University of Graz

Bachelor thesis (BSc)

Using machine learning for BSM particle identification

Lukas Reicht

supervised by
Axel Maas

May 30, 2018

Abstract

Calculating a certain matrix element yields an exponential function, whose exponent is the mass of a particle. A simulation produces data points that lie on this exponential function, however they are obscured by noise due to the statistical method used in the simulation. The usual approach to fitting such an exponential function through these points would be to use a least squares fit. However it is not clear, if this is the best possible way. I tried to find a better fit by applying a Genetic Algorithm (GA) to the problem.

For this purpose exponential functions with different exponents and noise were created artificially and they were used to test the GA against the standard algorithm. I successfully reproduced previous work on this problem [1] and improved it to a point, where the GA outperforms the standard least squares approach in most domains.

Contents

1 Introduction

The result of a specific simulation are points, which roughly correspond to an exponential function. These points have a high numerical inaccuracy. In this work a Genetic Algorithm (GA) is used to fit an exponential function to these points. An introduction to the underlying physical problem and to GAs is given.

1.1 The physical problem and its simulation

For simple quantum mechanical problems one can obtain a solution analytically or through perturbation theory, but this is not possible for most problems, therefore one needs to run a simulation. A specific simulation yields the matrix element

$$\langle a,0|a,t\rangle$$

This describes the time evolution of 'a'. Here I do not want to specify what exactly 'a' is. It is just formal. Using the time evolution Operator $U = e^{-iHt}$ (H = Hamilton operator) and inserting $1 = \sum_E |E\rangle\langle E|$ we can rewrite this equation:

$$\langle a,0|a,t\rangle = \langle a,0|e^{-iHt}|a,0\rangle = \sum_E \langle a,0|E\rangle\langle E|e^{-iHt}|a,0\rangle = \sum_E \langle a,0|E\rangle\langle E|e^{-iEt}|a,0\rangle = \sum_E |\langle a,0|E\rangle|^2 e^{-iEt}$$

We now replace $t \to i\tau$

$$\langle a,0|a,t\rangle = \sum_E |\langle a,0|E\rangle|^2 e^{-E\tau}$$

If we let τ tend to infinity, we can neglect all energies except the ground state energy E_0. Thus we get:

$$\langle a,0|a,t\rangle \xrightarrow{\tau\to\infty} |\langle a,0|E_0\rangle|^2 e^{-E_0\tau} + \mathcal{O}(e^{-E_1\tau}) \tag{1}$$

In reality we cannot compute the simulation for large enough t to make this approximation valid, because the computational effort rapidly increases - typically with t^5. Without the approximation made in (1) we get a sum of exponential functions. The energies $E_0, E_1, E_2, ...$ are connected to the masses of particles via $E = mc^2$.

Concluding we get a sum of exponential functions out of this simulation.

$$f(t) = a_1 e^{-m_1 t} + a_2 e^{-m_2 t} + a_3 e^{-m_3 t} + ... \tag{2}$$

here m_i are the masses of different particles and a_i are some constants.

Even though we cannot apply the approximation in equation (1) due to limited computational resources, as a first step I still make this approximation and search an algorithm that finds a fit to (3). As an extension to this work, one might assume multiple exponential functions, maybe two or three, like in equation (2). This will be discussed in the outlook in chapter 5.

$$f(t) = ae^{-mt} \tag{3}$$

The constant a can be normalized to 1.

If at some point t_0 the exponential function $f(t_0)$ in (3) is smaller than the numerical inaccuracy, this point and the points with $t > t_0$ have to be discarded, because all their information is lost in the numerical noise.

For this reason large m values result in few points. If m is so large that only 2 points remain, the fit is trivial. Therefore we require that m should be large enough that at least 3 points remain. On the other hand if m is very large, many points arise and the simulation takes very long to compute. If we assess our computational resources optimistically, we can say that any simulation with more than 16 data points takes too long. Thus the number of points lies between 2 and 16. This places an upper and lower bound on m, approximately

$$0.1 < m < 1.8$$

1.2 Introduction to Genetic Algorithms

Genetic Algorithms (GAs) fall under the category of machine learning, which in turn is a subcategory of artificial intelligence.

The 'genetic' in Genetic Algorithm stems from the fact that it is inspired by evolution. There are four preconditions for the occurrence of evolution by natural selection: [2]

1. Reproduction of individuals in the population
2. Variation that affects the likelihood of survival of individuals
3. Heredity in reproduction
4. Finite resources causing competition

This is true for biological evolution as well as GAs. In GAs the programmer defines a so-called fitness function, which determines how fit an individual is. There is a very wide range of possible fitness functions and individuals. In principle they could be anything, that one can write into a computer program. In my case an individual is just a pair of numbers a and b that correspond to the coefficients of an exponential function:

$$f(x) = ae^{-bx} \tag{4}$$

equivalent to equation (3). Throughout this work I will use the terms b and $mass$. They are identical.

The fitness function is subject to change and finding a better one is an integral part of improving the GA. However in the context of this problem it will always be some measure of how close the exponential function is to the data points, that we want to fit. The 'closer' (by some measure) $f(x)$ (4) lies on the data points, the 'fitter' is the individual and has a higher change of reproduction. There is a fixed number of individuals in the algorithm. The result is that the fit individuals reproduce, while the unfit individuals tend to die out.

In essence a GA can be thought of as an optimization process, which tries to optimize the fitness function.

Some important definitions:

- Mutation: a random change to an individual
- Crossover: 'Genes' (a and b in (4)) of two individuals get swapped. This has a similar effect as mutation.
- Generation: is one cycle consisting of reproduction and mutation (see figure 1). The number of generations can be increased, if the GA would need more time to find a good solution. As a trade-off computation time is increased.
- Population: is the set of all individuals

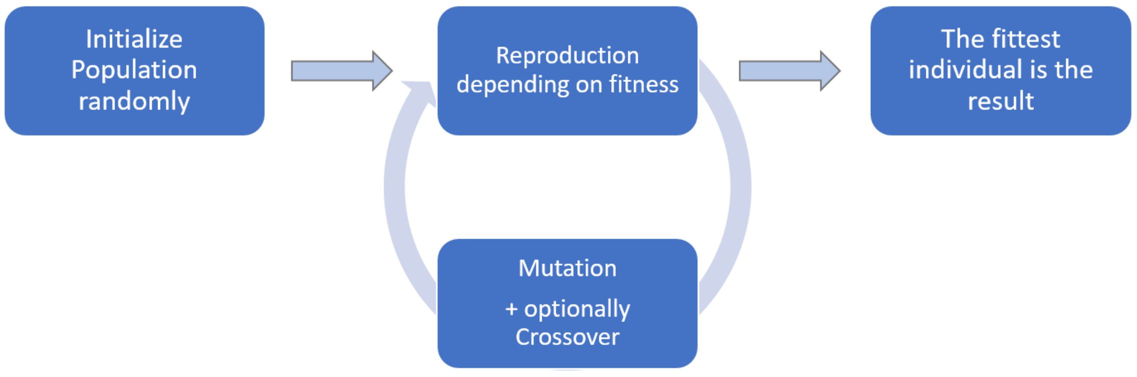

Initialize Population randomly
Reproduction depending on fitness
The fittest individual is the result
Mutation
+ optionally Crossover

Figure 1: The work steps of the algorithm. Reproduction and Mutation are repeated for the defined number of generations.

2 Reproduction of Wagner's genetic algorithm

A first goal of this work is to independently implement Wagner's GA and reproduce the results. The following is an explanation of his algorithm. For more details see [1].

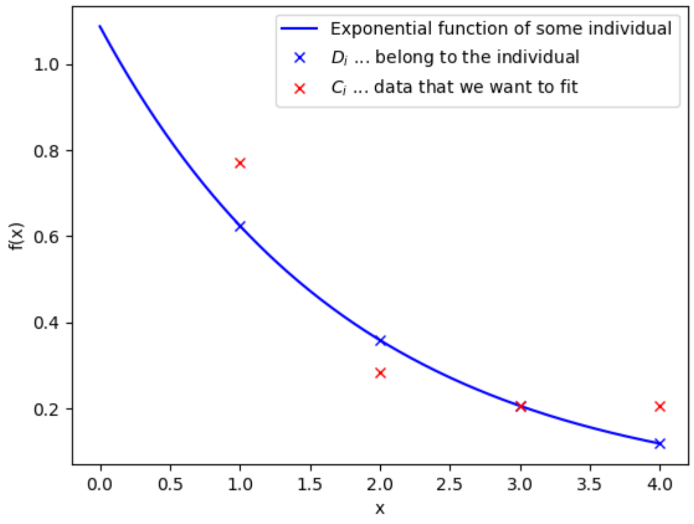


Figure 2: This figure illustrates the quantities in equation (5). A individual has a corresponding exponential function of the form (4) associated with it. Its fitness in Wagner's GA is given by (5). The index i runs from 1 to 4 in this case. The error bars of C_i are omitted in this plot. They would be $\sigma = 0.08$ high.

$\sigma = 0.08$	$\lvert C_i - D_i \rvert$	$\int_0^{\lvert C_i - D_i \rvert} (1 - e^{-\frac{x^2}{2\sigma^2}}) dx$
$i = 1$	0.14564992	0.08320853
$i = 2$	0.07334446	0.01960274
$i = 3$	3.53e-06	2.94e-15
$i = 4$	0.08881284	0.03089033

Table 1: The values associated with figure 2. The integral on the right has the effect that small $|C_i - D_i|$-values are taken into account linearly, while $|C_i - D_i|$-values that are equal or larger than σ have a higher weight in the Fitness. The Fitness is $3.4 * 10^{14}$. It is very high, because $2.94 * 10^{-15}$ is very close to zero.

According to Wagner's GA the fitness of an individual is given by:

$$Fitness_{individual} = \sum_{i=1}^{n} \frac{1}{\int_0^{|C_i - D_i|} (1 - e^{-\frac{x^2}{2\sigma^2}}) dx} \tag{5}$$

with n = number of points, that one wants to fit (in figure 2: $n = 4$).
C_i, D_i according to figure 2
σ is the uncertainty of the points, that one wants to fit.

In principle a properly set up GA should converge to a maximum of this fitness function. In order to achieve that, Wagner set the following parameters, which one could optimize further.

- number of generations = 15
- size of population = 450 individuals
- Coefficients a and b are mutated by adding random normally distributed noise to them. The mean of the normal distribution is 0 and its standard deviation is 0.001 for a and 0.005 for b. If an individual is supposed to get mutated, either its a or b value gets changes - with an equal probability of 50% each.
- The rate of mutation is 1. This means that every individual gets mutated one time per generation.
- no crossover
- The likelihood that an individual gets copied into the next generation is proportional to its fitness.
- The first generation gets initialized with uniformly distributed a between 0.9-1.1 and b between 0.07-1.9. (These values are approximate and are not given in his paper.) Wagner chose these values, because they lie outside the range of the data he tested his algorithm against. The tested data lies within 0.1-1.8 for b and is 1 for a.

  These initialization values are approximate and are somewhat arbitrary, but this is justified because through testing the algorithm I found that varying these values does not change the results. In particular if b is initialized between 0.00001-1.9, the results are the same.

In order to test his algorithm against the standard least squares fit, Wagner created data points that lie on an exponential function and added normally distributed noise to them. He did this for b values (b in equation (4)) between 0.1 and 1.8 with an increment of 0.1 as well as for different standard deviations of the normally distributed noise between 0.0035 and 0.1 with 18 uniformly spaced steps.

The results in a 18 by 18 grid, where each point corresponds to a (standard deviation, b)-pair. For each point he created different random noise many times in order to get rid of statistical fluctuations.

However he discarded some of these points with high b and high standard deviation. Namely those for which $f(x = 3) = e^{3*b}$ is smaller than the standard deviation, because in that case we would only get two meaningful points to fit and that would be a trivial fit.

2.1 Results of successful reproduction

To quantify the quality of an algorithm, Wagner defined the following measures of merit:

$$\text{MM1} = \frac{|m - m_{real}|}{m_{real}} \tag{6}$$

$$\text{MM2} = \frac{\sigma}{m_{real}} \tag{7}$$

where m = resulting mass of the algorithm
(Please note: b from equation (4) is identical to the mass)
m_{real} is the true mass before any noise was added. m_{real} is set to 0.1-1.8 for testing the algorithm.
σ is the error estimate of the algorithm, further discussed in section 2.2. (Note also that this is not the same σ as in equation (5), where it denotes the error estimate of the input data.)

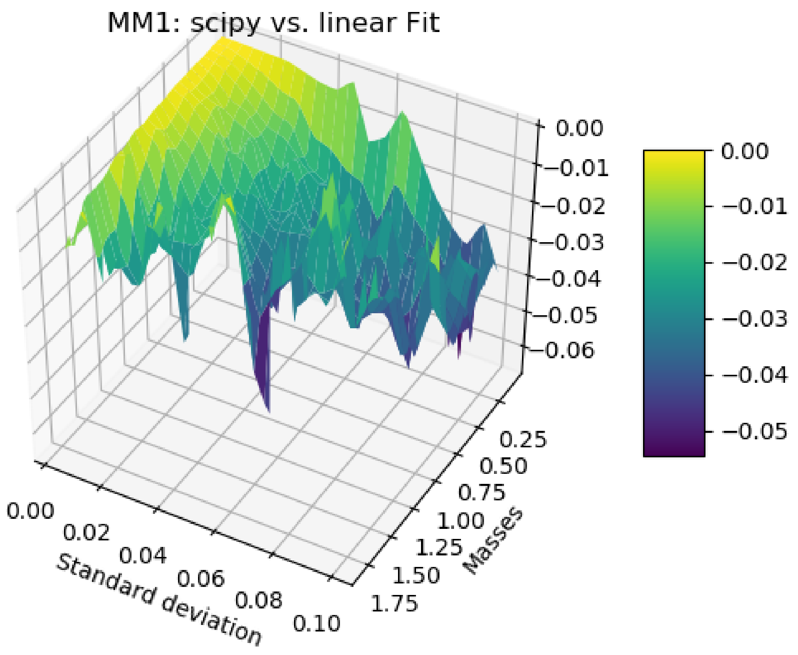
I will from now on refer to equation (6) as Measure of Merit 1 (MM1) and to equation (7) as Measure of Merit 2 (MM2).

