LUND UNIVERSITY

Combustion Simulations with reduced mechanisms using hybrid optimization

Marius Burman

Thesis submitted for the degree of Bachelor of Science

Project duration: 4 months

Supervised by Elna Heimdal Nilsson and Christoffer Pichler

Department of Physics

Division of Combustion Physics

May 2018

Contents

Abstract

In order to simulate complex combustion systems, the kinetic mechanisms describing the chemical processes have to be reduced. To minimize the error introduced by the reduction, coefficients of the reaction rates included in the mechanisms can be adjusted in ways so that simulations using the reduced mechanism behave like simulations that use detailed mechanisms. One can then run simulations with different sets of reaction coefficients and compare the result to that of a detailed mechanism thus knowing if the new set of coefficients was better than the old or not. This can be done using an optimization algorithm that in smart ways picks sets of coefficients and uses them in simulations. In this project, simulated annealing and a genetic algorithm is used together to create a *hybrid algorithm* in order to mitigate each other's weaknesses to ultimately find better coefficients.

1 Introduction

Combustion of fuel is currently the most used way of producing energy[2] despite its negative effects on the environment such as air pollution and global warming. The lack of ways to conveniently produce energy immediately as effective as combustion of fuel ensures that combustion is here to stay, at least for the coming future. It is thus not only important to optimize currently used engines and fuel to have as little impact on the environment as possible, but also to develop new types of fuel as we transition from using fossil fuel.

As of late, a very good way of studying combustion processes is to use computer simulations. They allow for studies of properties of combustion that otherwise can be very challenging or even impossible to study. Not only that, but it can also be very cost effective, and highly adaptable for whatever reaction one wants to study. For simpler combustion systems, one can use very detailed chemical mechanisms for the simulations, which gives accurate results. However, more complex systems like engines and gas turbines with a lot of turbulence require significantly more calculations and computational resources. Simulations of these systems with fully detailed mechanisms would then be very tedious and time-consuming. Instead, the detailed mechanisms are *reduced*, meaning that less important species and reactions of the mechanisms are removed. It is then practical again to simulate complex systems even though the accuracy of the results is reduced slightly.

In order to decrease some of the error introduced by the reduction of mechanisms, one can change the coefficients for the rates of reactions in the reduced mechanisms. For good choices of coefficients, the reduced mechanisms will produce similar results as the detailed mechanisms but for less computational resources. One can pick these coefficients by using an optimization algorithm, which in smart ways depending on the algorithm, tests different values of the coefficients until a good value is found. This is done by letting the algorithm pick a set of coefficients, using them in a simulation and comparing the results to the results from a simulation with the detailed mechanism. This way, the algorithm knows if it is picking better or worse coefficients for every iteration, and can in that way find a good set quicker. For this project, these coefficients will be optimized using a hybrid algorithm consisting of two algorithms that when working together, can find an even better set of coefficients. The algorithms that will be used are *simulated annealing* and a *genetic algorithm*. The weakness of one of these algorithms is often the strength of the other, and so the intention is that they will mitigate each others weaknesses and amplify each others strengths. The hybrid algorithm will then be tested on a set of test parameters and a test functions to ultimately be implemented in real simulations to find better coefficients than is being currently used.

2 Background

2.1 Combustion Physics

Despite the environmental consequences of using fossil and biofuel, over 90% of the worlds energy comes from combustion of these fuels[7], and it does not seem like it will change for a while. It is currently an energy source that is easily accessible, convenient, and even cost effective. Because of the large energy density of standard fuel used in combustion, its also a great tool for vehicles of all forms, and is used accordingly. Renewable energy sources such as solar and wind power are often unreliable, which also increases the relevancy of fuel for combustion as it serves as a mean for backup. It is therefore important that devices using combustion and its fuel are thoroughly studied, as improvement of even a few percent can lead to a big decrease of pollution[7]. In the last few decades, combustion has also been used to extract energy from renewable sources such as ethanol and biodiesel. It has lately been of high interest as especially ethanol as a biofuel was seemingly easy to merge with the existing infrastructure and relatively low cost of production due to the existing alcohol industry[5]. Of course, fuels from renewable sources are going to be exceedingly more important as the sources of fossil fuel are depleted, and increased knowledge of combustion of these fuels can improve many aspects of the fuel for the future. Aside from transportation and energy production, combustion can also be used for other purposes, such as waste incineration, applications in industrial processes, safety and transforming soot which can have many negative effects on environment and human health[2]. Overall, combustion is very frequently used all over the world and has a large impact on society and the environment meaning there is a lot of reasons to study and optimize combustion processes.

Combustion is an exothermic chemical reaction between a fuel and an oxidant, where exothermic means that energy is released in the process. The fuel can be a wide variety of things, but in general it is a substance with a lot of bound energy. When the fuel is introduced to an oxidant(and often high temperature) they react to form many new substances with a lower amount of bound energy. The difference in bound energy from before and after is then released and can be used in many different ways, for example to power engines [16]. An oxidizer can be any substance that causes other substances to lose electrons, but commonly oxygen and hydrogen peroxide are used. Describing a combustion of a specific fuel can be a very hard task, as even combustion of simple fuel results in a large amount of reactions and species. A description of these reaction and species for a fuel is often called a *mechanism*, and working out fully detailed mechanisms is a challenge that is currently being worked on for many fuels. The amount of species and reactions that a mechanism contains can differ widely between mechanisms. Some fuels can easily be described by tens of species and reactions while more complex fuels like hydrocarbon fuel that is described by a n-heptane-air mechanism contains 561 species and 2539 reactions and it is still growing[3]. The reaction rates of the reactions in the mechanisms can be described by the *Arrhenius equation*. It accounts for the rate with which molecules collide, their orientation, the activation energy for the reactions to occur, and the temperature of the system. The rate of the reactions is then

$$k = Ae^{\frac{-E_a}{RT}} \tag{1}$$

where E_a is the activation energy, R is the universal gas constant, T is the temperature and A is the rate constant.

For long, these mechanisms have been studied using experimental methods including open engines, laser diagnostics and spectroscopy. However, lately computer simulations have opened an entire new world with a lot of opportunities to study combustion in an entirely new light.

2.2 Simulations of combustions

Realistic simulations of combustion are known to be very computationally demanding, and for a long time have been dismissed because of that. The reason they are so computationally heavy, is that a lot more than just the chemistry is simulated. Things like dynamics of the gas, turbulence, flames, convection and so on have to be accounted for, which makes simulation significantly harder to do. In the last 3 decades however, simulations of combustion have become reality with the fast and steady improvement of computers. The strengths of these simulations lie in both the vast

amounts of application, but also how one can tune the software for specific studies. Computer simulations also open the possibility to study processes that are nearly impossible to study otherwise, such as pressure fluctuations, vorticity and dilatation[3]. Of course, the great convenience of studying these things does not come for free. One always has to rely on that the simulations are accurate and have physical meaning, which can be rather hard to prove. Especially when one always has to choose between computational time and accuracy. A realistic simulation of a combustion process that includes all parts of the combustion with high precision, would take many months to complete, which kind of defeats the purpose of simulations. Therefore, approximations have to be made, but in ways that do not affect the credibility of the simulation too much.

There are also approximations to be made outside of the reactions. Simulations in 3 dimensions are often too computationally heavy, and so most simulations are in 2, 1 or even 0 dimensions[8]. However, sometimes it suffices with low dimension for simple simulations, but as more variables are added to the simulation - like turbulence, more dimensions are required to describe the system. The number of dimensions used is thus very project specific. The choice of flame for the combustion also plays a big role in the simulations. There are many types and mixtures of flames, but the two main categories are *laminar flame* and *turbulent flame*. Low rate of gas flow often causes a laminar flame, which is relatively "calm" and simple, with low flame velocity. Adverse to that, turbulent flames are often the result of a higher gas flow. They cause a lot of turbulence in the combustion, which can increase and complicate the reactions greatly and so a laminar flame is usually used for simulations to decrease the computation time[10].

One of the reasons that these simulations are so complicated is that reactions often are significantly more complex than one would think. For example, a reaction where hydrogen and oxygen are converted into water, and vice versa, can be written as $2H_2 + O_2 \rightleftharpoons 2H_2O$ which looks simple and straight forward. In reality however, this combustion consists of 8 species and over 38 reactions. For more complex fuel, this gets even more complicated and the combustion can consist of many hundreds and thousands of species and reactions which of course makes it a challenge to simulate.

In the Department of Combustion of Lund University, detailed simulations of simple combustion systems are done using Chemkin[18]. However, in order to simulate turbulent systems, such as engines, one needs to account for many parameters coming from the physics of the system as described earlier. These additions to the simulations require demanding calculation and can take a very long time to finish. In order to produce these simulations, approximations to the mechanisms need to be made.

2.3 Reduced chemistry

There are many various methods for reducing mechanisms which create several major categories. Some methods include, for example, *skeletal reduction* which simply removes species and reactions that have low impact on the detailed mechanism. There are many different methods to decide which reactions and species that are removed, even including the genetic algorithm. The left over mechanism is then called the skeletal mechanism[4]. Another method is called *Category Lumping*[4], which is when species of reactions of similar nature simply are combined, which means that the total number of variables to keep track of is reduced. This is usually applied to species with similar traits, such as diffusivity and thermal properties. This can reduce simulation time significantly, especially for larger mechanisms[4]. There are many more ways of doing mechanism reduction, and one generally has to pick one that fits the problem at hand the best.

