying them more closely to the variables they mutate, which are the only objects directly selected for fitness. As a result, I had to ensure that mutations controlled by σ were the last changes made to the sampled points before they were assessed for fitness and selected- otherwise the effects of a mutation on a variable would be garbled by recombination and other such procedures. And this would then decrease the selective pressure on the σ.

Ultimately, these 4 changes are all designed for the same purpose: to apply selective pressures to the standard-deviations of the sampled point mutations σ, so that as we approach the global minima, convergence will be possible. NB: There are also included rotation angles in the algorithm, however I have decided to focus less attention on them, since convergence is less dependent on them.

Returning to the original problem, aside from introducing selectivity to the standard deviations we could have also gradually cooled the standard deviation and standard deviation mutations as time passes by. This is a strategy reminiscent of simulated annealing, where temperatures are gradually decreased, and will be discussed later.

Below I have plotted graphs that test how the algorithm with modifications (1-4) as described performs. The same parameter settings as that of the old algorithm are tested. What can be seen is a large improvement in results. All parameter settings, except for those with an equal number of parents and children (which is not a very selective setting) converge quite closely to the global minimum. This can be seen since all curves reach the lowest horizontal black line which marks the global minimum functional value.

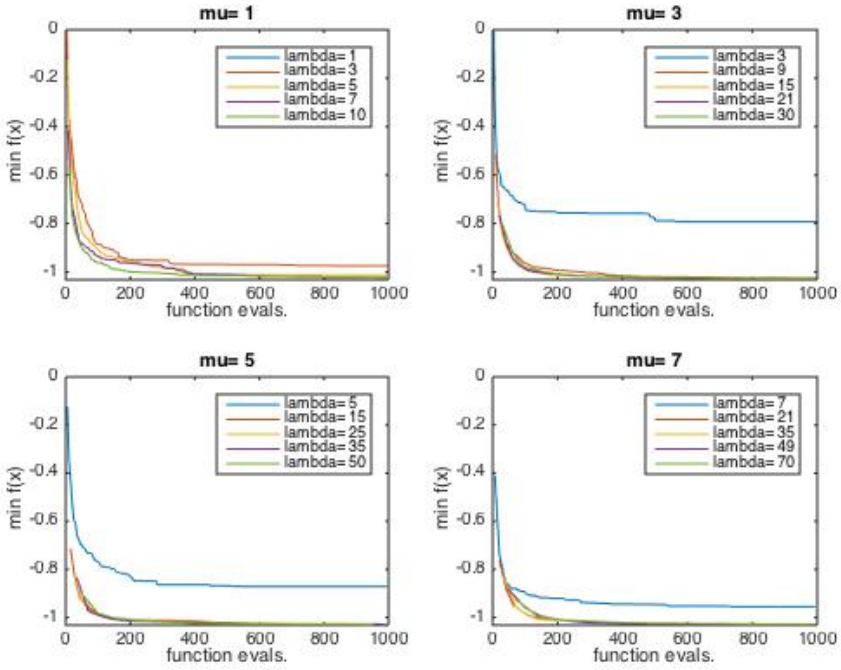


Figure 14: Best f-value found for μ and λ against func. evals. Averaged over 50 runs.

From these new graphs, I concluded that the best combination of parents was mu= 5 and lambda = 15, contrary to Schwefel's recommendation of 7 children per child. This seems to offer one of the fastest rate of convergence to the global minimum. However, many of the parameter settings seem to be good. As a matter of interest, I have also plotted the average length of the archives produced on termination, as a proxy measure for the algorithm's coverage. As we can see, the archives tend to be quite large, and provide evidence for a good range of coverage. The default parameter values for the next sections will be updated to mu=5 and lambda=15.