MM1 and MM2 of the GA are compared to those of a standard method. I compared two standard methods:

- **Linear least squares**: applying the $ln()$ to both sides of equation (4) results in a linear equation that can be solved by a least squares fit. The error estimate of this fit is calculated by a least squares fit through the upper and lower point of the error bar of the inputs and calculating the difference of the result, just like was done for the GA in section 2.2.2.

- **Scipy**: the exponential function can be handed to the *scipy.optimize.curve_fit* routine, which uses a Levenberg-Marquardt algorithm.

A comparison of these two standard methods is given in figures 3 and 4. Both standard methods yield similar results. I chose the least squares method, because Wagner chose it as well and it is a more 'classical' approach. To compare two algorithms, their measures of merit are subtracted from each other. An algorithm is better if its measure of merit is small.



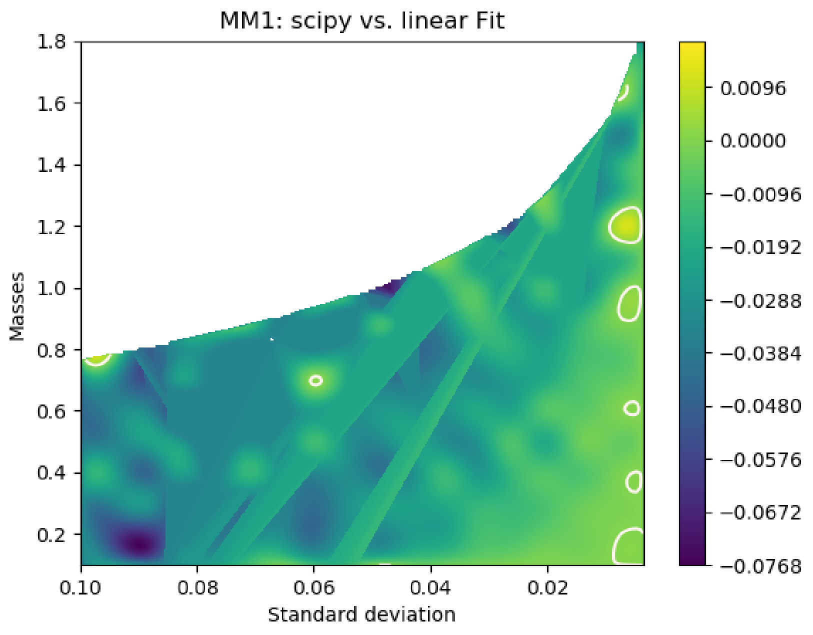
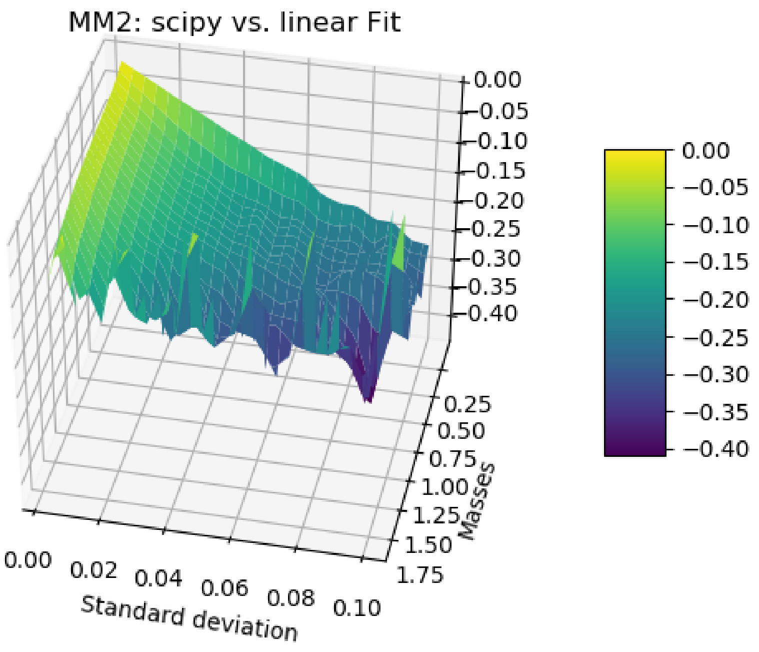


Figure 3: $MM1_{scipy} - MM1_{linearFit}$ is plotted. The linear least squares Fit is better where the value on the z-axis is above zero. Scipy is better by an negligible amount in MM1. Interpolation like in figure 5 was used, except that linear interpolation was used instead of spline interpolation.



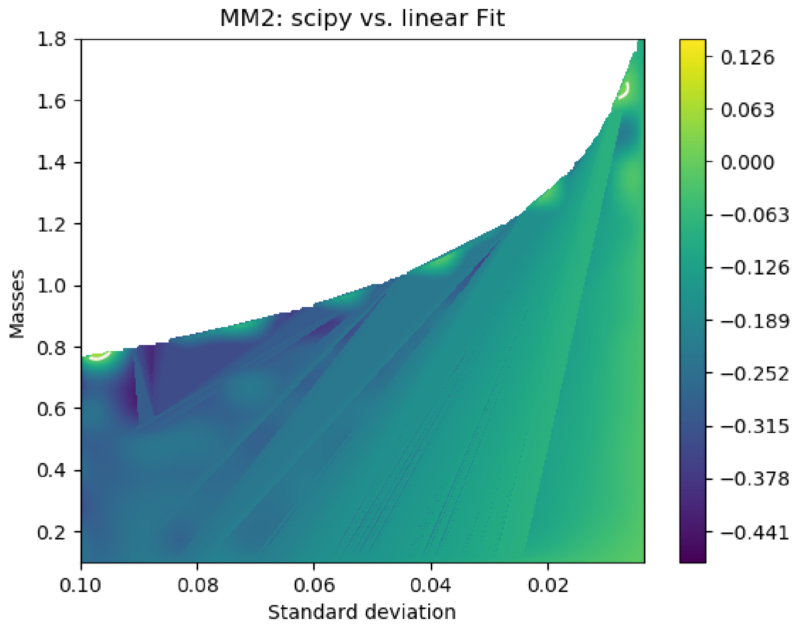
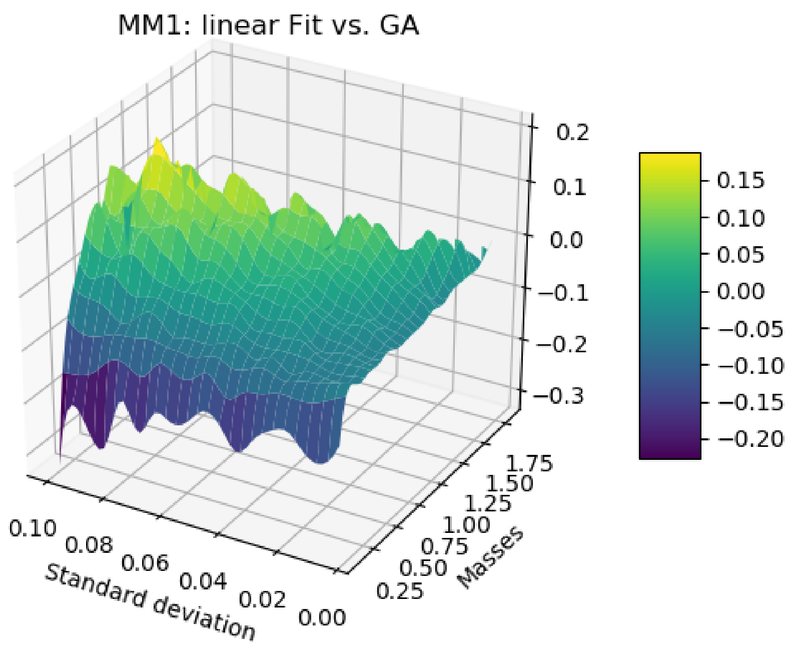


Figure 4: $MM2_{scipy} - MM2_{linearFit}$ is plotted. The linear least squares Fit is better where the value on the z-axis is above zero. Scipy is better in MM2, but this is probably because it underestimates the error. m_{real} is only in 55% of the cases within $m_{scipy} \pm error_{scipy}$ (should be more than 67%), while it is within $m_{linear} \pm error_{linear}$ in 88% of the cases. The same interpolation as in the previous figure 3 is used.



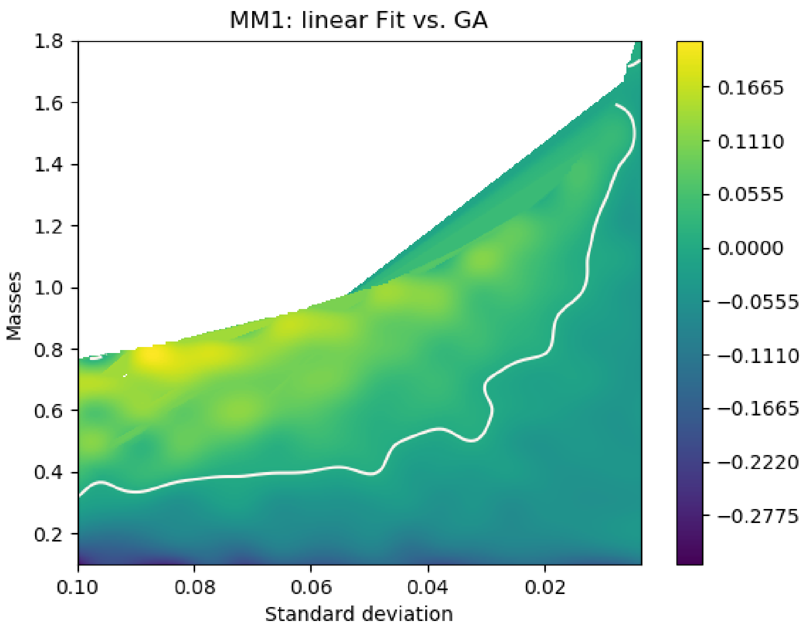
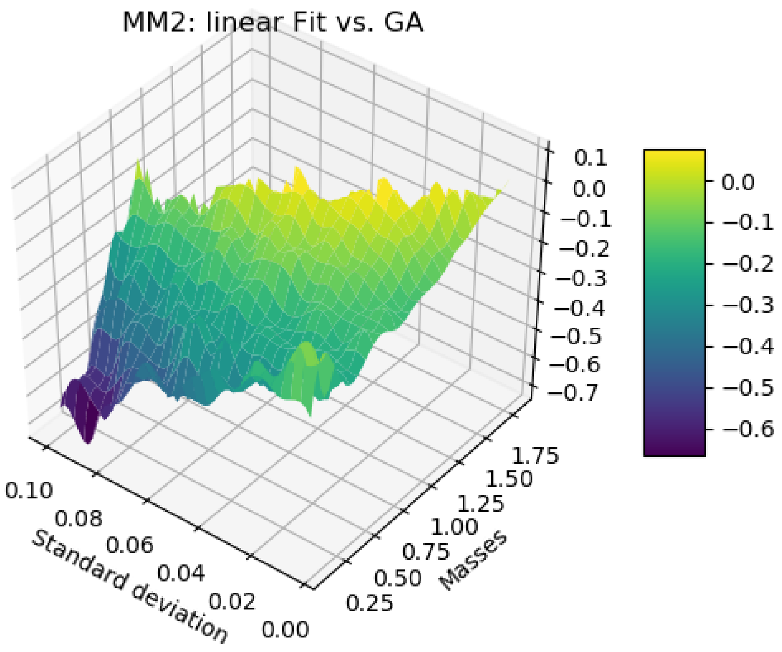


Figure 5: Successful reproduction of Wagner's Work for Measure of Merit 1. $MM1_{linearFit} - MM1_{GA}$ is plotted. The GA is better where the value on the z-axis is above zero. For the 3D plots spline interpolation with 3 times as many points on each axis were used. For the contour plots spline interpolation with 10 times as many points on each axis were used. This setting is used for all of the following interpolated plots as well.



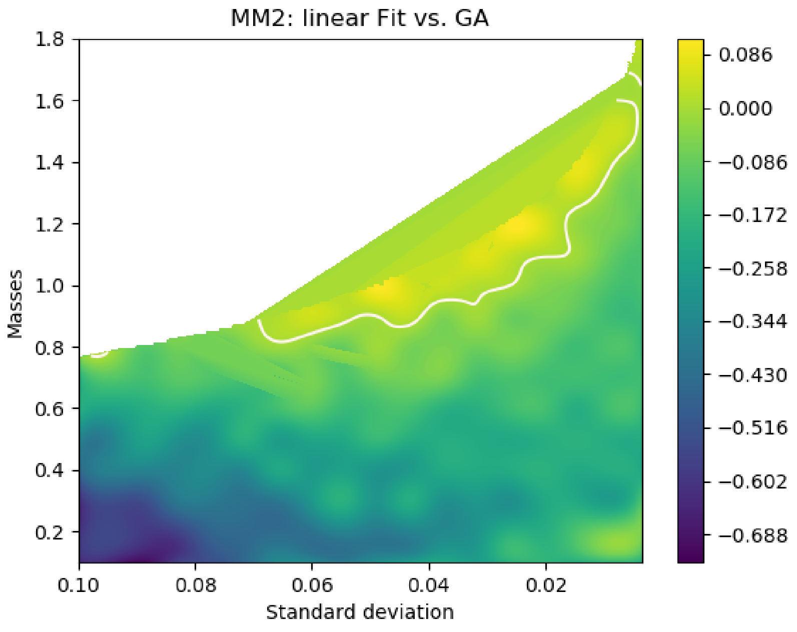


Figure 6: Successful reproduction of Wagner's Work for Measure of Merit 2. $MM2_{linearFit} - MM2_{GA}$ is plotted. The GA is better where the value on the z-axis is above zero.

For each of the points in the above plots, random inputs were created 160 times. It took about 24 minutes to produce the data of the GA for one plot on a common PC. The results in figure 6 are not exactly the same as Wagner's, because a different error estimate was used, as discussed in the next section 2.2.