2.4 Optimization of reduced mechanisms

Because of the necessary reduction of the mechanism, there are some errors introduced from the reductions. In order to combat this, one can change the reaction rate coefficients of the reduced mechanisms to increase the credibility again. A function is introduced that compares the results of simulations with reduced mechanisms with simulations using either the original mechanism, a more detailed mechanism or experimental data. It is thus possible to change the reaction rate coefficients and use that mechanism in a simulation to know if the change in coefficients made it more similar to a more detailed mechanism or not. This can be done by hand and trial and error,

but as there can be a lot of parameters to optimize, it is usually a good idea to use optimization algorithms if one does not want to spend a copious amount of time trying random combinations.

In general, an optimization algorithm is an algorithm that searches for the best element from a limited, but often big, set of elements. One can see it as a method to find global optima of a mathematical function. There are vast amounts of optimization algorithms with different strengths and drawbacks, which means that one can be smart about which one to use and pick the one that fits the problem at hand the best[11]. For this project, the choice of algorithms are *Simulated Annealing(SA)*[13] and *Genetic Algorithm(GA)*[14]. Simulated annealing is widely used, mostly for its impressive results, even though its a simple algorithm compared to many others. Its strength lies in its simplicity, effective usage for a wide variety problems, but most of all: Its ability to get stuck less often in local optima than other algorithms. The drawback is that it is comparably slow in finding a global optimum as it has to make many function evaluations when making guesses, with many guesses being bad and quickly dismissed, but more on that in *Method and Implementation*. The strength of genetic algorithm is also its weakness: its convergence behavior. It tends to find decent optima rather quick, but then also tends to get stuck there, meaning its very difficult to further improve on a solution from the genetic algorithm even if a lot of time and resources are being spent[12]. The two algorithms very different ways of finding optima also makes them very good together. The genetic algorithm searches widely over landscape that is being optimized, while simulated annealing is rather good at finding local optima. This means that one can use the genetic algorithm as a mean to find generally good spots for simulated annealing, which then searches for the local maxima there. The idea is to use these methods in tandem in an attempt to make them amplify each other's strength and mitigate each other's drawbacks, and create a *hybrid algorithm*.

3 Method and Implementation

3.1 Simulated Annealing

As stated earlier, one of the strengths of simulated annealing lies in its simplicity. It uses a temperature function that starts off at a certain temperature and is then cooled in a way best fit for the problem, often linearly or exponentially. It then picks out a random candidate point within a certain vicinity of the old point which throughout this project is a point within the closest 20%, as it produced the best results. This means that if the vectors that are being optimized have 100 elements, the closest 20 elements to the to the previous element can be chosen. Here, a candidate point is an array of currently chosen coefficients. The algorithm then has a certain probability to take a step to this point, calculated from the probability function

$$P = \frac{1}{1 + e^{(f_{new} - f_{old})/T}} \tag{2}$$

where f_{new} is the function value of the new point being evaluated, f_{old} the function value of the old point, T is the current temperature of the system and P is the probability for the algorithm to move to the new point. In this project, the goal for the algorithm is to find the global maximum and so the rest of the paper will focus on only that, but it works just as well for finding the global minimum. The probability function is the reason for why simulated annealing is good at avoiding getting stuck in local maxima. Since the next step of the algorithm is dependent on the probability function, and that in turn is dependent on the temperature, one has slight control over how the algorithm should behave from controlling the temperature.

When the algorithm evaluates the next candidate point, the probability function gives the probability of actually moving to that point. The idea is if the value of the function in that point is higher than the value in the current point, the probability function returns a high probability of taking that step, and of course a low probability for points with a lower function value. In the high temperature limit, the algorithm almost has a 100% chance of taking a step regardless if it is to a point with a higher function value (in the upwards direction), or to a point with a lower value. This means that it will essentially take random steps blindly. In the low temperature limit, the algorithm will only take a step if its in the upwards direction regardless of step size.

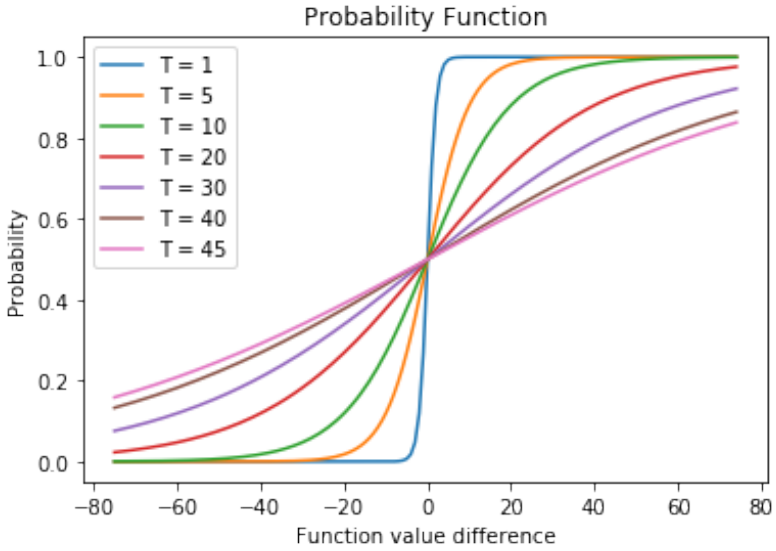

Figure 1: The probability of taking a step to the next candidate point is dependent on the difference in function value $f_{new} - f_{old}$, if it is in the upwards or downwards direction, and the temperature T. Here the probability function is plotted for different temperatures and one can clearly see that for low temperatures, the algorithm will almost exclusively take steps in the upwards direction.

The idea of simulated annealing is then to start out with a relatively high temperature, and then gradually lower it for every iteration of the algorithm, according to the temperature function. It will always be more likely to take a step in the upwards direction, but so that there is still a chance to take it in the downwards direction to avoid getting stuck in a local maximum. As the temperature decreases, it will become increasingly more likely for the algorithm to only take a step in the upwards direction. During the final iterations, the temperature will then be low enough for the algorithm to be allowed to exclusively take steps in the upwards directions where hopefully the global maximum is.

When the temperature reaches low enough values, the algorithm basically looks for the next local maximum for the next couple of iterations, because it is not allowed to take steps downwards anymore, and so it will inevitably get stuck. How many iterations it takes before getting stuck is dependent on the temperature function and how close the algorithm is to a local maximum. One can set the number of iterations the algorithm is allowed to do to be just a bit more than the amount of iterations it generally takes for it to get stuck. Since this algorithm is used many times when using the hybrid algorithm, which will be discussed later in this section, it is also important to start out with a temperature that is not too high. This is because simulated annealing will then have a chance to pick a very bad point and almost completely throw out the progress made from previous iterations from both algorithms.

For this project, simulated annealing was partly used as a mean to help the genetic algorithm to not get stuck in local maxima, so an exponential temperature function was not needed. The idea behind an exponential function is to quickly lower the temperature from the high temperatures where the algorithm takes more "random" steps, and focus on steady climbing with lower temperatures. For this case, it is always advantageous to have some temperature to help the system get unstuck, and so the best temperature function was found to be simply linear, and starting at $T = 40^o$, getting lowered by 0.2^o every iteration, ultimately ending at $T = 0^o$ where the temperature is in arbitrary units.

3.2 Genetic Algorithm

A genetic algorithm is an algorithm inspired by natural selection, utilizing biological traits such as crossover or *breeding*, selection and even mutation[14]. It starts out with a set number of candidate points; the *population size*. It evaluates all the candidate points and runs them through a fitness

test which assigns them a certain fitness score dependent on what the user seems necessary. For this project, the global maximum is known and is simply 1, so the candidates gets a score proportional to how close they are to the global maximum. These candidate points then have a chance of breeding and becoming "parents" according to a probability function which is very similar to the one for simulated annealing;

$$P = \frac{1}{1 + e^{S_f}} \tag{3}$$

but here it is only dependent on S_f, which is the fitness score. The reason this probability function is used again is simply to take the fitness score and turn it into a probability between zero and one, which it does very well. The candidate points that did not pass the test will simply be deleted, or "extinct" by natural selection. If only one or no candidate passed the test so that there are no pairs of parents for breeding, new candidate points will be chosen at random and the fitness test will start over. If there are more than two parents but an uneven amount, the parent with the worst fitness score will be deleted.

The surviving parents will then be matched up in pairs and breed, which will create two new candidate points called *children*, that will have the mixed "genes" from the parents. In this context, a gene is an element in an array that corresponds to the reaction coefficients that are being optimized. For example: if 20 coefficients are being optimized, a candidate point will be a 20 elements long array, where each element corresponds to a coefficient of a reaction. The breeding is then done through uniform crossover, which means that first child will get half of the genes of each parent, and the other child will get the other half.

Figure 2: In uniform crossover, the first child will get half of the first parent's gene, and the other half from the other parent. The second child will then get the leftover genes, so that it becomes the "opposite" of the first child.