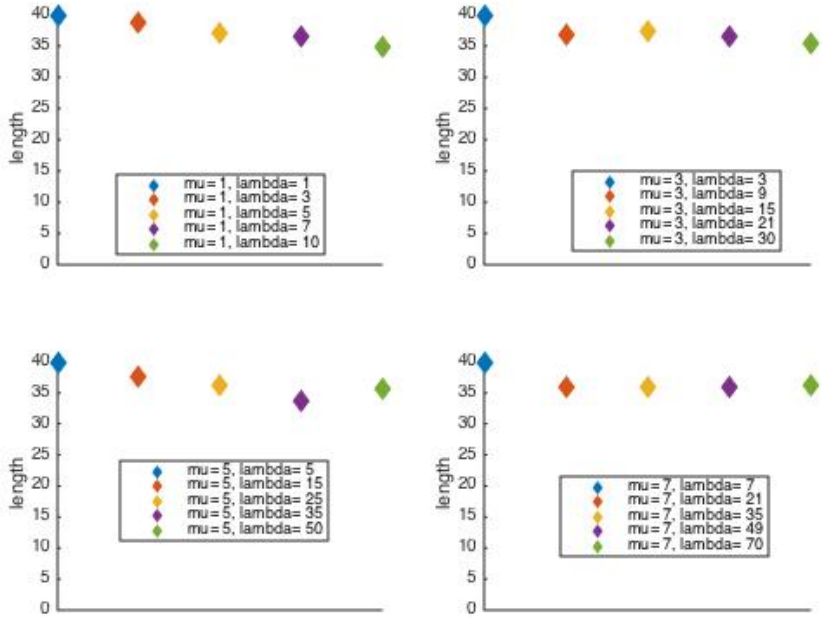


Figure 15: Archive length found for μ and λ after 1000 function calls. Averaged over 50 runs.

Recombination Techniques

The next stage my algorithm involved testing four different kinds of recombination technique for both the strategy parameters (σ, α) and also the control or sampled points $\mathbf{x}$. I implemented this in the function 'recombine( )', listed in the appendix.

Recombination is the process of creating a new 'child' vector by recombining features from 'parent' vectors. The child is typically built feature by feature. A note on my definitions: I have defined 'bivariate' recombination as the process of recombination where the parents used to create a new child must be the same for every feature. The definitions of global, discrete recombination and intermediate recombination are as described in the lecture notes for evolution strategies. Below I plot performance graphs for the various types of recombination. I have recombined the strategy variables and control variables independently. In the graphs below, 'gi' stands for 'global intermediate', 'gd' stands for 'global discrete', 'bd' stands for bivariate discrete' and 'bi' 'bivariate intermediate'.

From these graphs it can be seen that the bivariate recombination for the strategy parameters performs pretty poorly, and the algorithm seems to favour global recombination which would give rise to a more diverse set of vectors (as explained in the note below).

Both global-discrete recombination and global-intermediate recombination work well as methods to recombine the strategy parameters. A combination (strategy, control) = ('gd','gd'), seems to offer the best convergence, and is rather better than a combination of (strategy, control) = ('gi','gi'). Perhaps this is because global intermediate recombination involves a greater degree of exploration in the parameter space, with the extra exploration resulting in less convergence near the minima. I decided to use the (strategy, control) = ('gd','gd') parameter setting as the default values for further exploration.

(NB: global recombination involves more exploration because the linear interpolation between two vectors typically results in values for the elements of a vector that have hitherto been unseen. This is not the case for discrete recombination, which only reproduces elements of the parent vectors.)