2.2 Error estimate of the GA

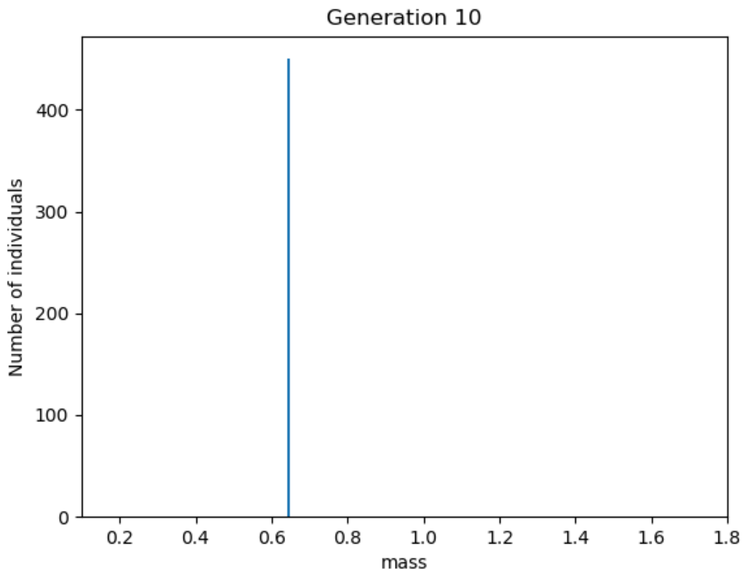
To obtain an error estimate for the algorithm, Wagner applied his algorithm about 50 times on the same data, and calculated the mean and the standard deviation of the algorithm's results. The mean was his final result and the standard deviation was his error estimate of the algorithm.

However this method is computationally expensive. It would be desirable to run the algorithm just once and getting an error estimate out of just this one run, effectively decreasing the computation time by a factor of about 50.

2.2.1 First approach: range of masses

As a first attempt to achieve this, I tried to infer an error estimate out of the range of masses of the population. (An illustration of 'range of masses' is given in figure 7.) This can be done by sorting the final generation by their masses, then taking an individual, whose mass is greater than 17% of the other individuals, and taking an individual, whose mass is greater than 83% of the other individuals, and subtracting the masses of the two individuals. Choosing 17%-83% is a natural approach, because the error estimate σ corresponds to a 67% chance that the true solution m_{real} lies within the range $m \pm \sigma$.

However I found that this error estimate is most of the time zero, which is bad for obvious reasons. I chose 10%-90% instead of 17%-83% to improve the situation, but still the error estimate was zero is about 86% of the cases. In order to resolve this problem, I changed the parameters of Wagner's GA, discussed in section 2, to make the range of masses wider. This can be achieved by taking the root of the fitness for each individual, as illustrated in figure 7. For example when taking the fourth square root, the error estimate is zero in only 4% of the cases.



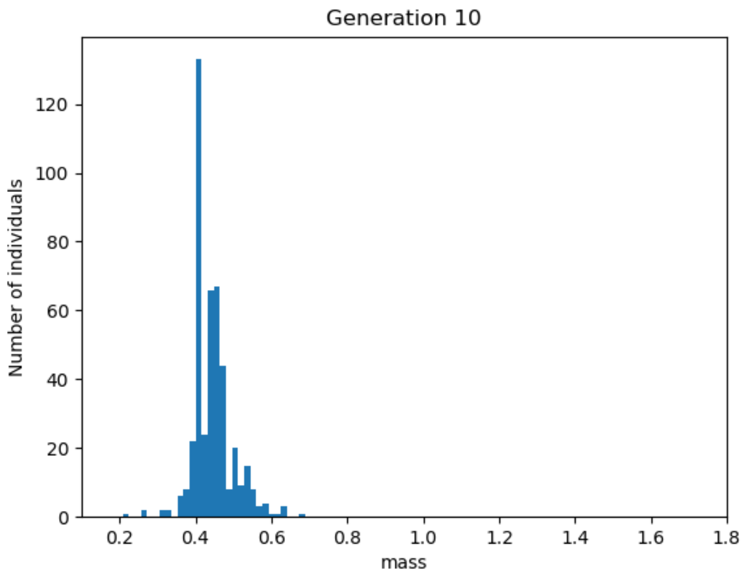


Figure 7: An illustration of the range of masses. On the upper plot there is an illustration of the mass width of Wagner's GA. The population is very concentrated on a small range of masses. For the algorithm in the lower plot the mutation width is 10 times as high and the fitness is taken to the power of $\frac{1}{10}$. Further discussion and a definition of 'mutation width' is given in section 3.1.

For these results, that I just discussed, the last action before the measurement of the error estimate was reproduction, like in figure 1. It is possible to mutate the population once again and measure the error estimate right after the mutation. In this case the error estimate is never zero, however this comes at the cost that it is dependent on the parameters of the mutation. This is not a desirable property, so this method was discarded.

When plotting the error estimate for different standard deviations of noise and masses, like it was done in figure 3-6 for MM1 and MM2, one can see that the error estimate is equal for all values of the standard deviation of noise (see figure 8). This shows that the error estimate does not reflect the real error, because the latter will clearly be greater for large noise. For this reasons this error estimate was regarded as not useful.

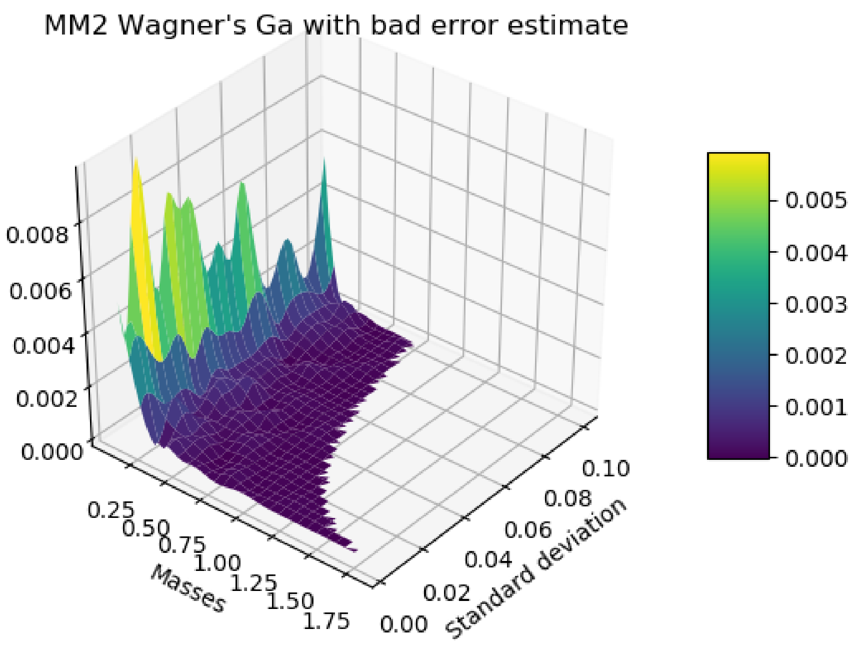


Figure 8: Wagner's GA with the error estimate discussed in this section 2.2.1.

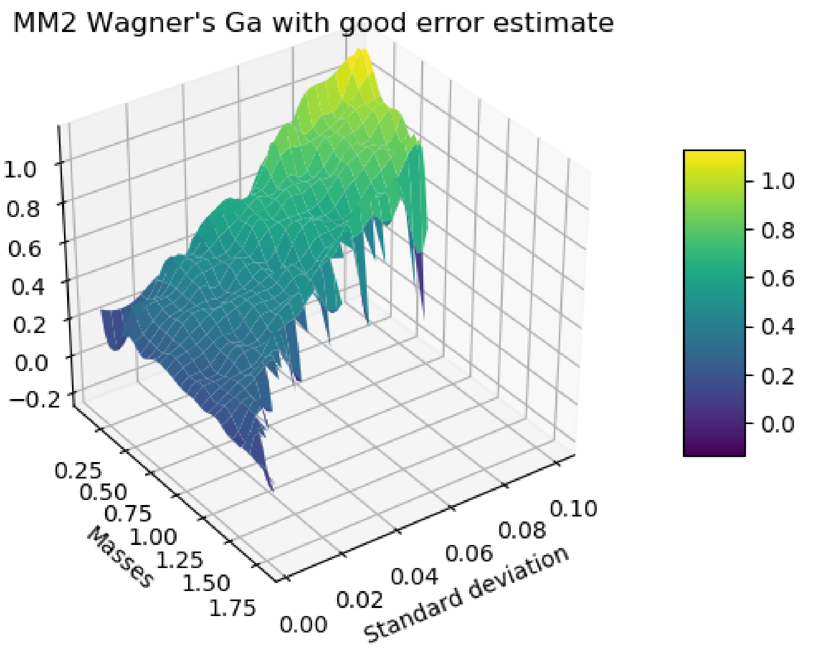


Figure 9: Wagner's GA with the error estimate discussed in the next section 2.2.2.

2.2.2 Second approach: upper and lower error bar

For this error estimate the GA is applied on $input + uncertainty_{input}$ and $input - uncertainty_{input}$ and the uncertainty of the GA is defined as the difference of these two results. This method requires two times more computation as would be needed to run the algorithm without an error estimate.

This error estimate looks realistic (see figure 9). It has the desired property of being higher for larger noise. In addition it is rather pessimistic, usually overestimating the error. This error estimate was chosen.

3 Improvements to the GA

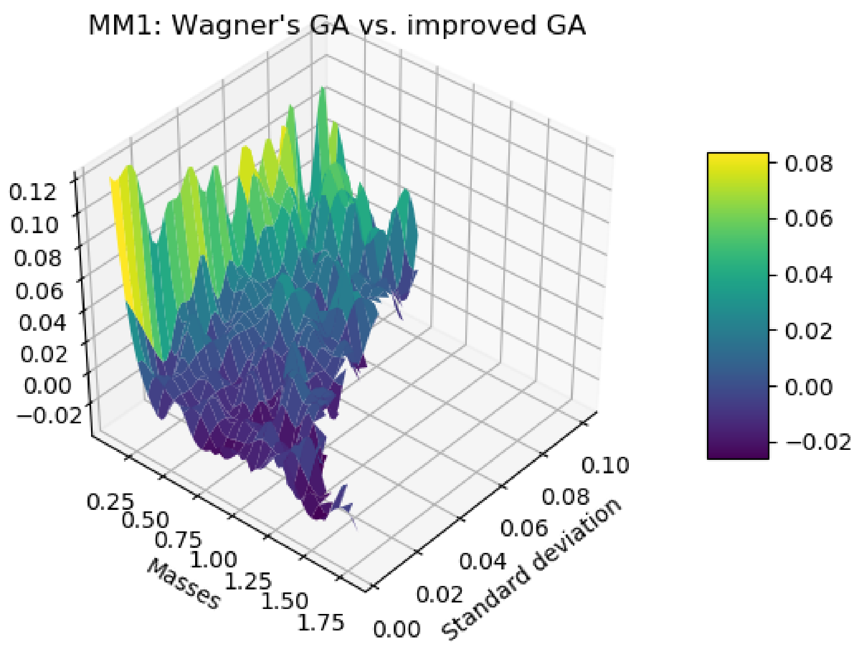
Subsequent to the reproduction of Wagner's results, I tried to improve his algorithm. For comparison the MM1 (measure of merit) and MM2 of the original algorithm was plotted against the (potentially) improved algorithm, like it was done in figure 5 for the linear Fit method. Ideally one would like to improve both measures for a wide range of masses and standard deviations of noise for as much as possible. In reality there will likely be a trade-off between MM1 and MM2 or there might only be an improvement for some area of the plot.

3.1 Improvements to the GA with unchanged fitness function

As a first step I changed parameters of the GA while leaving the fitness function the same. The fitness function defines *what* should be optimized, whereas all the other parameters determine *how* the algorithm should achieve this. When these parameters are chosen poorly, a GA might quickly fall into a local minimum or take too long to converge to a solution. Fast convergence is a big issue for computationally intensive GA, however this is not the case with this algorithm at hand.

Therefore I focused on the former issue. For example a too strong selection pressure might force the algorithm in a local minimum. If the difference in fitnesses in a population is not so high, the algorithm will take longer to converge on a specific solution, but it will explore more of the solution space. The solution space is the space of all possible solutions.

One way to achieve this is to take the logarithm or the square root of the fitness. Another way is to increase the standard deviation of the Gauß distribution used to mutate the individuals, to which I may now refer as 'mutation width'. A large mutation width increases the time it takes the GA to converge, because a strong mutation annihilates the fit individuals.



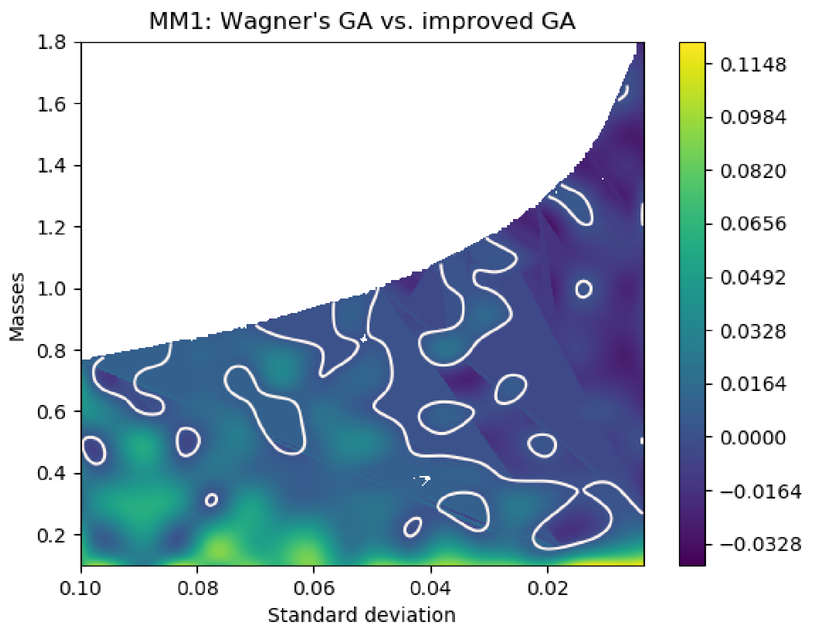
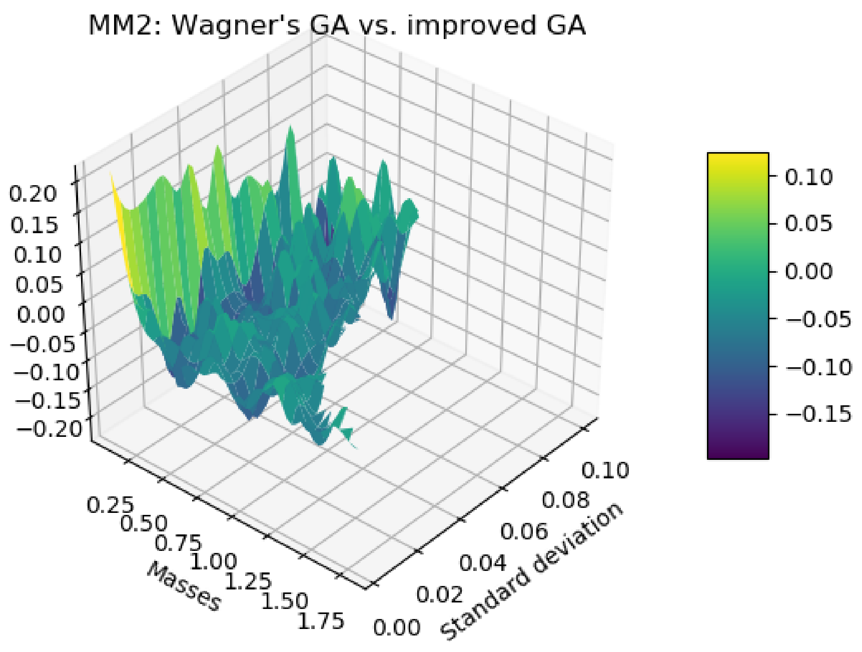


Figure 10: MM1: The 'improved GA' is Wagner's GA with fitness to the power of $\frac{1}{4}$ and mutation width times 4. These numbers were chosen somewhat arbitrary and could definitely be improved. This was not done, because the changes to the fitness made in section 3.2 result in a much better algorithm. Therefore optimizing the parameters for this fitness function became unnecessary. The white line indicates $z = 0$. Wagner's GA is better where $z < 0$.