When the breeding is finished, each surviving parent and child still has a chance to get their genes mutated. This is done in an attempt to diversify the population in order to sustain its convergence capacity[15]. A mutation, analogous to biology, is when a gene takes a random value within some predefined limits. Every gene of the parents and children has a $P_{mutation}/L_{array}$ chance to get mutated, where $P_{mutation}$ is the chance of a mutation happening and L_{array} is the length of the array. This is done in order to keep the chance of a mutation happening constant, regardless of the length of the array. If the result of the mutation is bad, the point that got the mutation will simply die off during the fitness test. If the result is good, the candidate points are more diversified and the chance of getting stuck in a local maximum decreases.

The parents and children are then added to the list of candidates that will be sent to the next iteration of the genetic algorithm. If there are less parents and children than the original population size, new random points are chosen as candidates. If the candidate points where exceptionally good and there are more children and parents than the population size, they are all still send to another fitness, test although this will mean that more evaluations are done, however, this does not happen too often. For this project, where an initial population size of 10 is used (picked randomly), there is typically one or two pairs of parents. Since the algorithms are compared with the total amounts of evaluations they do, not amount of iterations, this means that the genetic algorithm will go through less iterations than the simulated annealing, while still doing the same amount of evaluations.

3.3 Hybrid Algorithm Optimization

The idea behind the hybrid algorithm is simple: use one of the algorithms for some amount of iterations, take the point with the highest function value, and give that to the other algorithm and let it start from that point. For this case, simulated annealing was used first in order to find a decent starting point for the genetic algorithm. This way, there is a higher chance for the genetic algorithm to find good candidates to breed, and fewer unnecessary evaluations have to be made. There are many reasons the algorithms need to run some amount of iterations separately. First of all, a temperature function would not make sense if the algorithms would run one iteration each, as the high temperature in simulated annealing would have a high risk of throwing away decent points given by the genetic algorithm. The genetic algorithm would not make sense either, since it would just take the point given by simulated annealing and some random points, and have just a chance of breeding before giving back the point to simulated annealing again. They are simply not built to only run one iteration, and it is therefore necessary to run them separately for some time before they trade points with each other.

Because the genetic algorithm utilizes many points every iteration, it is possible to run simulated annealing multiple times, finding many decent points that the genetic algorithm can start from. A lot more evaluations have to be made by simulated annealing, at the same time a lot less unnecessary evaluations have to be made by the genetic algorithm. It then has a high chance of breeding most of the candidate points, leading to a strong evolution of the points by natural selection, hopefully leading the hybrid algorithm closer to the global maximum.

3.4 Testing the algorithms

During the optimization, every evaluation of the chosen coefficients means that a simulation is done with the current coefficients. As said before, these types of simulations take a considerable amount of time, which means that every evaluation takes a considerable amount of time. It is therefore very time consuming and inefficient to test which parameters of the algorithms give the best results for the least amount of computing time, even if it gives the best accuracy. Instead, all the testing of the algorithms is done on a predefined set of mathematical functions in form of vectors. The vectors consist of sinus functions with different frequency and amplitude from each other. When they are evaluated, they go through a simple function where they are added to each other, which means that the global optimum is found when the maximum of every single vector is found. When optimizing the coefficients of the reduced mechanisms for the simulations, the result of the simulation is compared to a result of previous simulations with detailed mechanisms, and the difference is normalized to a score between -1 and 1, where 1 is very good correlation and -1 is the opposite. Because the global maximum of the test function simply is all the amplitudes added together, the global maximum is also known. This is then normalized in a similar way, so that the global maximum is 1 for the test function as well.

When testing different parameters for the algorithms, it is important that they always start from the same point. Otherwise, the tests would mean nothing as the complexity around different points might differ greatly and the results would lose meaning. However, if the tests are run from only one starting point, the algorithms would be optimized around that point only. For that reason, testing of different parameters is run on many separate starting points that are all chosen randomly beforehand. The algorithms can then be tested on a variety of points, as long as they are compared only when starting on the same point as each other.

4 Results and discussion

4.1 Test functions

The following results are taken using the test functions with a set of constant starting points. Each figure contains results from optimization from a different starting point. For the cases where only simulated annealing and the genetic algorithm are run, they are run from 4 different starting points: A, B, C and D which can be seen in Figure 3 and 4. The other tests where only one plot is shown all used starting point A.

4.1.1 Simulated Annealing

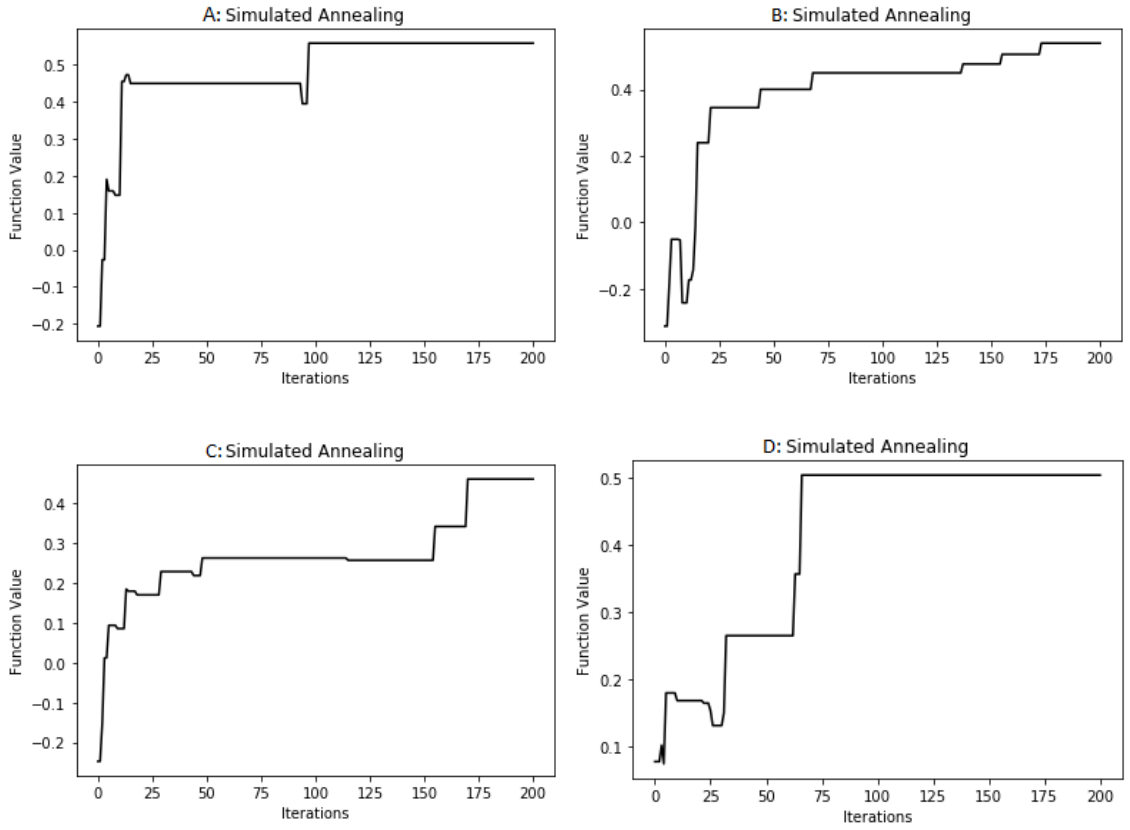

Figure 3: The function value of the points being evaluated is plotted for every iteration. During early iterations when the temperature is high, the algorithm is allowed to take steps in the downwards direction letting it move around more freely than when the temperature is low and it gets stuck at local maxima.

Here, simulated annealing is used to optimize the test function. In the four plots in Figure 3, the algorithm started out in 4 different starting points chosen randomly. The temperature started at $T = 45^o$ and decreased linearly to $T = 5^o$ after 200 iterations. Since there is only 1 evaluation for every iteration of simulated annealing, there were 200 evaluations done.

It becomes clear here that simulated annealing finds relatively good local maxima rather quickly, but then tends to stay around the same point for many iterations after that, even if the temperature is still high. When it reaches local maxima during low temperatures, it will keep making guesses where to move next, but as the function value of all nearby points probably is less than the one it is currently in, it will stay put, producing the flat lines in Figure 3. In order to save computational resources and reduce simulation time, it is therefore advantageous to cut the number of iterations of simulated annealing down to 75, where it tends to have already found a local maximum. This way, simulated annealing will find nearly as good results as for 200 iterations, but in less than half the time. One could also introduce functions that in smart ways would take measures to avoid these flat lines by, for example, increasing the temperature substantially for some iterations when the algorithm is stuck.

The choice of temperature function also has a big impact on the effectiveness of this algorithm. It is important that the temperature does not start out too high, as there will be considerable risk of taking a step away from a good maximum, leaving previous work useless. If the temperature starts out too low, the algorithm will instead get stuck rapidly and in very small local maxima. The goal of simulated annealing is in this case to effectively, and with minimal resources, find the best local maximum and give it to the genetic algorithm. This is why it is very good to cut the amount of iterations down to 75 and always have a temperature that is not close to zero; the algorithm will with relatively low cost find a decent maximum which will serve as a very good parent for the genetic algorithm. One could let the temperature go down to zero, which would only allow for steps in the upwards direction for even better candidates, but it would take significantly longer time and would only barely improve the results.

Even though measures are taken in order for simulated annealing to avoid getting stuck, it will happen. It is an unavoidable consequence of the algorithm and it only means that a good local maximum is found. There will be times where the genetic algorithm gives an already decently optimized local maximum, and a lot of unnecessary iterations will be done. In those cases, it would be beneficial to simply interrupt simulated annealing and let the genetic algorithm find a new point to start optimizing from. Unfortunately, because of time constraints it could not be implemented.