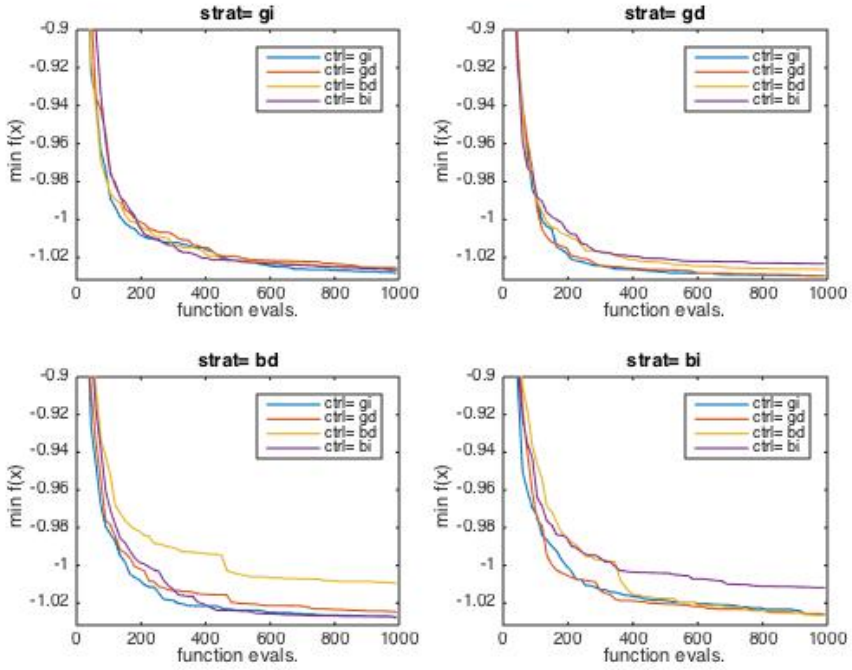


Figure 16: Best f-value found for recombination type against func. evals. Averaged over 50 runs.

From Simulated Annealing: Noise and Cooling

Having explored simulated annealing in the previous part of this coursework, I was struck by an interesting idea: we could improve convergence in an evolution strategy, by cooling the standard deviations of the population parameters, just as 'temperatures' are cooled in simulated annealing to improve convergence. The cooling rate is a simple multiplicative factor applied at each time step (measured in function evaluations) to both the standard deviations σ and the parameter τ which controls for the mutations in standard deviations: $\sigma_i' = \sigma_i' exp(\tau \cdot \aleph_i)$. Furthermore, I also decided to vary what can be thought of as an 'initial temperature' by multiplying the initial values for σ by a constant factor, sigmult. Below I graph the results I obtained for various different 'initial temperatures' sigmult and cooling rates.

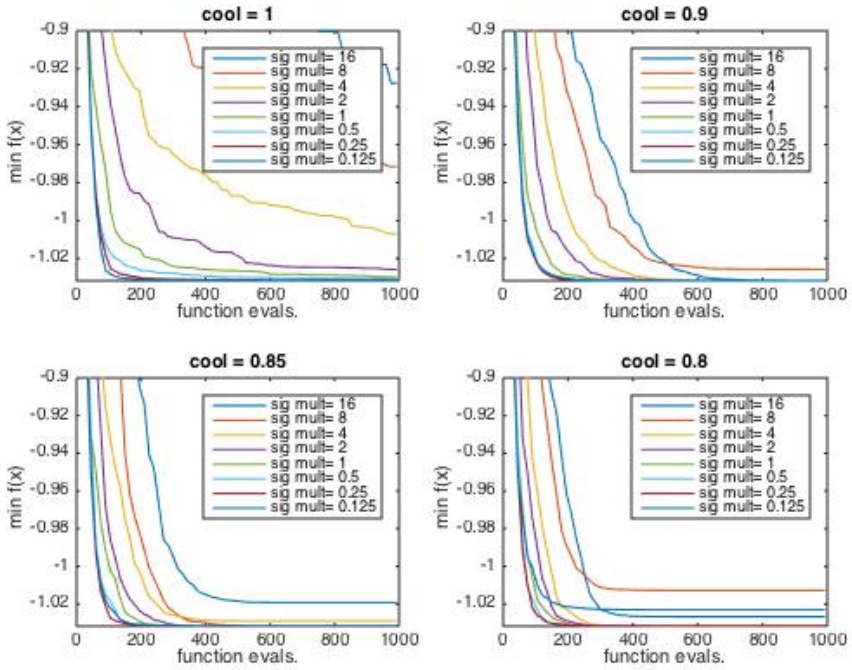


Figure 17: Best f-value found for (cool, temp) against func. evals. Averaged over 50 runs.

There are a number of good cooling rates: both 0.85 and 0.9 seem to offer quick convergence to the global minimum for most initial temperatures. Raising the temperature typically does not help us converge to the global minimum. However, if the global minimum were located further away, then a higher initial temperature would prove to be useful. With an initial temperature of 1, and a cooling rate of 0.85, it takes just under 200 steps to converge to the global minimum. Compare this with the graph plotted in the previous section, where none of the curves completely converges (within my eye's resolution) for even 1000 steps. (NB: I had increased the resolution of the graph in the previous section and also this section, so that better comparisons could be made as the algorithms had improved.)