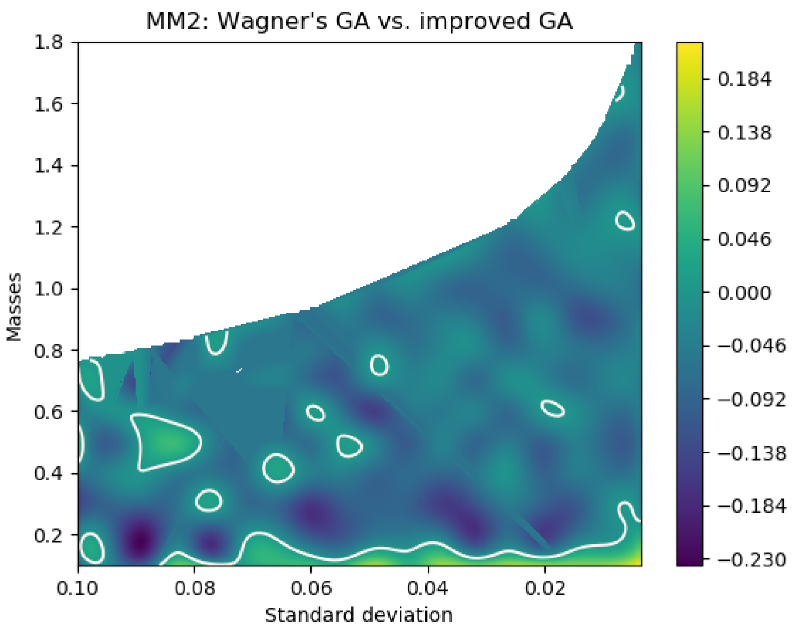


Figure 11: MM2: The 'improved GA' is Wagner's GA with fitness to the power of $\frac{1}{4}$ and mutation width times 4. The white line indicates $z = 0$. Wagner's GA is better where $z < 0$. The improved GA is better where $z > 0$.

In figures 10 and 11 one can see that some improvements can be made by letting the algorithm converge slower. For mass = 0.1 the MM1 of the new algorithm is better, however the average is approximately the same. MM2 of Wagner's GA is better in most areas. This still can be regarded as an slight improvement, because one could combine Wagner's GA and this algorithm by estimating the true mass and applying this algorithm only for small masses.

The validity of figures 10 and 11 were verified by applying the algorithm and Wagner's GA on different randomly created input data. The introduction of crossover did not change the results in figures 10 and 11.

3.2 Changes to the fitness function

The most potential for improvement lies within the fitness function. In Wagner's work the fitness function is given by (5).

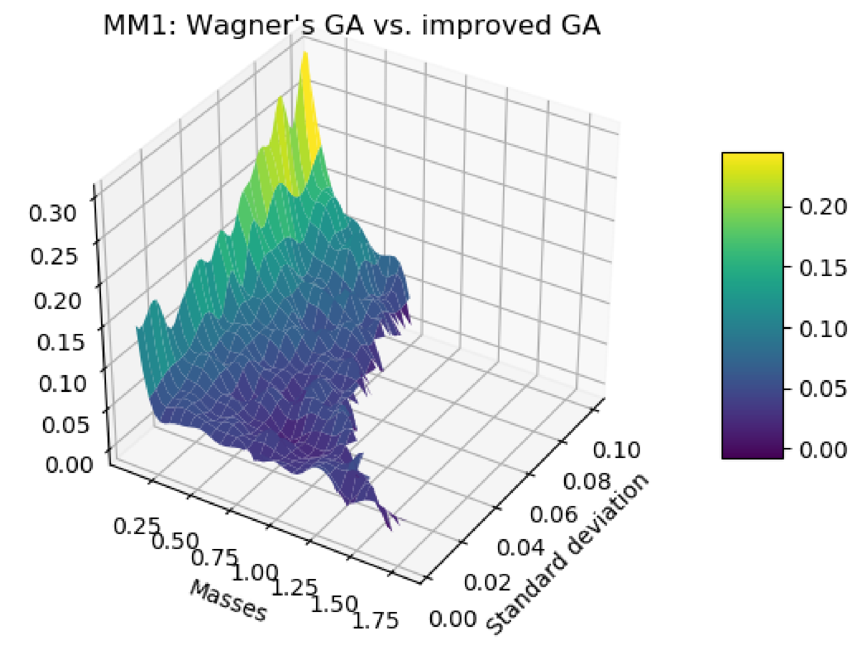
3.2.1 Moving the summation into the denominator

I changed Wagner's fitness function to

$$Fitness_{individual} = \frac{1}{\sum_{i=1}^{n} \int_0^{|C_i - D_i|} (1 - e^{-\frac{x^2}{2\sigma^2}}) dx} \tag{8}$$

With Wagner's fitness function it is beneficial for an individual to fit just one point very close. If it does so, its fitness approaches infinity as the distance to the point goes to zero. This resulted in some individuals having a much higher fitness than the rest of them and therefore dominating the reproduction process.

Moving the sum into the denominator has the effect, that individuals focus more on fitting all points instead of just one. This also makes the algorithm converge slower, because the difference in fitnesses in the population are not as dramatic as in Wagner's case.



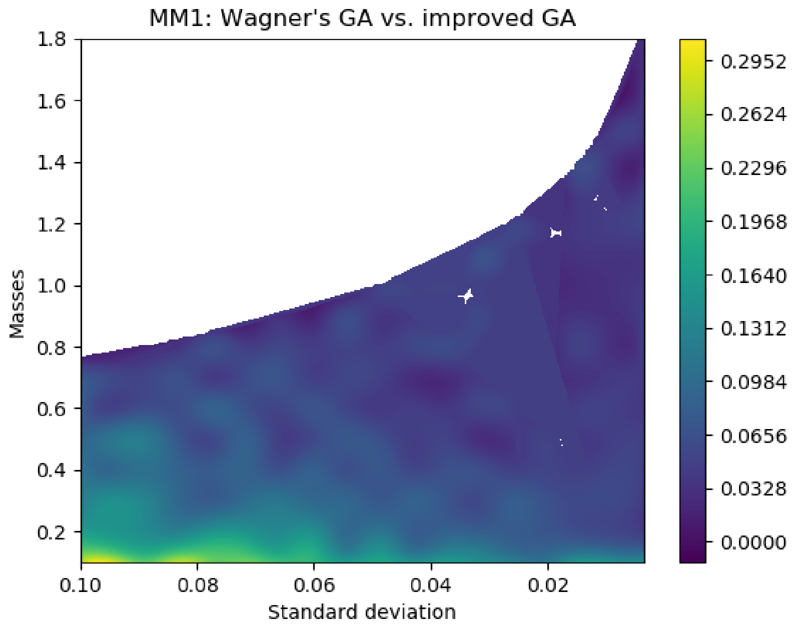
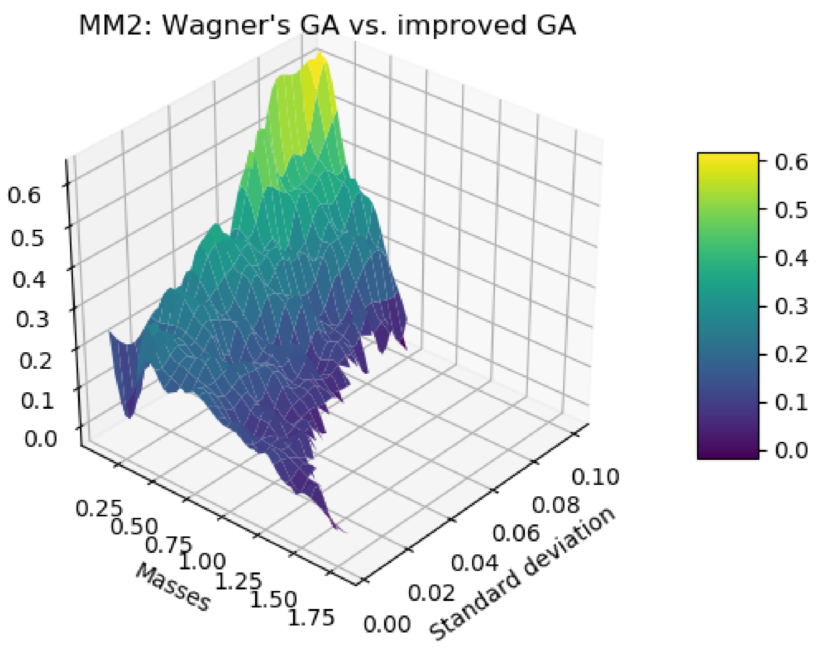


Figure 12: MM1: Improvements in almost all domains can be made by changing the fitness function from (5) to (8). In areas where the value on the z-axis is above zero, the improved GA is better. Spline interpolation like in figure 5 is used.



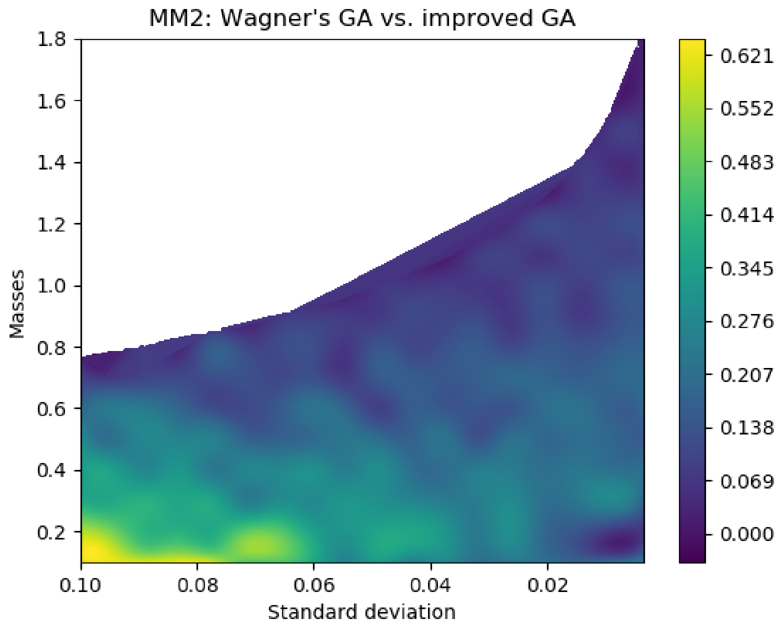


Figure 13: MM2: Improvements in almost all domains can be made by changing the fitness function from (5) to (8). In areas where the value on the z-axis is above zero, the improved GA is better.

The input data for Wagner's GA and the improved GA were independently created with random time based seeds. The whole process was repeated with different random input data and the results were the same.

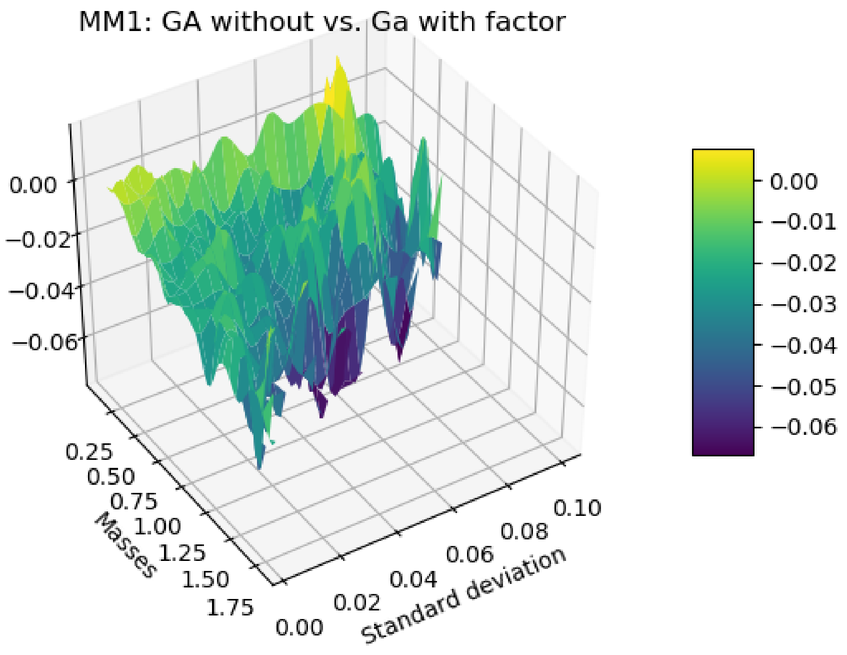
3.2.2 Weight for relative uncertainties

The absolute uncertainty of the input data $y(x)$ is the same for all x-values, however the value of the exponential function decreases for higher x. For this reason the relative uncertainties of the input data increase with increasing x.