4.1.2 Genetic Algorithm

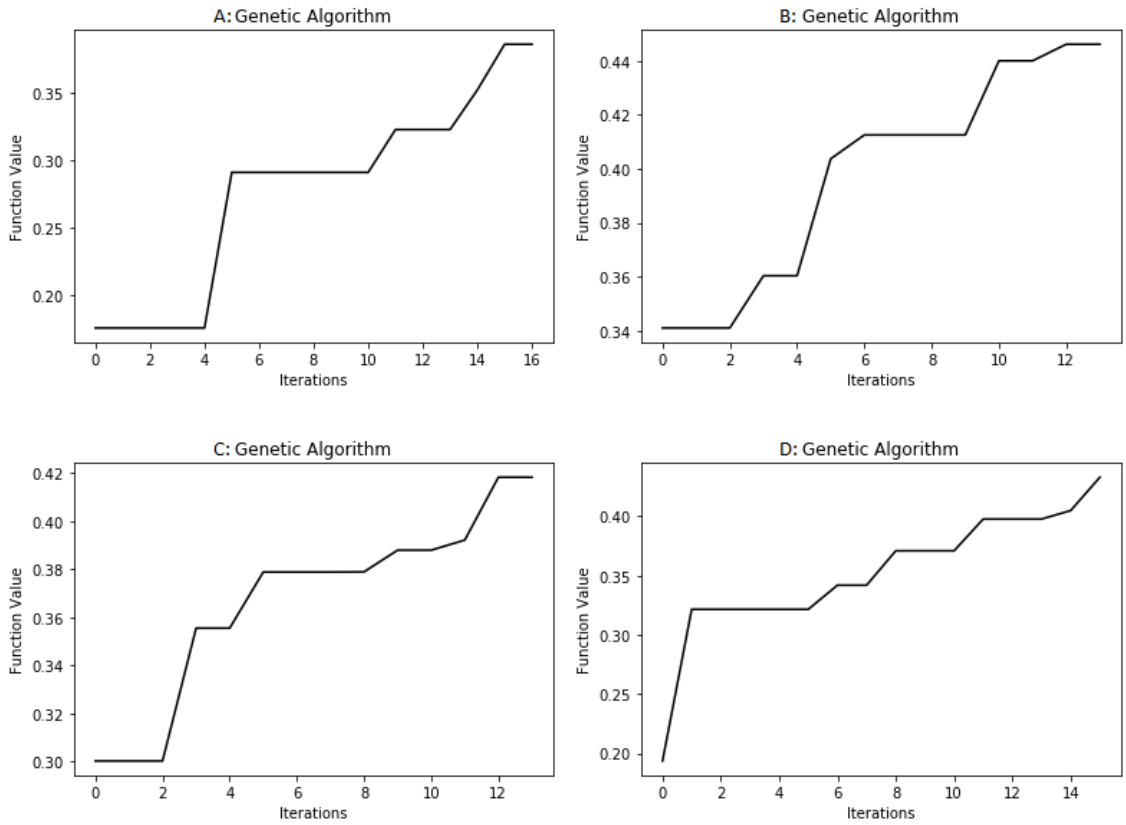

Figure 4: The function value of the best evaluation is plotted for every iteration. Here, 200 evaluations were allowed, which results in approximately 15 iterations.

Similarly for the genetic algorithm, Figure 4 shows the progression of the genetic algorithm on the same test function for 4 different starting points. An initial population size of 10 candidate points is used here, with a 50% crossover chance of the genes between parents, and a 10% mutation rate. Just like in simulated annealing, the limit is set to 200 evaluations. If a lot of the candidate points are good and pass the fitness test, many become parents and in turn create many children. These are all passed on to the next iteration, sometimes outnumbering the initial population size. This means that more than 10 evaluations are completed during one iteration, and the total amount of iterations decreases. There will thus be a maximum of 20 iterations done, but as seen in Figure 4, it is usually less then that.

Adverse to simulated annealing, the genetic algorithm generally has slower but more steady climb and tends to not get stuck as fast, which means that no cut has to be made here. In the test runs in the plots of Figure 4, it is clear that one could simply increase the amount of iterations and have a very high chance of getting better results. However, the steady climb in these situations is more because of the relatively low starting points. As the algorithm approaches the global maximum, genetic algorithm will tend to get stuck more quickly as well, and so it is appropriate to keep the amount of iterations when optimizing for the simulations.

As stated before, because the results of simulations with the reduced mechanisms are compared to simulations with more detailed mechanisms, the global maximum is known. This is a big advantage as one can have a simple fitness function that only depends on how close the algorithm is to the global maximum. Fitness functions for other systems might depend on the candidates' relative value to each other, increasing the risk of getting stuck locally. If the genetic algorithm gets stuck in local optima for this case, it is a high chance of it being for high values, or near the global maximum.

4.1.3 Repeated Simulated Annealing

In order to more easily be able to compare the separate algorithms with the hybrid algorithm to see if a hybrid algorithm is an improvement, each algorithm was run 300 iterations, with the same parameters as described before.

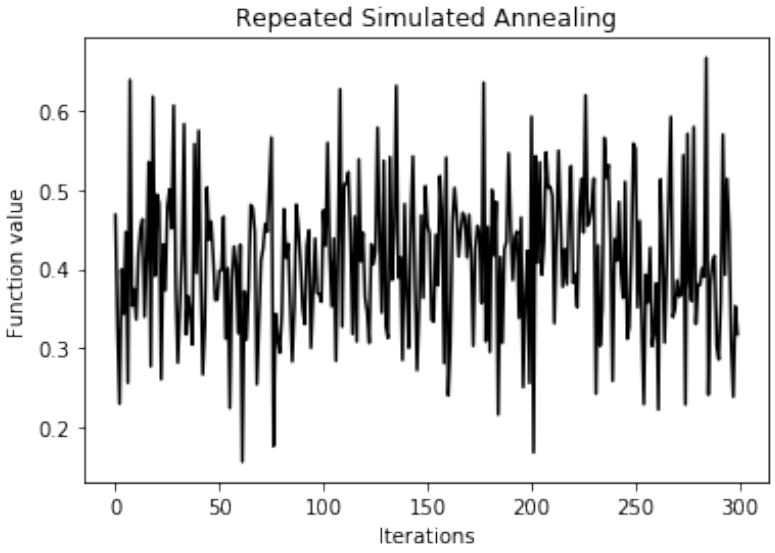

Figure 5: Every iteration here is the best value of a separate simulated annealing algorithm run for 200 evaluations. Every iteration, the algorithm finds a local maximum and the next iteration, the temperature will have to be reset so it will escape the local maximum, but sometimes to a point with a relatively low function value.

Every iteration in the plot in Figure 5 consists of 200 iterations of simulated annealing. This is then simply a 60 000 iterations long simulated annealing, where the temperature is reset every 200 iterations. As one can see, this way of optimizing is very inefficient, often wasting significant amounts of time looking around lower values. This clearly shows the strength and weakness of simulated annealing; its ability to find local maxima very quickly but then getting stuck there. Having a big increase in the temperature is required for it to not simply be stuck in the same position for 60 000 iterations, but as seen in the plots, introduces the risk to take too big leaps downwards.

Although the best value of repeated simulated annealing was relatively low, it found a decent value in a very low amount of iterations. It then finds similar values multiple times, only improving them slightly. One can then draw the conclusion that increasing the number of iterations of simulated annealing is a very ineffective way of finding the global maximum for this system.

4.1.4 Repeated Genetic Algorithm

Similarly as for repeated simulated annealing, the same parameters as for the separate case in Figure 4 are kept, and genetic algorithm is repeated 300 times. Because there is no temperature function in the genetic algorithm, this is exactly the same as simply having 60 000 iterations of the separate algorithm.

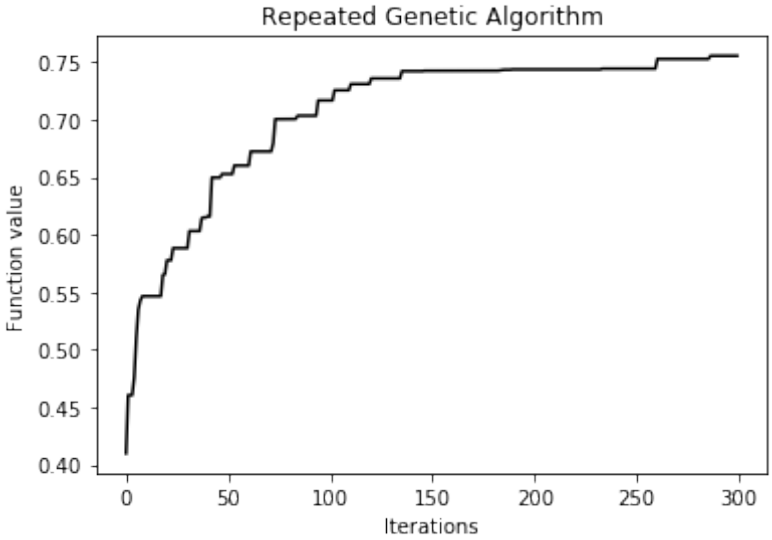

Figure 6: Just like in Figure 5, here the genetic algorithm is repeated with 200 evaluations being done for every iteration.