Now would be a good time to explore the average length of archives on termination, to see how well our algorithms truly explore the search space. Very low initial temperatures mean we do not conduct enough exploration, as expected. High 'temperatures' ensure that we are left with fuller archives. However, it should be noted that larger archives do not necessarily result in better quality solutions.

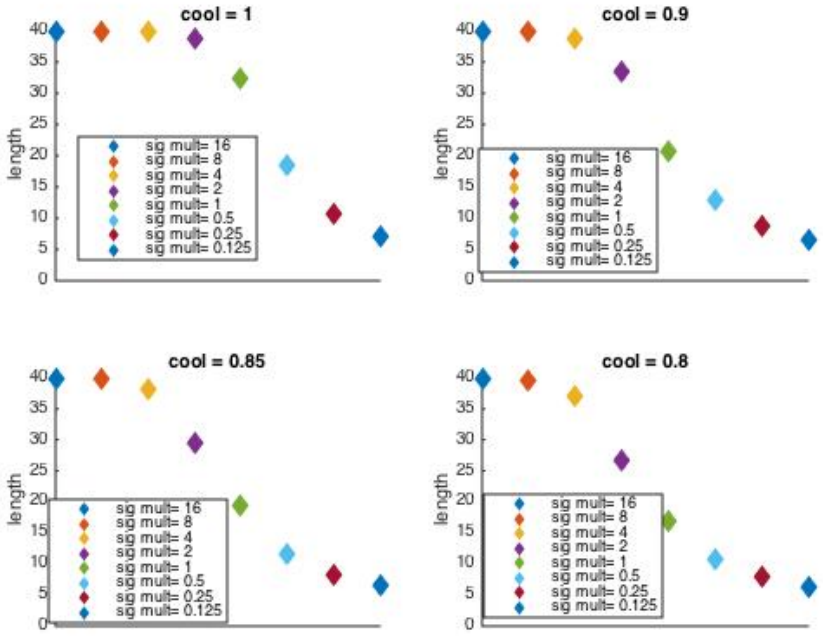


Figure 18: Archive length found for different cool and sig mult after 1000 function calls. Averaged over 50 runs.

I will now choose a cooling rate of 0.9 and a sigmult of 1, to demonstrate the evolution of the algorithm. From the series of graphs we can see that the points sampled have a large dispersion initially. After 50 function evaluations, the points have converged to the two global optima. After only 200 evolutions, all the points are focused on a single optimum. Well before the 1000th function call, the points have converged to a single point: a global minima.