A factor was added to account for this issue.

$$Fitness_{individual} = \frac{1}{\sum_{i=1}^{n} \frac{\sigma}{D_i} \int_0^{|C_i - D_i|} (1 - e^{-\frac{x^2}{2\sigma^2}}) dx} \tag{9}$$

Here D_i is the value of the individual at point i, as can be seen in figure 2. The σ in the factor of (9) can be drawn in front of the summation. Therefore it can be omitted, because it just multiplies a constant to all individuals and that does not change the relative fitness values in the population.



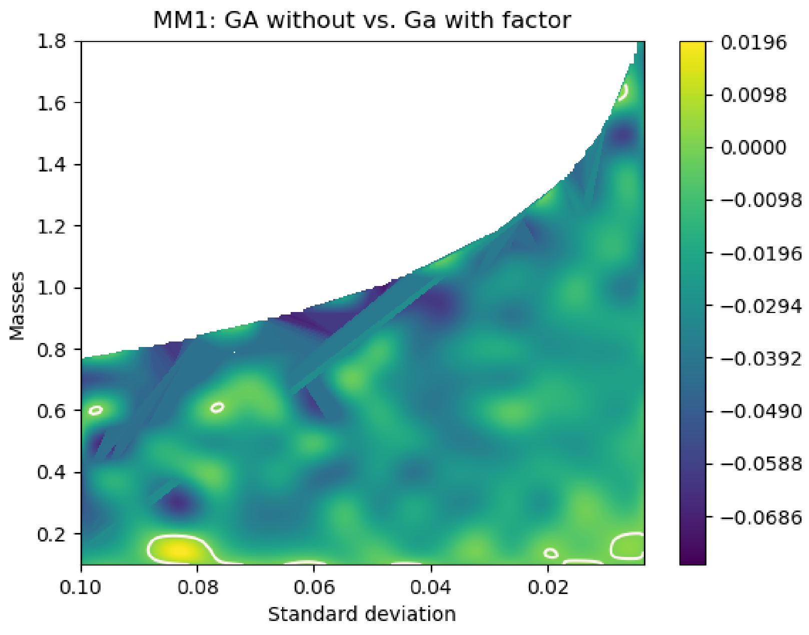
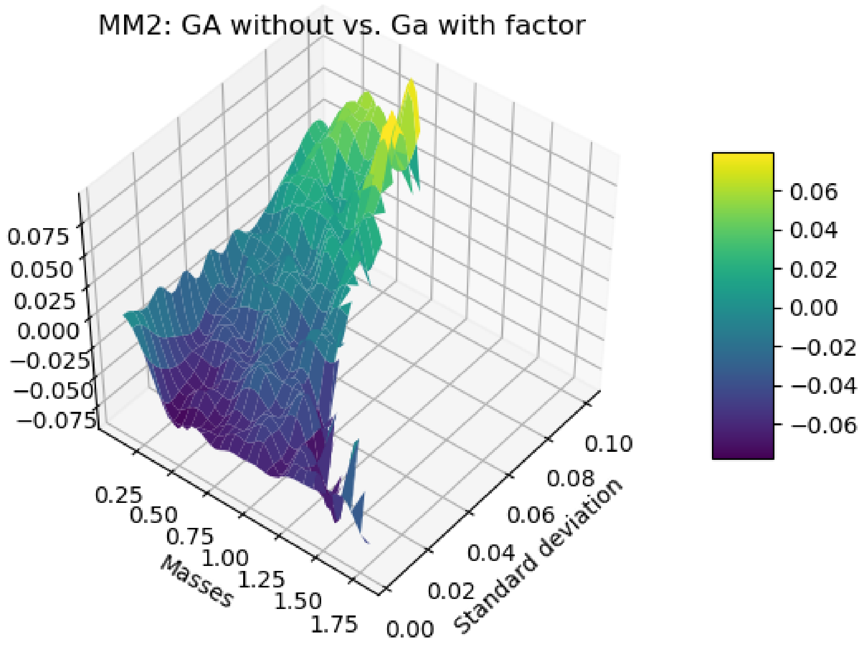


Figure 14: MM1: Comparison of algorithms with fitness function like (8) and (9). The GA without the factor is better in almost all domains for MM1.



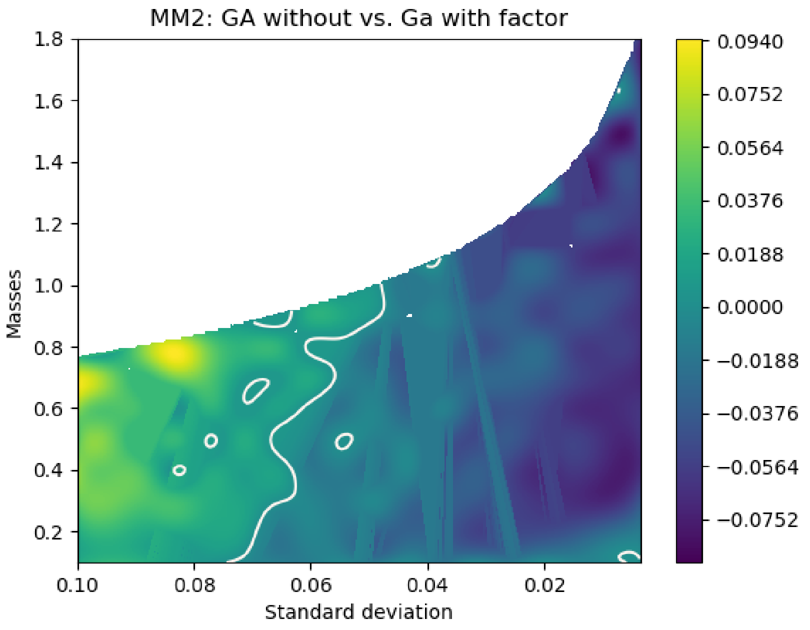


Figure 15: MM2: Comparison of algorithms with fitness function like (8) and (9). Two regions emerge, where the respective algorithms are better. Overall MM2 gets worse with the addition of the factor.

Figures 14 and 15 show that the algorithm gets worse if a factor like in (9) gets added.

3.3 Further optimizations

In this section the Fitness function (8) is chosen as a baseline and other parameters are optimized in the hope of further improving the GA.

To make this task feasible, I did not optimize all parameters simultaneously, but rather just one at a time. Wagner's values are kept, while one parameter gets optimized. In addition only one point of the plot 12 gets optimized. The point of the optimization is chosen to be at

(mass = 0.5, standard deviation = 0.05),

because it lies in the middle of the plot.

For each point in the following plots, random inputs are created 3000 times to reduce numerical noise. In contrast to create all above plots random inputs were only created 80 or 160 times for each point.

3.3.1 Number of generations

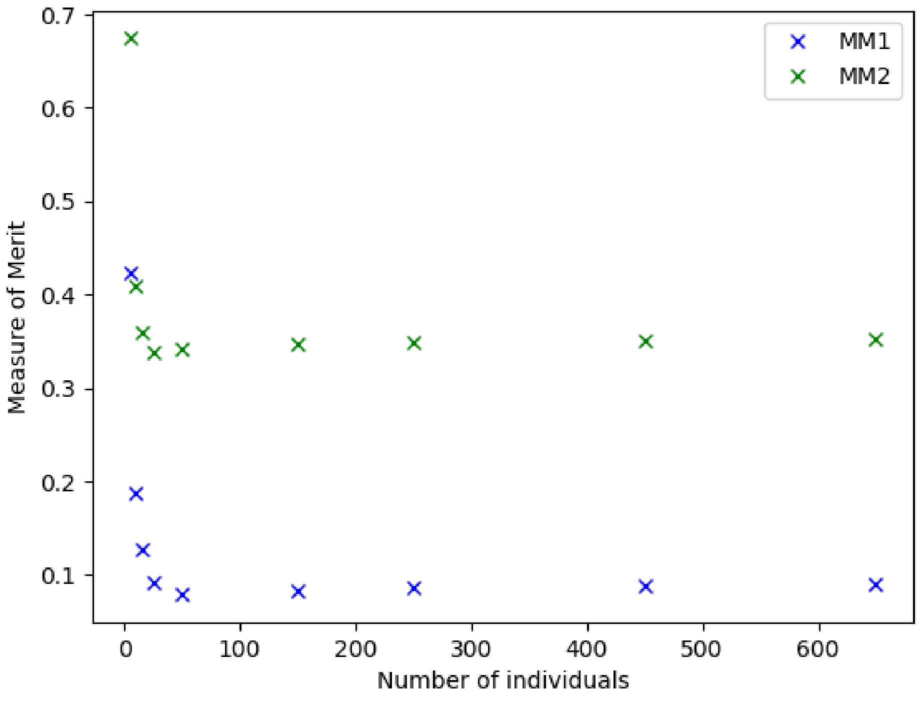


Figure 16: Optimization of the number of individuals

If the number of individuals is very small, the algorithm cannot find a good solution. If the size of the population exceeds a certain threshold, making it larger does not affect the results.

3.3.2 Fitness to the power of x

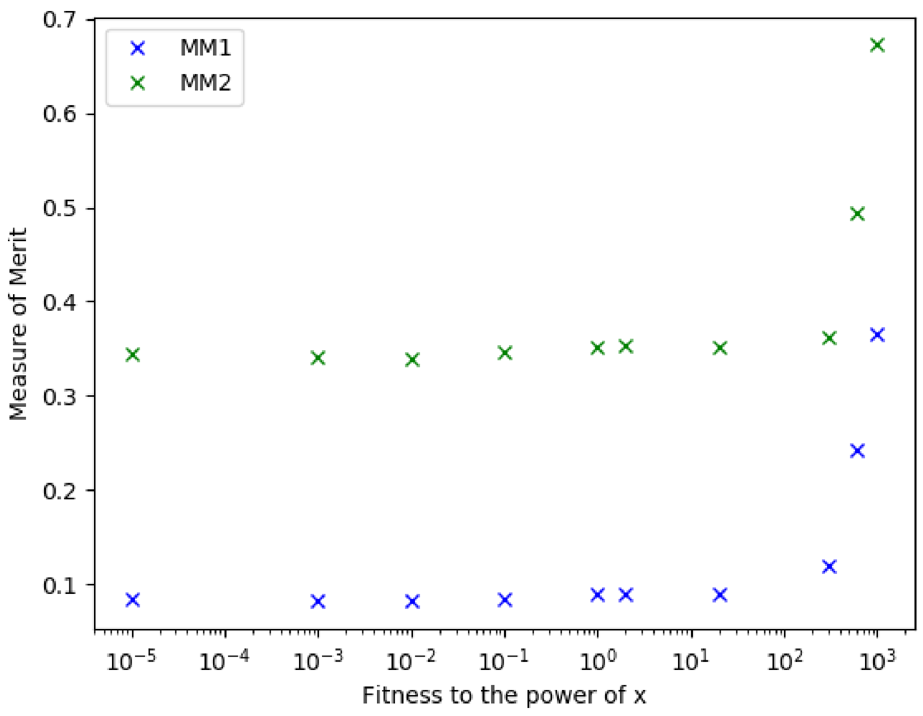


Figure 17: After the fitness of an individual is evaluated as usual, it is taken to the power of x. Note the logarithmic x-axis.

Taking the fitness to the power of x with $x > 1$ has the effect that the algorithm converges faster and the fit individuals dominate the reproduction process more. On the other hand a x with $0 < x < 1$ has the effect that the chance of reproduction for the unfit individuals is increased relative to the situation with $x = 1$. This has the effect of slower convergence.

For a large exponent the fit individuals dominate the population too much and the results are worse, because the algorithm falls into a local optimum too quickly. For a very small exponent the GA needs very many generations to converge. Because the number of generations was fixed, the algorithm could not converge and after the 15 generations the population was still essentially random. The results are nevertheless just as good as for the original version of the GA, because a random population gives very good results already, as discussed in section 3.3.5.

3.3.3 Mutation width

In the mutation routine a normally distributed number is added to either a or b (a and b as defined in (4)). The mean of the normal distribution is zero, while its standard deviation is referred to as 'mutation width'.

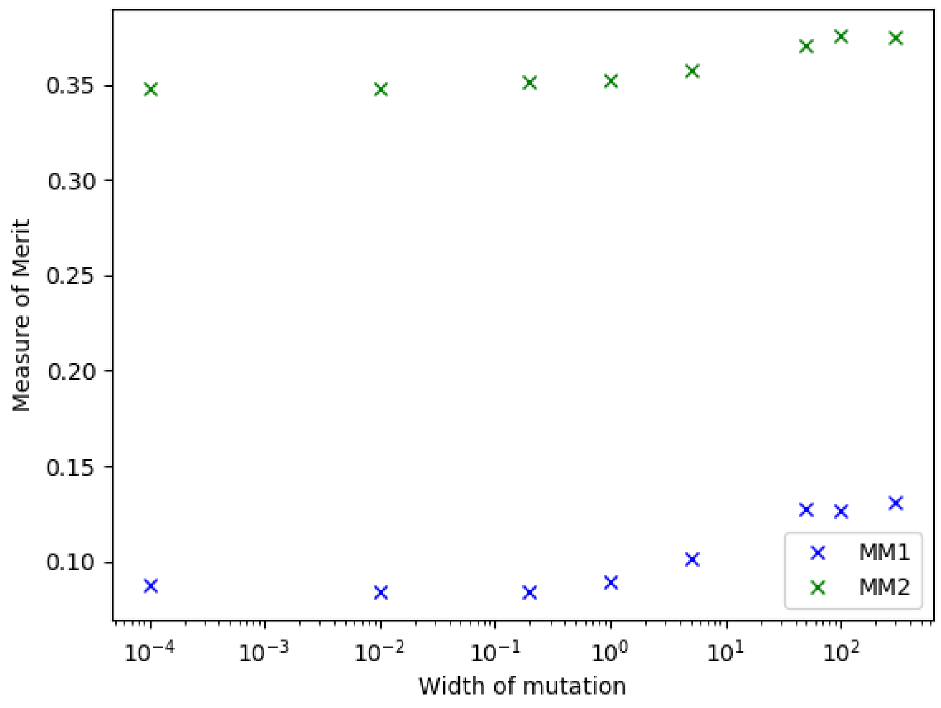


Figure 18: Optimization of the mutation width. Note the logarithmic x-axis.