In Figure 6, it is very easy to see how the genetic algorithm acts for many iterations. It starts off with a relatively steep climb that converges to some global maximum, which it then spends a lot of time being stuck in, just as predicted. Compared to repeated simulated annealing, it does take some more iterations to pass a function value of ~ 0.63, which simulated annealing found rapidly. However, when it does pass that value, it keeps on improving, ultimately finding a significantly better maximum. Because there is a lot of random elements in optimization, it is important to note that this is not always the case, but for a majority of the time it is. Sometimes simulated annealing will find a better global maximum from "being lucky", but it is rare, and almost every time the genetic algorithm will have an advantage for larger amounts of iterations.

4.1.5 Hybrid Algorithm

Here the two algorithms are run every other iteration, with the same parameters as before. Just as then, every iteration of each algorithm consists of 200 evaluations. In Figure 7, one can see characteristics of both algorithms; both the very rapid climb of simulated annealing for early iterations and then the slower steady climb of the genetic algorithm. Figure 7 also shows the strength of the two algorithms combined. As the algorithm presumably reaches a local maximum, simulated annealing takes a relatively big leap downwards, effectively "dodging" the local maximum, ultimately leading to a higher function value in the end. Whether or not this value is higher or lower than if simulated annealing did not take this downwards step, is hard to know for this specific case, but in general this should avoid getting stuck in local maximum more, resulting in higher function values on average. For the test functions that this algorithm is tested on, the hybrid algorithm regularly finds better function values than the repeated separate algorithms.

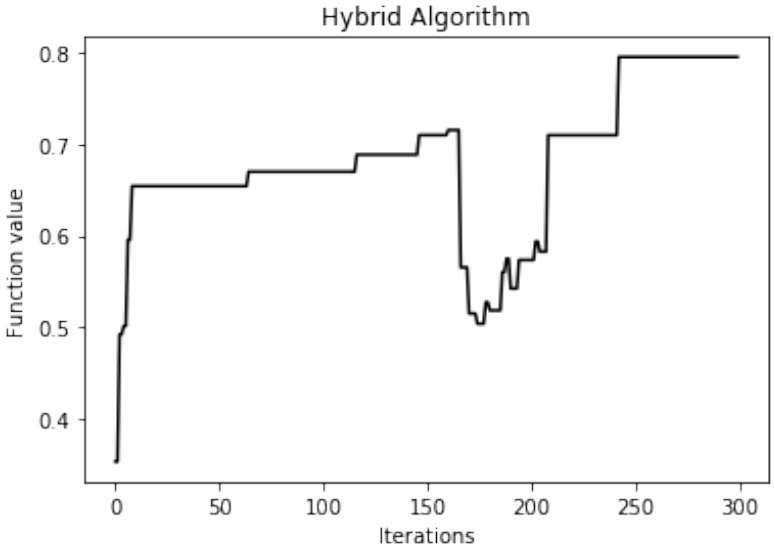

Figure 7: Simulated Annealing and Genetic algorithm is every other iteration of this plot for 200 evaluations each.

4.1.6 Variations of Hybrid Algorithm

Because the genetic algorithm utilizes multiple evaluations every iteration, one can give it many inputs instead of taking random points. This means that one can run simulated annealing multiple times, quickly finding many local maxima and giving them all to the genetic algorithm as strong starting points. As usual, this is done for 300 iterations and the results can be seen in Figure 8.

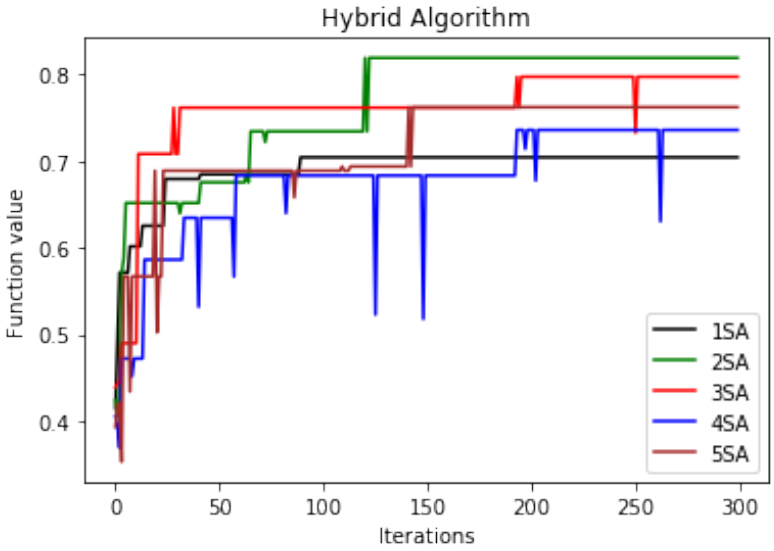

Figure 8: Simulated annealing can be run multiple times before the genetic algorithm in order to give it multiple decent candidates for breeding. Here, "1SA" means that simulated annealing is run every other iteration, "2SA" means that simulated is run 2 times before the genetic algorithm and so on.

One can see that the different versions of the hybrid algorithm do behave similarly, but some tend to do better than others. Two, three and four iterations of simulated annealing seem to almost always find the best maximum, meaning that running simulated annealing multiple times is advantageous for this system as long as it is not done excessively. It is also common for the algorithms with more iterations of simulated annealing to take a step in the downwards direction and then almost immediately find its way back to where it was, which is characteristic of simulated annealing. During early iterations, running simulated annealing multiple times quickly finds relatively high function values, as it generally picks good candidates for the genetic algorithm to start with, instead of letting it pick randomly. This means that the algorithm often starts out very strong, even if there would be many bad points to choose from. However, when the algorithm approaches maxima that are difficult to escape from, simulated annealing will very often find the same maximum every time it is run, meaning that the genetic algorithm will have many identical candidates. Since all the candidates are at, or close to a local maximum, they have a high function value, meaning it is very likely that they will pass the fitness test and be used multiple times. This would greatly decrease the diversity of the genetic algorithm, thus increasing the likelihood of it getting stuck. This means that it is almost entirely up to simulated annealing to escape the local maximum, which it can be quite good at as there are nearby points to escape to.

One can also see that there is definitely an abundance of local maxima with function values close to the global maximum, as the different algorithms tend to get stuck in unique maxima, just like in the plot. Even though this algorithm is supposed to be good at dealing with local maxima, it is almost unavoidable to get stuck at some point while still having an effective algorihtm, otherwise one would have a perfect optimization algorithm. Finding the global maximum is the ideal case, but finding good maxima with function values close to the global maximum is often more practically realistic.

4.2 Optimization of reaction coefficients

The previously described algorithms were then used to optimize the reaction coefficients in the reduced mechanisms used to run simulations in Chemkin with the same parameters that were used before. Because time was a very limiting factor and each evaluation requires a simulation that takes very long time compared to the test functions, there could only be one test done for each algorithm. Therefore, the parameters could not be changed and tested for the new system in order to fit it better. Another difference between optimization of the coefficients of the reduced mechanism and the test function, is that the test function used vectors with a finite amount of elements while for this case, the coefficients are simply a scalar that can be changed to any value. Generally, it is only changed within some percentage of the previous value.

The coefficients that were originally used for simulations give an evaluated value of $V_{original} = 0.8209$ so if coefficients are found that produce an evaluated value higher than this, they are an improvement.

4.2.1 Simulated Annealing

Again, in order to more easily be able to compare the two algorithms to each other, they were first run separately and were only allowed to make 300 evaluations each. Here, the same temperature and temperature function as usual was used in simulated annealing, only difference being that the temperature was reset every 75 iterations instead of every 200 because of the cut.

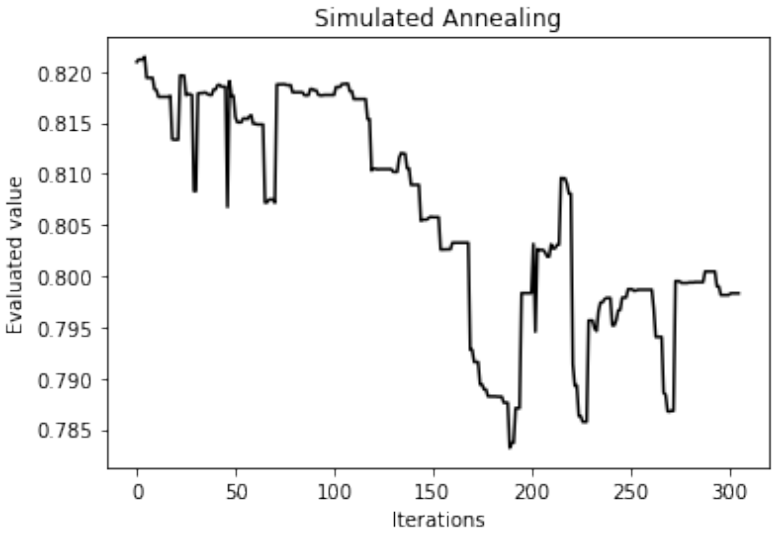

Figure 9: Simulated annealing was first used individually to optimize the reaction coefficients of the reduced mechanism.