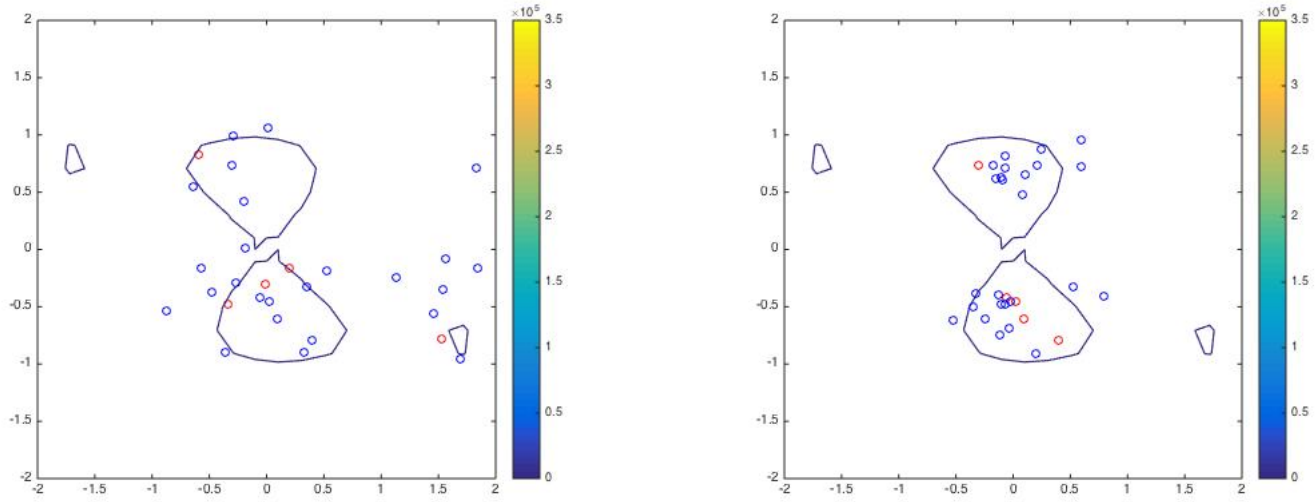


Figure 19: A (μ =5,λ =15,ctrl='gd,strat='gd', cool=0.9, sig_mult=1) population at left: t=1, right: t=50

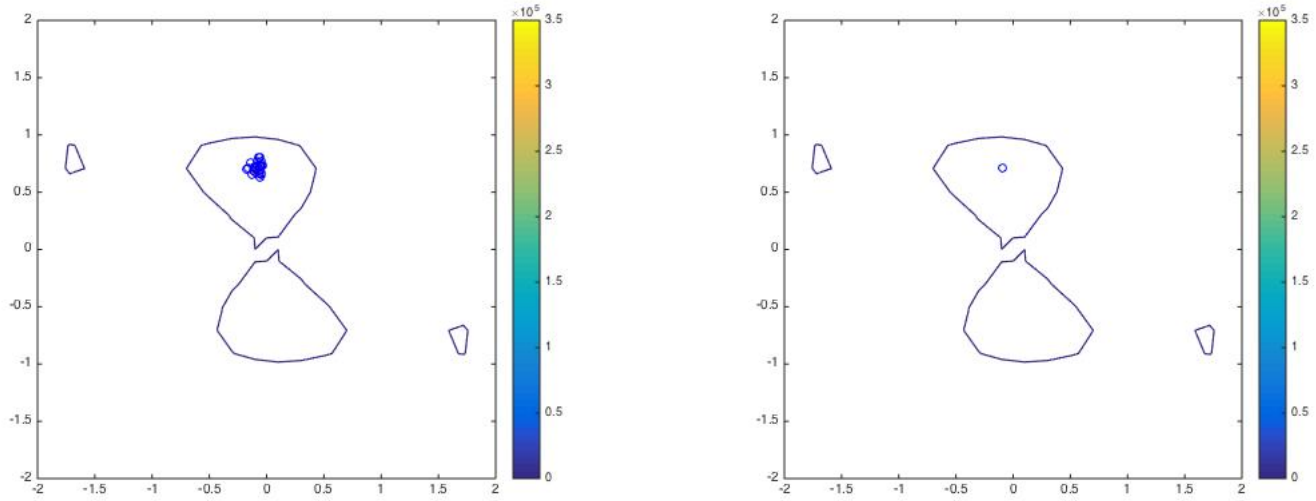


Figure 20: A (μ =5,λ =15,ctrl='gd, strat='gd', cool=0.9, sig_mult=1) population at left: t=200, right: t=1000

Conclusions

I explored a number of parameter settings based on the initial description of evolution strategies given in the 4M17 course. After implementing the evolutionary strategy method discussed, I investigated different population sizes, recombination techniques and also modified the manner in which mutations are performed by giving each member of the population its own standard deviations. This was intended to improve convergence through an 'evolutionary' selection processes. Finally, I looked at whether cooling standard deviations would result in improved convergence, drawing from an analogy with simulated annealing. The changes made resulted in improvements: some marked, others less so. The optimal parameter settings that I discovered were: (μ =5,λ =15, ctrl='gd, strat='gd', cool=0.9, sigmult=1) which obtained convergence in roughly 200 function evaluations.

References

Barber, Bayesian Reasoning and Machine learning: Section on Metropolis Hastings

Schwefel et al, Evolution Strategies a Comprehensive Introduction (2002)

4M17 Lecture Notes on Evolution Strategies

Wikipedia article on Global Optimisation