3.3.4 Weight for relative uncertainties to the power of X

Even though the prefactor in (9) makes the algorithm worse, it might be possible that the factor to the power of some number X makes the algorithm better.

$$Fitness_{individual} = \frac{1}{\sum_{i=1}^{n}(\frac{1}{D_i})^X \int_0^{|C_i-D_i|}(1-e^{-\frac{x^2}{2\sigma^2}})dx} \tag{10}$$

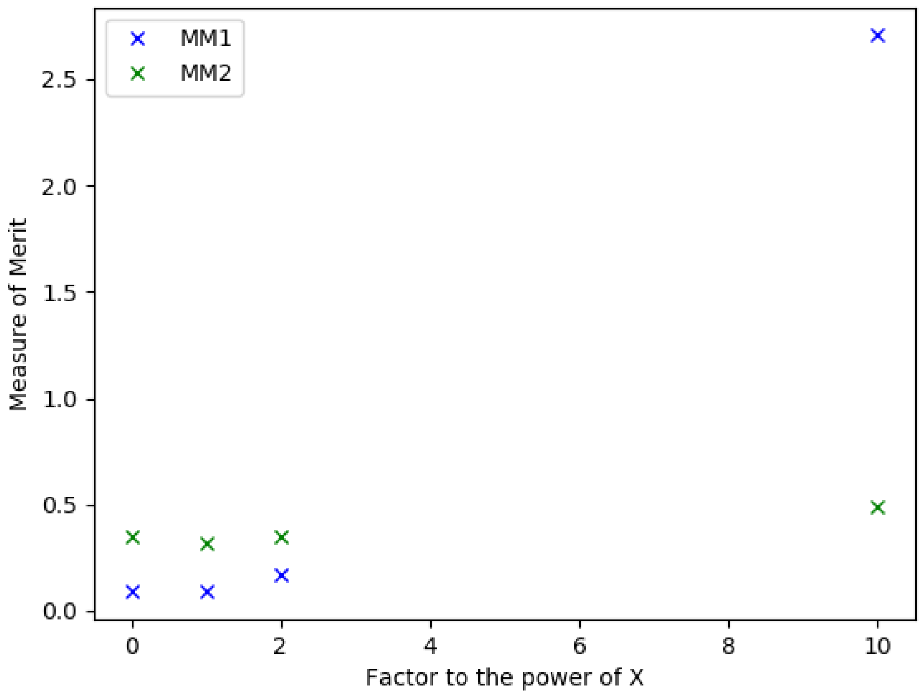


Figure 19: Optimization of the capital X in (10). The different values for X are drawn on the x-axis of this figure. When $X = 0$ the factor disappears and the GA is just the baseline GA, that I want to optimize.

3.3.5 Mutation rate

Wagner chose his mutation rate to be equal to 1. This means that every individual gets mutated once per generation. An mutation rate of 0 corresponds to no mutation at all.

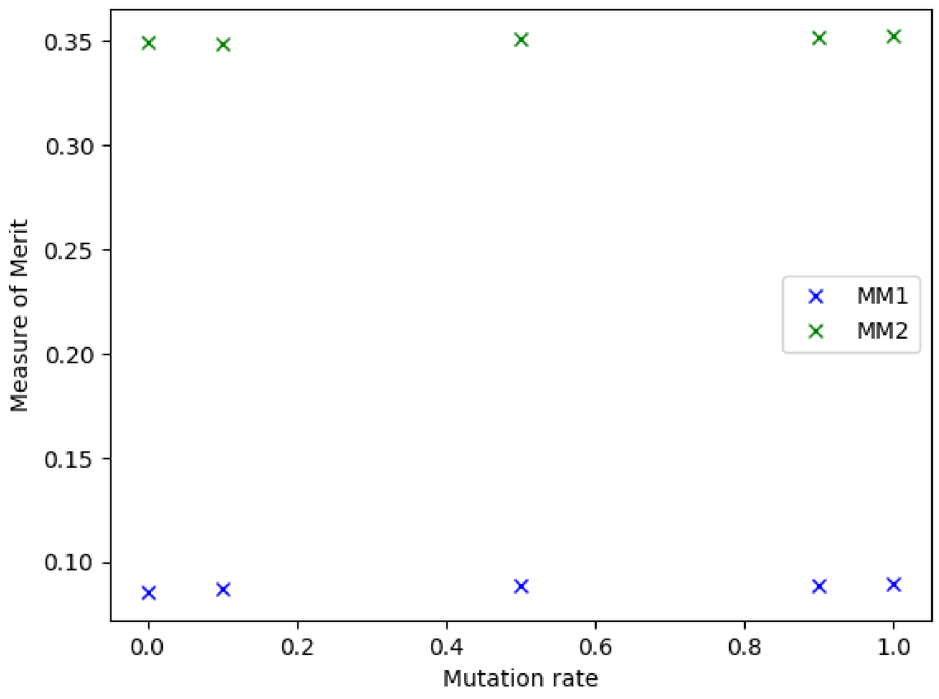


Figure 20: Optimization of the mutation rate.

At first it seems surprising that the Measure of Merits are the same if no mutation occurs. This observation means that the very first generation, which is initialized randomly, already contains a solution, which is so good that mutation only improves it by an imperceptible amount. This is in fact the case and will now be explained.

Upon initialization 450 individuals are created by an equal distribution over the interval 0.1-1.8. Therefore the distance between the masses of two individuals is on average

$$\frac{1.7}{450} \approx 0.00378$$

On the other hand the input data has a uncertainty of 0.0035-0.1. Thus the error of any algorithm will be of this size at the minimum. In the case here we set the uncertainty of the input data to be 0.05. In figure 20 we see MM2 = 0.35. The estimated error of the GA in this case is:

$$\sigma = MM2 * m_{real} = 0.35 * 0.5 = 0.175$$

We see that the population is simply so large that a good solution is created by chance in the random initialization. The error caused by this random spacing is about 0.00378. The general error of the algorithm, with or without mutation, is much larger than that.

$$0.175 >> 0.00378$$

Therefore the error inherent in the GA overshadows the smaller error caused when no mutation is applied. Hence the results in figure 20 can be explained. If one would create many more data sets and run the GA on all of them, we would get a result with less statistical fluctuation and we would probably see a small decrease in quality when no mutation is applied.

Even though mutation is redundant in this GA, it will become very important when one considers the more complex problem of multiple exponential function, as discussed in the outlook 5. In this scenario guessing a good solution by random initialization quickly becomes unlikely due to combinatorics. We would have to guess m_1 and m_2 correctly at the same time.

3.3.6 Crossover rate

Crossover is inspired by nature. The basic idea behind it is, that individuals with 'good genes' can spread them across the population.

Individuals participate in crossover with some probability called 'crossover rate', analogous to the mutation rate. A randomly chosen individual swaps either its a or b value with another randomly chosen individual. These two individuals have a 50% chance of swapping both a values and a 50% chance of swapping both b values, but never swap an a with a b value.

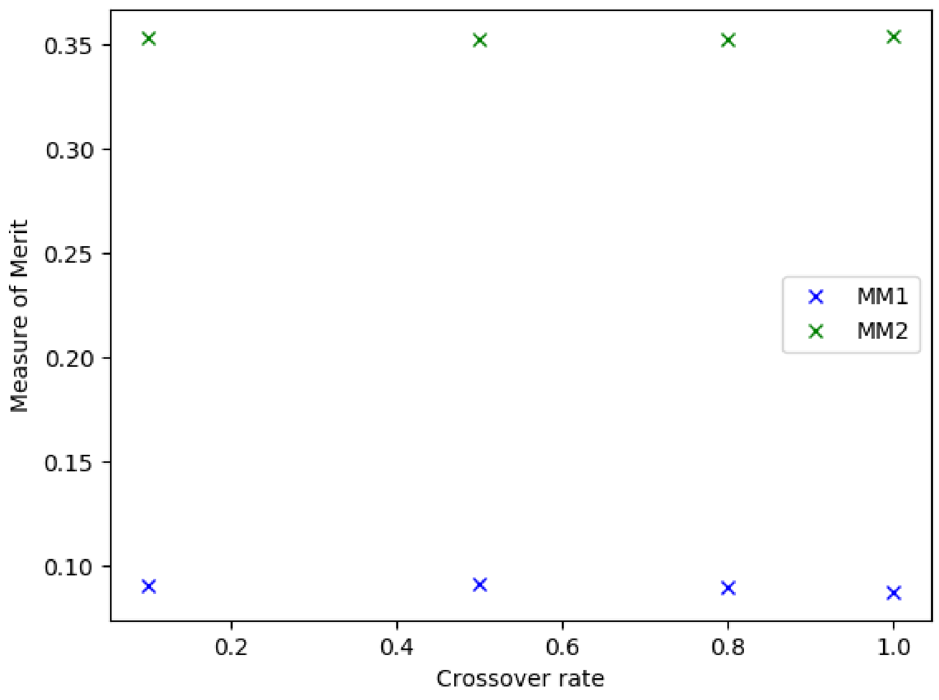
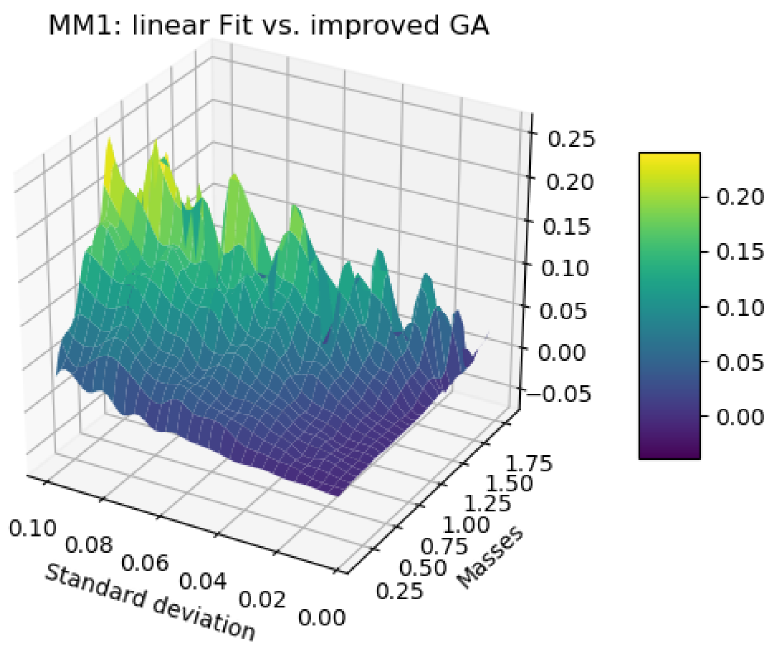


Figure 21: Optimization of the crossover rate. Mutation rate is chosen to be 0.5.

4 Results

Through the optimizations in section 3.3 it was not possible to improve the algorithm. The results were for a wide range of different values of 'number of individuals', 'fitness to the power of x' and 'mutation width'. Changing the crossover and mutation rate did not influence the result of the algorithm. No improvement could be made through the factor, which was meant to account for the relative uncertainties of the input data.

However the minor change of moving the summation of the fitness values into the fraction, like equation (8), improved the result dramatically.



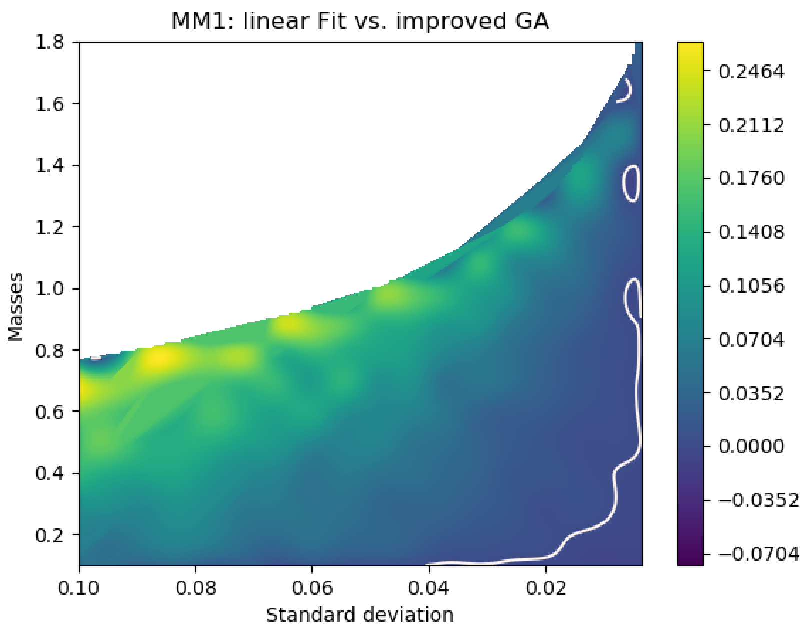
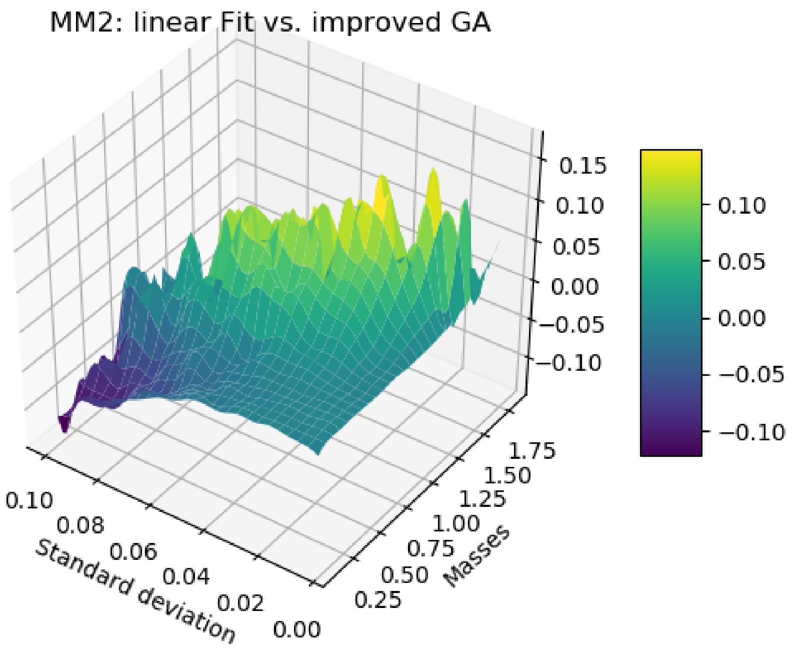


Figure 22: MM1 Comparison of the improved GA versus the linear Fit method. The GA is better where the z-value is above zero. The only difference to Wagner's GA is that the fitness value is determined like (8). A white line is drawn at $z = 0$ to distinguish the areas where the two algorithms are better. Spline interpolation like in figure 5 is used.