As can be seen in Figure 9, this run of simulated annealing clearly had a trend of going in the downward direction, although only decreasing the evaluated value by ~ 0.035 at worst. It did, however, find a slightly better set of parameters during very early iterations, which gave a evaluated value of $V_{SA} = 0.8215$. It is unfortunate that there was not more time to run simulated annealing, as it shows no signs of being close to getting stuck in Figure 7. Because simulated annealing picks random points and is allowed to take steps downwards, it will do so when it is close to a maximum. Given enough time, it might have found a better maximum but testing the hybrid algorithm had priority.

4.2.2 Genetic Algorithm

Because continuous coefficients are now being optimized instead of vectors with finite amounts of elements, a new parameter has to be introduced. Previously when the genetic algorithm picked a new random candidate, it simply picked a random element out of all elements of each vectors, thus being able to randomly pick any possible candidate of the system. For continuous coefficients, this is not possible as it would then have to pick any real number, which would create an infinite amount of random candidates which mostly are useless. Instead, a new parameter is used that tells the genetic algorithm how far from the value of the current coefficient it is allowed to pick a new coefficient. For the rest of this project, this is chosen to be within the closest 50% of the original value.

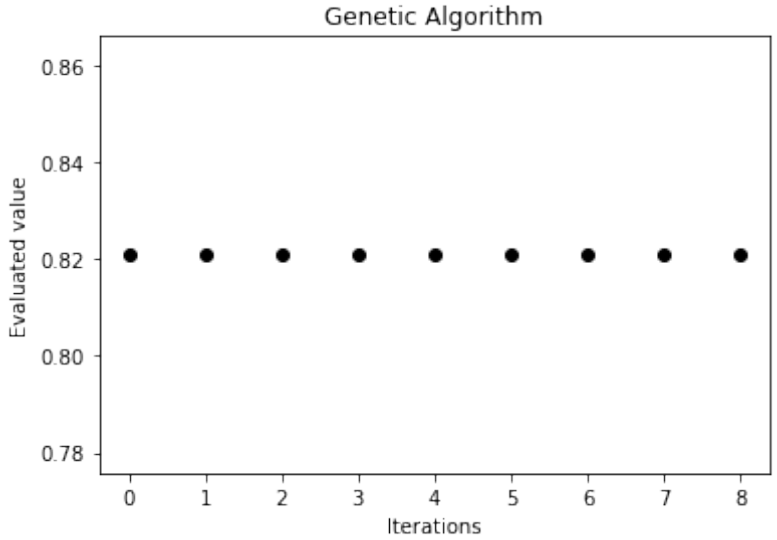

Figure 10: The genetic algorithm was run for 8 iterations, during which 300 evaluations were made.

In Figure 10, it might look like something is wrong, but everything worked as it should. Only the set of coefficients that produces the best evaluated value is saved for plotting, meaning that no improvements were found. Because the best candidates have a high chance of surviving the fitness test, they have a high chance of being reused the next iterations which is what happened here. The initial set of coefficients were simply good enough to survive every iteration, and since no better set was found, it produces this flat line.

Because the algorithm started optimizing from a set of coefficients that already give high evaluations, many of the random candidates that are added have a high chance of passing the fitness test. As a result, there will be many parents and children passed on to the next iteration, which again will have a significant chance of passing the fitness test. This means that even for early iterations, the amount of candidates will surpass the initial population size of 10. Every iteration thus has high probability of requiring more than 10 evaluations, making it so that only 8 iterations were done even though 300 evaluations were allowed. Because of this, the amount of iterations will in general be random for the genetic algorithm. When optimizing coefficients that give worse evaluations, less candidates will pass the fitness test, and more iterations will be done. Only for very high values like for this case will there be this many evaluations per iteration. If more time were given, the fitness functions could be modified to operate for high evaluation values which might improve the genetic algorithm for this case. One could also limit the amount of candidates that are allowed to be passed by the fitness function, but this would probably do more harm than good as one might then ignore the best candidate without knowing it.

4.2.3 Hybrid Algorithm

The two algorithms were then, as before, run every other iteration. The genetic algorithm was allowed to make 200 evaluations per iteration, while simulated annealing was run for 75 iterations. Again, because time was a limiting factor, the hybrid algorithm could only be run for 8 iterations, although a total of 1100 evaluations were done.

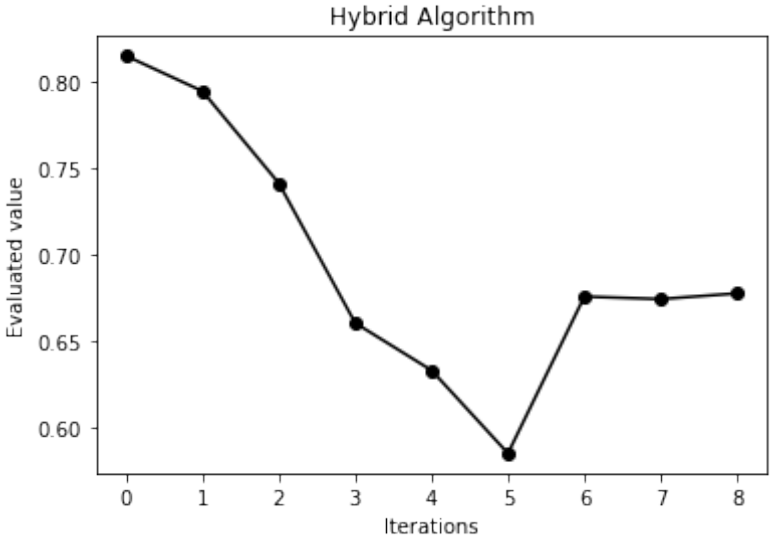

Figure 11: Simulated annealing and the genetic algorithm were again run every other iteration, allowed to make 75 and 200 evaluations respectively.

Surprisingly, the hybrid algorithm produced significantly worse results than the separate algorithms did. In just 5 iterations, it had regressed down to an evaluation value just under 0.6, which is a substantial decrease. What is even more surprising, is that for the first 5 iterations, both simulated annealing and the genetic algorithm exclusively took big steps in the downwards direction. This was not seen for either of the separate cases where the genetic algorithm took no steps downwards, and simulated annealing only took very small steps compared to those in Figure 11. Because the algorithm starts out with simulated annealing, one can see that it is run every even iteration, while the genetic algorithm is run every odd iteration. One can then also see that the only algorithm that took significantly good step was simulated annealing, which took a big step the 5th iteration. The genetic algorithm did take one good step as well in the final iteration, although it was minimal in comparison.

Because the fitness function of the genetic algorithm is random, it is never guaranteed for a good point to pass unless it is the global maximum. Even very good points run a risk of not passing, meaning that the algorithm is allowed to take steps in the downwards direction. However, generally this only happens rarely, especially as the algorihtm approaches evaluations close the the global maximum. It is therefore very odd to see it almost exclusively taking bad steps, especially of this magnitude. Although for low amounts of iterations such as in this case, the risk of it happening definitely is not negligible.

The main reason for the poor performance of the hybrid algorithm is likely because it was created and optimized for the test function which is an entirely different system. As said before, generally one has to pick an algorithm specifically for which system one is optimizing as every algorithm is unique and has different strengths and weaknesses. A big part of optimizing is also to tweak the chosen algorithm in ways the best fits the system at hand and is often done by simply testing different parameters. For this case, all the testing had to be done on a different system for many reasons, with the main reason being that every test of the unfinished algorithms would take significant time because of simulations. In order to make two working algorithms, many runs have to be done and the time it would take to do them with simulations is unreasonable. The test functions that were used also significantly differed from the real case, as they were vectors with a finite amount of elements, and the real coefficients were scalars. Perhaps by starting out with a test function that also depends on scalar values would make the systems more alike, producing better results in the end. It does not however avoid the big downside of using a test system, which is that one knows nothing or at best extremely little about the real system that is to be optimized, and making a test system that is similar is not possible.

5 Summary and conclusion

Every optimization algorithm is suited for different problems and has unique strengths and weaknesses. Simulated annealing and the genetic algorithm have strengths that coincidentally match with the the other's weakness and the other way around. The genetic algorithm's ability to spread out widely over the system that is being optimized combined with simulated annealing's ability to unstuck and quickly find local maxima together make for an algorithm that is efficient and adaptable for many optimization problems.

For this project, the hybrid algorithm exceeded both separate algorithms when applied to a set of test vectors and a test function when all algorithms were allowed to do the same amount of evaluations. Not only did it find decently good maxima as quick, or quicker than the other algorithms, it also frequently found better maxima. When used to optimize the reaction coefficients of a reduced mechanism in simulations, however, it produced poor results. This is largely a result of the hybrid algorithm being created and optimized for a different system, and it could not be adapted to another system properly because of time constraints. These results could be improved upon in many ways given more time. For example, the algorithm could have been developed for optimization of the reaction coefficients originally, customizing it for the problem. Running the algorithm over an extended amount of time would also assumably improve the results, as optimization is very time dependent. Overall, hybrid optimization using the genetic algorithm and simulated annealing is potentially a very effective algorithm that should be further explored.

References

[1] Ustun D, Akdagli, Ali. A study on the performance of the hybrid optimization method based on artificial bee colony and differential evolution algorithms. Software Engineering, Faculty of Tarsus Technology Mersin University Mersin, Turkey. 2017

[2] Fiorina B, Veynante D, Candel S, Modeling Combustion Chemistry in Large Eddy Simulation of Turbulent Flames, Laboratoire d'Energ´etique Mol´eculaire et Macroscopique. 2015

[3] Menon S, Fureby, C, Computational Combustion, Georgie Institute of Technology, Atlanta, GA, USA. 2010

[4] Lu T, Law K. C, Towards accommodating realistic fuel chemsitry in large-scale computations, Department of Mechanical and Aerospace Engineering, Princeton University, Princeton. 2009

[5] Bergthorson M. J, Thomson J. M, A review of the combustion and emissions properties of advanced transportation biofuels and their impact on existing and future engines, Department of Mechanical Engineering, McGill University, 2015; 42 1393-1417

[6] Blurock S. E, Tuner M, Mauss, F Phase optimized skeletal mechanisms for engine simulations. In: Combustion Theory and Modeling, Taylor & Francis, 2012. p. 295-313

[7] Thierry Poinsot M, Laurent Selle M, Numerical study of limnar and turbulent flame propagating in a fan-stirred vessel, Institut National Polytechnique de Toulouse. 2014

[8] Philpp O, Hoepler R, Chucholowsi C, Dynaware T, Zero-Dimenstional Combustion Simulation in Real Time, AutoTechnology. 2007

[9] Laminar and turbulent flame speed, from Huth M, Heilos A, Modern Gas Turbine Systems, 2013, https://www.sciencedirect.com/topics/chemistry/laminar-flame. Visited 2018-04-08

[10] A Williams, Flames, http://www.thermopedia.com/content/766/. Visited 2018-04-08

[11] Kumar R, Optimization: algorithms and applications, Boca Raton : CRC Press. 2015

[12] Guo H, Zhou B, Yang P, Gu X, Application of modified Stribeck model and simulated annealing genetic algorithm in friction parameter identification, Intelligent Systems and Knowledge Engineering. 2017

[13] Kirkpatrick S, Gelatt D. C, Vecchi P, M, Optimization by Simulated Annealing, science, New Series, Vol.220, p 671-680. 1983

[14] Melanie M, Genetic Algorithms: An Overview,in: An Introduction to Genetic Algorithms, A Bradford Book The MIT Press, London; 1999. p.2-4.

[15] Srinivas M, Patnaik M. L, Adaptive Probabilities of Crossover and Mutation in Genetic Algorithms, IEEE TRANSACTIONS ON SYSTEMS, MAN AND CYBERNETICS, VOL. 24, NO. 4, APRIL 1994

[16] Glassman I, Yetter A. R, Combustion, British Library: ISBN: 978-0-12-088573-2, 2008.

[17] Arrhenius Equation, www.khanacademy.org/science/chemistry/chem-kinetics/arrhenius-equation/v/arrhenius-equation, visited 2018-05-02

[18] Chemkin, company, product and information, http://www.reactiondesign.com/products/chemkin/