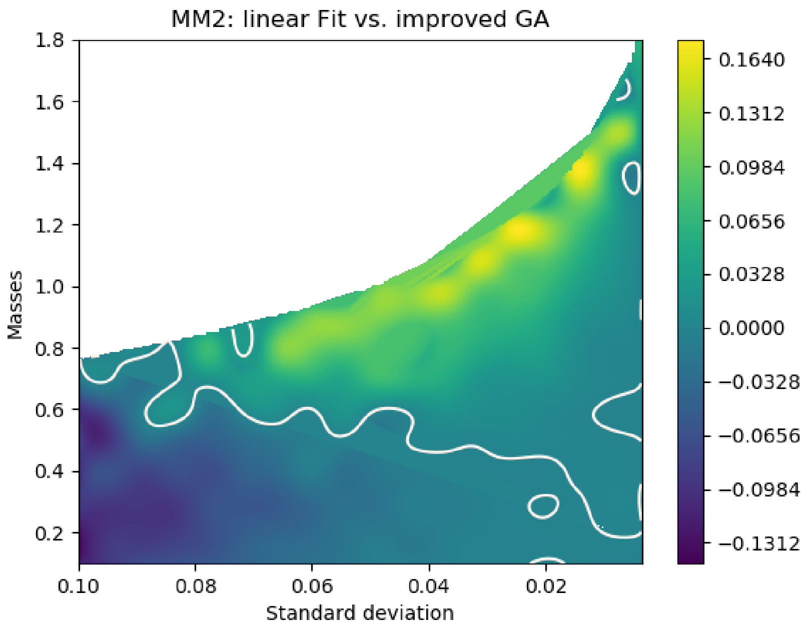


Figure 23: MM2 Comparison of the improved GA versus the linear Fit method. The GA is better where the z-value is above zero. The only difference to Wagner's GA is that the fitness value is determined like (8). A white line is drawn at $z = 0$ to distinguish the areas where the two algorithms are better. Spline interpolation like in figure 5 is used.

In figure 22 one can see that the GA is better regarding MM1 in almost all domains, except for very small masses and standard deviations. For MM2 each algorithm is better in a certain domain. These domains are of roughly equal size.

It is important to check whether the error estimate of each algorithm is too optimistic or pessimistic. Every algorithm gives a result and an estimate for the uncertainty of the result. The true value should lie within the interval

$$[result - uncertainty, result + uncertainty]$$

in 66% of the cases. Usually one does not know the true value and an algorithm is applied to find it, but in the case at hand the data was created arbitrarily and the true value is known. To create figure 22 and 23 random inputs were created 29600 times and the GA was applied on each of them, which took about 24 minutes on a common PC. This was done to diminish statistical fluctuation. For this number of runs, one gets:

true solution within $result_{improvedGA} \pm uncertainty_{improvedGA}$: **98.1%**
true solution within $result_{Wagner'sGA} \pm uncertainty_{Wagner'sGA}$: **84.5%**
true solution within $result_{linearFit} \pm uncertainty_{linearFit}$: **87.7%**

The uncertainty of the linear Fit method was calculated the same way as was done for the GA, see section 2.2.2. All algorithms overestimate their uncertainty. The improved GA overestimates it the most.

5 Summary and outlook

Wagner's work was successfully reproduced and his algorithm has been improved.

One could extent this work by applying the GA on a superposition of i exponential functions, like in equation (2). This can be achieved by changing the definition of an individual being just two numbers (a, b) to

$$(a_1, b_1, a_2, b_2, ..., a_i, b_i)$$

In addition the vast possibilities on choosing the fitness function and other parameters of the GA certainly leave some room for improvement, yet to be explored. One could even imagine combining different algorithms, that are best suited for certain masses and uncertainties of inputs. For this purpose a first estimate of the mass could be made and the algorithms could be applied accordingly based on this estimate.

References

[1] Raphael Wagner. Using machine learning for bsm particle identification, 2018.

[2] Wolfgang Banzhaf, Peter Nordin, Robert E Keller, and Frank D Francone. *Genetic programming: an introduction*, volume 1. Morgan Kaufmann San Francisco, 1998.

A Program code

The GA is written in c++. The analysis of the results is written in Python, because creating plots is easier in this language. The program mainGA.cpp has two modes, which can be toggled by changing the variable JUST_ONE_RUN_MODE to either true or false.

- **just one run mode**: runs the GA just once. The results can be viewed in 'Plots for just one run mode.py'
- **many runs mode**: runs the GA many times in order to compare the results for many different masses and standard deviations. The results can be viewed in 'Plots for comparison of performance.py'. This mode is used to create the 3D plots.

When one wants to start the program on a PC for the first time, empty files named input_data.txt, massAndSd_noise.txt, output_data_justOneRun.txt and output_data_manyRuns.txt will need to be created. The paths for these files will need to be set correctly in the C++ programs.

A.1 mainGA.cpp

```
#include <iostream>
#include <random>
#include <chrono>    // time for seed; and for measuring execution time
#include <algorithm> // std::max_element
#include <fstream>   // read file
//#include <vector>       // already included in generateInput.h
#include <cmath>     // for exp() and errror function (erf)
#include <numeric>   // std::partial_sum; not used because it doesn't work in my case
#include "generateInput.h"
#include <windows.h> // for Beep()

using std::cout;
using std::endl;

# define MIN_A 0.9 // minimum a, that an individual can have upon initialization
# define MAX_A 1.1
# define JUST_ONE_RUN_MODE false // Just_one_run_mode runs the GA only once

// construct a random generator engine from a time-based seed
unsigned seed = std::chrono::system_clock::now().time_since_epoch().count();
std::default_random_engine generator(seed);

struct Individual {
    double a;
    double b;
    double fitness;

    Individual(double p_a, double p_b)
            : a(p_a), b(p_b) {
    }
};

class Population {
    int size;                   // size of population
    Individual fittest = Individual(0, 0);
    std::vector<Individual> pop;  // vector of individuals of size "size"

    // function used for integration in eval_fitness()
    double f(double x, double sigma) {
        return x - sqrt(M_PI) * sigma * erf(x / (sqrt(2) * sigma)) / sqrt(2);
    }

public:
    explicit Population(int size_in);

    void eval_fitness(const std::vector<double> &input, double std_noise) {
        // inputs are the data points we want to fit. std_noise is the standard deviation of
            the inputs.
        // I integrated the function for the weights analytically

        double f0 = f(0, std_noise);     // define f0 for speed
        fittest.fitness = 0;
        double indiv;
        double fit_temp;
        // loop over all individuals
        for (auto &individual : pop) {
            individual.fitness = 0;                          // reset fitness for individual
            // loop over all points (about 3-18)
            for (int x = 0; x < input.size(); x++) {
                indiv = individual.a * exp(-individual.b * (x + 1));
                fit_temp = fabs(indiv - input[x]);
                fit_temp = f(fit_temp, std_noise) - f0;
                //fit_temp = fit_temp * std_noise / indiv;         // weight for relative
                    uncertainty; makes GA worse
                individual.fitness += 1 / fit_temp;                // Wagner's way
                //individual.fitness += fit_temp;                  // my improved way
            }
            //individual.fitness = 1 / individual.fitness;         // my improved way
            //individual.fitness = pow(individual.fitness, 0.25); // This does not change the
                results
            if (individual.fitness > fittest.fitness) {
                fittest = individual;
            }
        }
    }


    // create new population based on fitness of individuals
    void next_pop() {
        double cumFitness[size];
        cumFitness[0] = pop[0].fitness;
        for (int i = 1; i < size; ++i) {
            cumFitness[i] = cumFitness[i - 1] + pop[i].fitness;
        }
        double sum_of_fitness = cumFitness[size - 1]; // cumFitness[size - 1] is last element
            of cumFitness

        // create temporary new pop
        std::vector<Individual> pop_new;
        pop_new.reserve(size);

        std::uniform_real_distribution<double> distribution(0.0, sum_of_fitness);

        for (int i = 0; i < size; ++i) {
            // create random r
            double r = distribution(generator);
            // search for individual
            for (int j = 0; j < size; ++j) {
                if (r < cumFitness[j]) {
                    // individual j is chosen for reproduction
                    pop_new.emplace_back(pop[j]);
                    break; // break out of search for individual
                }
            }
        }
        pop = pop_new;
    }


    void mutate(float mutation_rate) {
        std::uniform_real_distribution<double> distribution1(0.0, 1.0);
        std::normal_distribution<double> distribution_a(0.0, 1 * 0.001); // 'mutation width'
        std::normal_distribution<double> distribution_b(0.0, 1 * 0.005); // 'mutation width'
        for (int i = 0; i < size; ++i) {
            double r = distribution1(generator);
            if (mutation_rate > r) {
                int r1 = rand() % 2;
                if (r1 == 0) {
                    pop[i].a += distribution_a(generator);
                } else {
                    pop[i].b += distribution_b(generator);

                    // This part was added to avoid negative masses.
                    if (pop[i].b < 0) {
                        pop[i].b = fabs(distribution_b(generator));
                    }
                }
            }
        }
    }


    void crossover(float crossover_rate) {
        std::uniform_real_distribution<double> distribution2(0.0, 1.0);
        std::vector<int> chosenI; // list of chosen Individuals
        double r1;
        int partner;
        double temp;
        double r2;
        for (int i = 0; i < size; ++i) {
            r1 = distribution2(generator);
            if (crossover_rate > r1) {
                chosenI.push_back(i);
            }
        }
        if (chosenI.size() % 2 == 1) {
            std::uniform_int_distribution<int> distribution3(0, chosenI.size() - 1);
            chosenI.erase(chosenI.begin() + distribution3(generator));
        }
        for (int i = 0; i < chosenI.size() / 2; ++i) {
            partner = chosenI[i + chosenI.size() / 2]; // chosen partner is about half of the
                pop away

            r2 = rand() % 2;
            if (r2 == 0) {
                temp = pop[i].a;
```

```
                    pop[i].a = pop[partner].a;
                    pop[partner].a = temp;
                } else {
                    temp = pop[i].b;
                    pop[i].b = pop[partner].b;
                    pop[partner].b = temp;
                }
            }
        }

        double get_mass_fittest() {
            return fittest.b;
        }

        double get_a_fittest() {
            return fittest.a;
        }

        double get_fitness_fittest() {
            return fittest.fitness;
        }

        void print_fittest() {
            cout << "\nFittest individual. Fitness: " << fittest.fitness << " a: " << fittest.a
                 << " mass: "
                 << fittest.b
                 << endl;
        }

        void print_pop(int gen) {
            cout << "\n Generation " << gen << endl;
            for (int i = 0; i < size; ++i) {
                cout << pop[i].fitness << " ";
            }
            cout << endl;
        }

        void save_gen_to_file() {
            std::ofstream outFile;
            outFile.open((R"(C:\Users\... YOUR PATH \output_data_justOneRun.txt)"),
                         std::ios_base::app);
            if (outFile.is_open()) {
                outFile << fittest.fitness << ", " << fittest.a << ", " << fittest.b << ", " <<
                      size;
                for (int i = 0; i < size; ++i) {
                    outFile << ", " << pop[i].b;
                }
            } else cout << "Unable to open Output file";
            outFile << "\n";
            outFile.close();
        }
};

// Population Constructor
Population::Population(int p_size) {
        size = p_size;
        pop.reserve(size);
        // e-function has the form e = a*exp(-b*x)
```

```
    std::uniform_real_distribution<double> distribution_a(MIN_A, MAX_A);
    std::uniform_real_distribution<double> distribution_b(0.01, 1.9); // Wagner: 0.07 - 1.9.
         Does not affect results
    for (int i = 0; i < size; i++) {
        pop.emplace_back(distribution_a(generator), distribution_b(generator));
    }
}

int main() {
    int max_generations = 15; // Wagner: 15
    int size_population = 450; // Wagner: 450
    float mutation_rate = 1;
    //float crossover_rate = 0;
    int inputs_generated_for_each_pair = 160; // number of times, inputs for the same mass
         -standard deviation-pair should be created
    float min_mass = 0.1;
    float max_mass = 1.8;
    float steps_mass = 0.1;
    int iter_mass_and_std = 18;

    if (round((max_mass - min_mass) / steps_mass) - ((max_mass - min_mass) /
         steps_mass) > 0.00001)
        cout << "Error: unallowed steps" << endl;

    float min_std = 0.0035; // minimum value of standard deviation of noise
    float max_std = 0.1;
    double steps_std = (max_std - min_std) / (iter_mass_and_std - 1);

    /////// run the routine for JUST_ONE_RUN_MODE;
#if JUST_ONE_RUN_MODE
    double mass = 0.5;
    double std = 0.08; // standard deviation of noise

    // clear output files
    std::ofstream ofs;
    ofs.open(R"(C:\Users\... YOUR PATH \output_data_justonce.txt)",
             std::ofstream::out | std::ofstream::trunc);
    ofs.close();

    //generate Input
    std::vector<double> input = generateInput(mass, std, seed, true);

    // initialize population
    Population pop(size_population);

    for (int j = 0; j < max_generations - 1; ++j) {
        pop.eval_fitness(input, std);
        pop.save_gen_to_file();
        pop.next_pop();
        pop.mutate(mutation_rate);
        //pop.crossover(crossover_rate);
    }
    pop.eval_fitness(input, std);
    pop.next_pop();
    pop.eval_fitness(input, std);

    // find sigma by running GA with inputs moved to upper error bar and lower error bar
         respectively
    std::vector<double> inputU; // inputs on upper error bar
    std::vector<double> inputL; // inputs on lower error bar
    inputU.reserve(input.size());
    inputL.reserve(input.size());
    Population popU(size_population);
    Population popL(size_population);
    for (auto i : input) {
        inputU.emplace_back(i + std);
        inputL.emplace_back(i - std);
    }
    for (int j = 0; j < max_generations; ++j) {
        popU.eval_fitness(inputU, std);
        popL.eval_fitness(inputL, std);
        popU.next_pop();
        popL.next_pop();
        popU.mutate(mutation_rate);
        popL.mutate(mutation_rate);
        //popU.crossover(crossover_rate);
        //popL.crossover(crossover_rate);
    }
    popU.eval_fitness(inputU, std);
    popL.eval_fitness(inputL, std);

    double sigma = popL.get_mass_fittest() - popU.get_mass_fittest();
    cout << "sigma: " << sigma << endl;
    return 0; // end main()
#endif

    ///////// Main programm
    auto start_run = std::chrono::high_resolution_clock::now();

    // create array of true masses and print them
    std::vector<double> all_masses; // list of all masses that are going to be calculated
    cout << "Masses: ";
    for (int l = 0; l < iter_mass_and_std; ++l) {
        all_masses.push_back(l * steps_mass + min_mass);
        cout << l * steps_mass + min_mass << " ";
    }
    cout << endl;

    // create array of standard deviations of noise and print them
    std::vector<double> all_std; // list of all standard deviations that are going to be
        calculated
    cout << "Standard deviations: ";
    for (int l = 0; l < iter_mass_and_std; ++l) {
        all_std.push_back(l * steps_std + min_std);
        cout << l * steps_std + min_std << " ";
    }
    cout << endl << endl;

    int total_GA_runs = iter_mass_and_std * iter_mass_and_std *
        inputs_generated_for_each_pair;
    cout << "Running the GA about " << round(total_GA_runs * 0.57101) << " times,
        estimated time "
         << total_GA_runs * 3 * 0.0001640777 << " min... Percent done:" << endl;

    // clear output files
    std::ofstream ofs2;
```

```
ofs2.open(R"(C:\Users\... YOUR PATH \output_data_manyRuns.txt)",
        std::ofstream::out | std::ofstream::trunc);
ofs2.close();

std::ofstream outFile;
outFile.open((R"(C:\Users\... YOUR PATH \output_data_manyRuns.txt)"), std::ios_base
    ::app);
if (!outFile.is_open()) cout << "Unable to open Output file";

int smallerThanZero = 0;
long int counting = 0; // this counts the number of inputs and can be used to create fixed
    input data

// loop over masses
for (int i_mass = 0; i_mass < iter_mass_and_std; ++i_mass) {
   double mass_real = all_masses[i_mass];
   cout << 100 * i_mass / iter_mass_and_std << "% ";

   // loop over standard deviations
   for (int i_std = 0; i_std < iter_mass_and_std; ++i_std) {
      double std = all_std[i_std];

      // for each mass-standard deviation pair, inputs get generated a couple of times
      for (int i_input_generated = 0; i_input_generated <
          inputs_generated_for_each_pair; ++i_input_generated) {
         // skip if points are too low
         if (exp(-3 * mass_real) < std) {
            outFile << mass_real << ", " << std;
            for (int i = 0; i < 19; ++i) {
               outFile << ", 0";
            }
            outFile << ", " << inputs_generated_for_each_pair << "\n";
            continue;
         }

         //generate Input
         counting++;
         std::vector<double> input = generateInput(mass_real, std, counting);
         unsigned int number_inputs = input.size();

         Population pop(size_population);
         for (int j = 0; j < max_generations; ++j) {
            pop.eval_fitness(input, std);
            pop.next_pop();
            pop.mutate(mutation_rate);
            //pop.crossover(crossover_rate);
         }
         pop.eval_fitness(input, std);

         // find sigma by running GA with inputs moved to upper errorbar and lower
            errorbar respectively
         std::vector<double> inputU;
         std::vector<double> inputL; // inputs on upper and lower errorbar
         inputU.reserve(number_inputs);
         inputL.reserve(number_inputs);
         Population popU(size_population);
         Population popL(size_population);
         for (auto i : input) {
                inputU.emplace_back(i + std);
                inputL.emplace_back(i − std);
            }
            for (int j = 0; j < max_generations; ++j) {
                popU.eval_fitness(inputU, std);
                popL.eval_fitness(inputL, std);
                popU.next_pop();
                popL.next_pop();
                popU.mutate(mutation_rate);
                popL.mutate(mutation_rate);
                //popU.crossover(crossover_rate);
                //popL.crossover(crossover_rate);
            }
            popU.eval_fitness(inputU, std);
            popL.eval_fitness(inputL, std);

            double sigma = fabs(popL.get_mass_fittest() − popU.get_mass_fittest());

            if (sigma < 0)smallerThanZero++;

            // append to output file
            outFile << mass_real << ", " << std << ", " << number_inputs;
            for (int i = 0; i < number_inputs; ++i) {
                outFile << ", " << input[i];
            }
            for (int i = 0; i < 16 − number_inputs; ++i) {
                outFile << ", 0";
            }
            outFile << ", " << pop.get_mass_fittest() << ", " << sigma << ", " <<
                 inputs_generated_for_each_pair
                     << "\n";
        } // end of loop over input−creation
    } // end of loop over standard deviations
} // end of loop over masses

cout << endl << "sigma is smaller than Zero: " << smallerThanZero << endl;
auto finish_run = std::chrono::high_resolution_clock::now();
std::chrono::duration<double> elapsed_run = finish_run − start_run;
cout << "\nElapsed time run: " << elapsed_run.count() / 60 << " min\n";
cout << "GA ran " << counting << " times.";
Beep(523, 500);
return 0;
}
```