6 Appendix A: Code

The code used for testing the optimization algorithms is the following: (looks very weird because of the indenting of latex)

```
import pylab as pl
%matplotlib inline
import numpy as np
import random
from copy import copy, deepcopy

def createFunction(vectorAmount, xLength):
        vectorList = []
        globalMax2 = 0
        vectorListFile = open("vectorListFile.txt", "w+")
        x = np.linspace(0, xLength, xLength)
        for i in range(0, vectorAmount):
                Amplitude = 0.01*random.randint(50, 100)
                Frequency = random.randint(3,10)
                currentVector=[0 for y in range(xLength)]
                for j in range(0, xLength):
                        currentVector[j]=(Amplitude*np.sin(Frequency*j/10))

        vectorListFile.write(str(currentVector) + "\n")
        globalMax2 = globalMax2 + Amplitude
        vectorList.append(currentVector)
        vectorListFile.close()

        globalMaxFile = open("Global_Max.txt", "w+")
        globalMaxFile.write(str(globalMax2))
        globalMaxFile.close()

        #print("Global Max is " + str(globalMax2))

        return vectorList

def makeInitialPoints(xLength, vectorAmount):
        initialPointAmount = 10
        initialPointFile = open("initial_Points.txt", "w+")
        for j in range(0, initialPointAmount):
                initialPoint = []
                for i in range(0, vectorAmount):
                        initialPoint.append(random.randint(0, xLength-1))
                initialPointFile.write(str(initialPoint) + "\n")
        initialPointFile.close()


def simFunc(vectorList, Points):
        #Adds the vectors together to form the "landscape" to optimize
        addedValue = 0
        #print("i simfunc:" + str(Points))
        for i in range(0, len(Points)):
                addedValue = addedValue + vectorList[i][Points[i]]
        #print("still inside" + str(addedValue))
        return addedValue

        def pickRandom(vector, lastPoint, saRange):
        vInterval = int(round(len(vector)*saRange))#How many of the
        #adjacent points can be picked from
```

```
        lowEnd=lastPoint-int(round(vInterval/2))
        highEnd=lastPoint+int(round(vInterval/2))

        newPoint = random.randint(lowEnd, highEnd)
        #IF the new point is out of bound, assignt a new point until
        #it's in bounds
        while newPoint > len(vector)-1 or newPoint<0:
                newPoint = random.randint(lowEnd, highEnd)

        return newPoint

def mutate(newCandidateList, pMutation, vectorList):
        for j in range(0, len(newCandidateList)):
                for i in range(0, len(newCandidateList[j][:])):
                        randomTemp = random.random()
#probability of mutation is (mutation chance)/(length of vector)
#the probability to mutate is the same regardless of length of candidate list

                if randomTemp <= pMutation/len(newCandidateList[j][:]):
                        newCandidateList[j][i]=random.randint(0, len(vectorList[i][:])-1)
                        #print("Gene succesfully mutated!")

        return newCandidateList

def simAnn(vectorList,
                   T,
                   saIteration,
                   usePreviousGAInSA,
                   bestGACoordinate,
                   saPlot,
                   bestPreviousSA,
                   usePreviousSAInSA,
                   saRange):

        bestCoordinate = []
        history = []
        randomPoint = []
        normalizedHistory = []

        #INITIALIZATION
        if usePreviousGAInSA == 0 and usePreviousSAInSA == 0:
                #Take random initial points
                for i in range(0, len(vectorList[:])):
                        randomPoint.append(random.randint(0, len(vectorList[i][:])-1))
                simValue=simFunc(vectorList, randomPoint)
                history.append(simValue)
                normalizedHistory.append(history[0]/globalMax)

        elif usePreviousSAInSA == 1:
                simValue=simFunc(vectorList, bestPreviousSA)
                history.append(simValue)
                normalizedHistory.append(history[0]/globalMax)
                randomPoint = deepcopy(bestPreviousSA)

        elif usePreviousGAInSA == 1:
                simValue=simFunc(vectorList, bestGACoordinate)
                history.append(simValue)
                normalizedHistory.append(history[0]/globalMax)
```

```
                randomPoint = deepcopy(bestGACoordinate)#DEEPCOPY



        #MAIN SA LOOP
        for j in range(0, saIteration):
                for i in range(0, len(vectorList[:])): #Picks next value for vector
                        randomPoint[i]=pickRandom(vectorList[i], randomPoint[i], saRange)
                newValue=simFunc(vectorList, randomPoint)
                #Compare next step to previous step with the probability function
                probFunc = 1/(1+np.exp(-100*(newValue-simValue)/T))
                randomFloat = random.random()
                if randomFloat < probFunc:
                        simValue = deepcopy(newValue)
                        bestCoordinate = deepcopy(randomPoint)


#IF a better coordinate was never found,
#and we're at the last iteration, pick last best point
                if j >= saIteration-1 and len(bestCoordinate) <= 2:
                        bestCoordinate = deepcopy(bestGACoordinate)

                #Add the history of the algorithm for plots
                history.append(simValue)
                normalizedHistory.append(history[j]/globalMax)

                #TEMPERATURE FUNCTION
                #Round to one decimal to avoid precision warnings in exp function
                T=round(T-0.2, 1)
                if T<=0:
                break

                #PLOT
                if saPlot == 1:
                title = "Simulated Annealing"
                xName = "Iterations"
                yName = "Function Value"
                k = 1
                historyAxis = [x for x in range(len(history))]
                plot(historyAxis, normalizedHistory, xName, yName, title, k)

        #print("SA succesful")
        return bestCoordinate

def geneticA(gaIteration,
                        populationSize,
                        pCrossover,
                        pMutation,
                        vectorList,
                        knowMax,
                        globalMax,
                        bestSACoordinate,
                        bestGACoordinate,
                        maxEvaluations,
                        gaPlot,
                        usePreviousSAInGA,
                        usePreviousGAInGA,
                        parentMortality):
```

```
Candidates = []
History = []
bestValue = -1*globalMax
bestCoordinate = 0
evaluationCount = 0

#Pick random candidates
for j in range(0, populationSize):
        currentCandidate=[0 for i in range(len(vectorList[:]))]
        for i in range(0, len(vectorList[:])):
                currentCandidate[i]=random.randint(0,len(vectorList[i][:])-1)
        Candidates.append(currentCandidate)
#Add candidates from SA
if usePreviousSAInGA == 1:
        for x in range(0, len(bestSACoordinate[:])):
                Candidates.append(bestSACoordinate[x])
#Add previous best GA point again
elif usePreviousGAInGA == 1:
        Candidates.append(bestGACoordinate)


for k in range(0, gaIteration):
Evaluation = []
#Evaluate the candidates
for i in range(0, len(Candidates)):
        Evaluation.append(simFunc(vectorList, Candidates[i]))
        evaluationCount = evaluationCount + 1
#Save the highest Value and its coordinate
for i in range(0, len(Evaluation)):
        if Evaluation[i] > bestValue:
                bestValue = Evaluation[i]
                bestCoordinate = Candidates[i]
History.append(bestValue)

#Fitness test
#Loop through candidates and give them a score now only
#based on how close they are to the global maximum
parentCoordList = []
lowestFitness = globalMax
lowestFitnessCoord = 0
for i in range(0, populationSize):
        currentEval=Evaluation[i]
        fitnessPoint=currentEval/(globalMax/4)
        #Calculate probability for candidate to become parent
        parentProb= 1/(1+np.exp(-(fitnessPoint)))

        tempRand = random.random()
        if tempRand < parentProb:
                parentCoordList.append(i)
                #Save the candidate with lowest fitness score to remove
                #if the amount of parents become uneven
                if fitnessPoint < lowestFitness:
                        lowestFitness = fitnessPoint
                        lowestFitnessCoord=lowestFitnessCoord+1

#Because I have to start the variable at 0, the whole list gets shifted by 1.
lowestFitnessCoord=lowestFitnessCoord-1
```

```
if len(parentCoordList) > 2: #If there's an uneven amount of parents, remove the worst.
        if len(parentCoordList) % 2 != 0:
                parentCoordList.remove(parentCoordList[lowestFitnessCoord])


newCandidateList = []
#Breeding time, but only if there's atleast 2 parents.
if len(parentCoordList) >= 2:

parent1 = 0
parent2 = 1
for j in range(0, int(len(parentCoordList)/2)):
currentChild1 = [0 for j in range(len(vectorList[:]))]
currentChild2 = [0 for j in range(len(vectorList[:]))]

for i in range(0, len(vectorList[:])):
#There's a 50% chance that the genes of the parents
#"swap" places when creating the children
geneMixProb = random.random()
if geneMixProb < 0.5:
        currentChild1[i] = Candidates[parentCoordList[parent1]][i]
        currentChild2[i] = Candidates[parentCoordList[parent2]][i]
else:
        currentChild1[i] = Candidates[parentCoordList[parent2]][i]
        currentChild2[i] = Candidates[parentCoordList[parent1]][i]

        #Add the parents and children to the new list of points for the
        #next evaluation. In order to avoid having the same parents over
        and over, have a 50% of them dying.
        #It is only used situationally, and wasn't used for simulations
        if parentMortality == 1:
                survivalRate = random.randint(1, 2)
                if survivalRate == 1:
                        newCandidateList.append(Candidates[parentCoordList[parent1]][:])
                        newCandidateList.append(Candidates[parentCoordList[parent2]][:])
                        newCandidateList.append(currentChild1)
                        newCandidateList.append(currentChild2)
                elif parentMortality == 0:
                        newCandidateList.append(Candidates[parentCoordList[parent1]][:])
                        newCandidateList.append(Candidates[parentCoordList[parent2]][:])
                        newCandidateList.append(currentChild1)
                        newCandidateList.append(currentChild2)

                parent1 = parent1 + 2
                parent2 = parent2 + 2

        #MUTATION
        if pMutation != 0:
        newCandidateList = mutate(newCandidateList, pMutation, vectorList)


        #If the amount of parents and children are less than
        #the population size, add new random candidates
        newCandidateAmount = 0
        while len(newCandidateList) < populationSize:
                currentCandidate=[0 for i in range(len(vectorList[:]))]
                        for i in range(0, len(vectorList[:])):
```

```
                                currentCandidate[i]=random.randint(0, len(vectorList[i][:])-1)
                        newCandidateAmount = newCandidateAmount + 1
                        newCandidateList.append(currentCandidate)

                Candidates=newCandidateList

                #Only a certain amount of evaluations are allowed to happen
                if evaluationCount >= maxEvaluations:
                        break

        #PLOTS
        if gaPlot == 1:
        xName = "Iterations"
        yName = "Evaluation"
        title = "Genetic Algorithm"
        k = 1
        historyAxis = [x for x in range(len(History))]
        plot(historyAxis, History, xName, yName, title, k)

        return bestCoordinate

def plot(x, y, xName, yName, title, k):
        #pl.figure()
        if k == 1 :
                color = "black"
                legendName = "1SA"
        elif k == 2:
                color = "green"
                legendName = "2SA"
        elif k == 3:
                color = "red"
                legendName = "3SA"
        elif k == 4:
                color = "blue"
                legendName = "4SA"
        elif k == 5:
                color = "brown"
                legendName = "5SA"
        #pl.figure()
        pl.plot(x, y, color, label=legendName)
        pl.title(title)
        pl.xlabel(xName)
        pl.ylabel(yName)
        pl.legend(loc = 'best')
        #pl.show()



#from IPython.core.debugger import Tracer; Tracer()()
#GENERAL SETTINGS#############################################################################################################################
#Amount of vectors optimized
vectorAmount = 20
#Length of the vectors
xLength = 20
#makeInitialPoints(xLength, vectorAmount)
#Create the list of vectors that are to be optimized
#vectorList = createFunction(vectorAmount, xLength)
```

```
#Read in vectorList
vectorList = []
vectorListFile = open("vectorListFile.txt", "r")
lines = vectorListFile.readlines()

for line in lines:
        line = line.replace('[', '').replace(']', '').replace('\n', '')
        items = line.split(', ')

        ret_array = []
        for item in items:
                ret_array.append(float(item))

        vectorList.append(ret_array)

vectorListFile.close()

#Read in global Max
with open('Global_Max.txt', 'r') as globalMaxFile:
        globalMax=float(globalMaxFile.read())
globalMaxFile.close()

print(globalMax)

#Read in Initial Points
initialPoints = []
initialPointsFile = open("initial_Points.txt", "r")
lines = initialPointsFile.readlines()

for line in lines:
        line = line.replace('[', '').replace(']', '').replace('\n', '')
        items = line.split(', ')

        ret_array = []
        for item in items:
        ret_array.append(int(item))

        initialPoints.append(ret_array)

initialPointsFile.close()

#Use SA in the main loop
useSA = 1
#Use GA in the main loop
useGA = 1
#Use previous best point from GA in SA
usePreviousGAInSA = 0
#Use previous best point from SA in GA
usePreviousSAInGA = 0
#Use PREVIOUS best point from SA in SA
usePreviousSAInSA = 0
#Use preivous best point from GA in GA
usePreviousGAInGA = 0
mainIteration = 200

#SIMULATED ANNEALING SETTINGS
saIteration = 75
saRange = 0.2
```

```
saPlot = 0


#GENETIC ALGORITHM SETTINGS
gaIteration = 1000
maxEvaluations = 200
populationSize = 10
pCrossover = 0.5
pMutation = 0.1
knowMax = 1
parentMortality = 0
gaPlot = 0

#MAIN PART
for k in range(1, 6):

        bestGACoordinate = deepcopy(initialPoints[1])
        bestGA = []
        bestSA = []
        bestCoord = []
        normalizedBestCoord = []#simFunc(vectorList, bestGACoordinate)
        axisForPlots = [i for i in range(0, mainIteration)]
        bestSACoordinate = [0 for k in range(0, k)]
        iterationCount = 0
        bestPreviousSA = 0
        usePreviousGAInSA = 1
        for i in range(0, mainIteration*100):
                if useSA == 1:
                        for j in range(0, k):
                                T = 45
                                bestSACoordinate[j]=simAnn(vectorList,
                                                           T,
                                                           saIteration,
                                                           usePreviousGAInSA,
                                                           bestGACoordinate,
                                                           saPlot,
                                                           bestPreviousSA,
                                                           usePreviousSAInSA,
                                                           saRange)

                                bestCoord.append(simFunc(vectorList, bestSACoordinate[j]))
                                normalizedBestCoord.append(
                                simFunc(vectorList, bestSACoordinate[j])/globalMax
                                )

                        iterationCount = iterationCount + 1
                        if iterationCount >= mainIteration:
                                break

                if iterationCount >= mainIteration:
                        break

                if useGA == 1:
                        usePreviousSAInGA = 1
                        bestGACoordinate=geneticA(gaIteration,
                                populationSize,
                                pCrossover,
                                pMutation,
```

```
                        vectorList,
                        knowMax,
                        globalMax,
                        bestSACoordinate,
                        bestGACoordinate,
                        maxEvaluations,
                        gaPlot,
                        usePreviousSAInGA,
                        usePreviousGAInGA,
                        parentMortality)

                iterationCount = iterationCount + 1

                bestGA.append(simFunc(vectorList, bestGACoordinate))
                bestCoord.append(simFunc(vectorList, bestGACoordinate))
                normalizedBestCoord.append(
                simFunc(vectorList, bestGACoordinate)/globalMax
                )


                if iterationCount >= mainIteration:
                        break

#PRINT BEST GA
#print(axisForPlots)
#print(bestGA)
#plot(axisForPlots, bestGA)
#Plot hybrid algorithm
        xName = "Iterations"
        yName = "Function value"
        title = "Hybrid Algorithm"
        plot(axisForPlots, normalizedBestCoord, xName, yName, title, k)
        pl.show
```