A.2 generateInput.h

```
//
// Created by Lukas on 04.12.2017.
//

#ifndef BACHELORARBEIT_CLION_GENERATEINPUT_H
#define BACHELORARBEIT_CLION_GENERATEINPUT_H

#include <vector>

std::vector<double> generateInput(double mass, double sd_noise, long int counting, bool
     write_input_to_file = false);
```

```
#endif //BACHELORARBEIT_CLION_GENERATEINPUT_H
```

A.3 generateInput.cpp

```
//
// Created by Lukas on 04.12.2017.
//

#ifndef BACHELORARBEIT_CLION_GENERATEINPUT_H
#define BACHELORARBEIT_CLION_GENERATEINPUT_H

#include <vector>

std::vector<double> generateInput(double mass, double sd_noise, long int counting, bool
    write_input_to_file = false);

#endif //BACHELORARBEIT_CLION_GENERATEINPUT_H
```

A.4 Plots for just one run mode.py

```
# Lukas Reicht Output data Processing for Bachelor thesis

import matplotlib.pyplot as plt
import matplotlib.animation as animation
import numpy as np
from scipy.optimize import curve_fit
import scipy.integrate as integrate

with open("output_data_justOneRun.txt") as outFile:
    dataOut = np.loadtxt(outFile, delimiter=", ")

# transform output data
 fitnessFittest  = dataOut[:, 0]
aFittest  = dataOut[:, 1]
massFittest = dataOut[:, 2]
populationSize = dataOut[:,3]
populationSize = populationSize[0].astype(int)
masses = dataOut[:, 4:(populationSize+4)]
different_values  = np.ma.size(dataOut,1) # different values per Generation
gen = np.ma.size(dataOut) / different_values # number of generations
gen = int(gen)

# read in input points
with open("input_data.txt") as inFile:
    dataIn = np.loadtxt(inFile)

# read in true mass
with open("massAndSd_noise.txt") as massFile:
    mass_true = np.loadtxt(massFile)
sd_noise = mass_true[1]
mass_true = mass_true[0]

# This compares the result to scipy.optimize
compare_to_scipy_optimize = True
if compare_to_scipy_optimize:
    x_d = np.arange(1, np.size(dataIn) + 1)          # discrete x array (about 5 values)
    efun = lambda t,a,b: a*np.exp(−b*t)
    sigma = np.full(np.size(dataIn), 0.05)

    # bounds: a between 0.7 and 1.3. mass(b) between 0.1 and 1.8
    # bounds improve curve_fit by a ton!
    # , bounds=([1.0, 0.1], [1.9, 1.8])
    sp_solution, pcov = curve_fit(efun, x_d, dataIn, sigma=sigma, bounds=([0.7, 0.1], [1.3,
        1.8]) )

    # calculate errors; i am not 100% sure this is correct
    perr = np.sqrt(np.diag(pcov))

    print("True mass: ", mass_true)
    print("Scipy mass: ", sp_solution[1], "+−", perr[1])
    print("My GA solution: ", massFittest[gen − 1])
    print("sp a", sp_solution[0], "+−", perr[0])

    # printing the results of scipy
    x_c = np.linspace(0, np.size(dataIn), 300)       # "continuous" x array (100 values)
    fig, ax = plt.subplots()
    line1 = ax.plot(x_c, sp_solution[0] * np.exp(−sp_solution[1] * x_c), label="scipy",
        linestyle="dashed", color="blue")
    line2 = ax.plot(x_c, aFittest[−1] * np.exp(−massFittest[−1] * x_c), label="GA", linestyle
        ="dashed", color="green")
    line3 = ax.plot(x_c, 1 * np.exp(−mass_true * x_c), label="true data", color="orange")
    line4 = plt.errorbar(x_d, dataIn, yerr=sd_noise, color='blue', fmt='o', label="errorbars")
        # create errorbars
    plt.xlabel("x")
    plt.ylabel("f(x)")
    plt.legend()
    plt.show()


# This prints an animation of the evolution of the fittest individual
print_certain_generations = False
if print_certain_generations:
    # make an array of certain generations, that I want to print (e.g. [0, 1, 2, 26, 51, 76,
        100] for 100 generations)
    gens_to_print = [0, 1] + list(np.linspace(2, gen, 5))
    for i in range(np.size(gens_to_print)):
        gens_to_print[i] = int(round(gens_to_print[i]))

    # create figure
    fig, ax = plt.subplots()

    # create first plot (Generation 0)
    x_c = np.linspace(0, np.size(dataIn), 100)       # "continuous" x array (100 values)
    x_d = np.arange(1, np.size(dataIn) + 1)          # discrete x array (about 5 values)
    line, = ax.plot(x_c, aFittest[0] * np.exp(−massFittest[0] * x_c)) # first plot
    fig.suptitle('Generation 0')
    line1 = plt.errorbar(x_d, dataIn, yerr=sd_noise, color='blue', fmt='o') # create errorbars

    # define each frame of animation
    def animate(i):
        x_c = np.linspace(0, np.size(dataIn), 100)           # "continuous" x array (100 values
            )
        line.set_ydata(aFittest[i] * np.exp(−massFittest[i] * x_c))        # update the data
        fig . suptitle ('Generation ' + str(gens_to_print[i])) # update title
        return line

    ani = animation.FuncAnimation(fig, animate, np.arange(0, np.size(gens_to_print)),
                                  interval=300, repeat=True, repeat_delay=700)

    plt .show()

# This prints an animation that shows, how the mass (=b) distribution changes over the
    Generations
print_mass_distribution = False
if print_mass_distribution:
    bins = 150
    xlimFixedOn = 0 # Generation of which max and min element determine the xlimits

    # make an array of certain generations, that I want to print (e.g. [0, 1, 2, 26, 51, 76,
        100] for 100 generations)
    gens_to_print = list(range(gen))

    # create figure
    fig , ax = plt.subplots()

    # create first plot (Generation 0)
    ax. hist (masses [0,:], bins)
    plt .xlim ([0,1.8])
    plt .xlabel("mass")
    plt .ylabel("Number of individuals")
    #plt.xlim([min(masses[0,:]) , max(masses[0,:]) ])
    fig . suptitle ('Generation 0')

    # define each frame of animation
    def animate(i):
        ax. clear ()
        ax. hist (masses[i ,:], bins, range=[min(masses[xlimFixedOn,:]), max(masses[
            xlimFixedOn,:])])
        plt . xlabel ("mass")
        plt . ylabel ("Number of individuals")
        #plt.xlim([min(masses[xlimFixedOn,:]), max(masses[xlimFixedOn,:])])
        plt .xlim ([0,1.8])
        plt .ylim ([0,350])
        fig . suptitle ('Generation ' + str(gens_to_print[i]))
        return 0

    ani = animation.FuncAnimation(fig, animate, np.arange(0, np.size(gens_to_print)),
                                  interval=500, repeat=True, repeat_delay=1000)

    plt .show()

forExplaniningFitnessFunction = True
if forExplaniningFitnessFunction:
    x_d = np.arange(1, np.size(dataIn) + 1)          # discrete x array (about 5 values)
    efun = lambda t,a,b: a*np.exp(-b*t)
    SIGMA = 0.05
    sigma = np.full(np.size (dataIn), SIGMA)

    # printing the results of scipy
    x_c = np.linspace(0, np.size(dataIn), 300)
    fig , ax = plt.subplots()
    line1 = ax.plot(x_c, aFittest[-1] * np.exp(-massFittest[-1] * x_c), label="Exponential
        function of some individual", color="blue")
    line2 = ax.plot(x_d, aFittest[-1] * np.exp(-massFittest[-1] * x_d), 'x', color="blue",
        label="$D_i$ ... belong to the individual")
    line3 = ax.plot(x_d, dataIn, 'bx', color='red', label="$C_i$ ... data that we want to fit"
        )
    plt.xlabel("x")
    plt.ylabel("f(x)")
    plt.legend()

    # calculating stuff
    Fitness = 0.0
    for i in range(np.size(dataIn)):
        print(i + 1)
        C = dataIn[i]
        D = aFittest[-1] * np.exp(-massFittest[-1] * (i + 1))
        print('|Ci - Di|: ', round(abs(C-D), 8))
        def integrand(x):
            return 1 - np.exp(-x**2 / (2 * SIGMA**2))
        denominator = integrate.quad(integrand, 0, abs(C - D))
        print('denominator: ', round(denominator[0], 15))
        print(denominator)
        Fitness = Fitness + 1 /denominator[0]


    print(Fitness)
    plt.show()



print_single_mass_distribution = False
if print_single_mass_distribution:
    bins = 150
    gen_to_print = 15

    # create first plot (Generation 0)
    plt.hist(masses[gen_to_print,:], bins)
    #plt.xlim([min(masses[gen_to_print,:]), max(masses[gen_to_print,:])])
    plt.xlim([0.1,1.8])
    plt.xlabel("mass")
    plt.ylabel("Number of individuals")
    #plt.xlim([min(masses[0,:]), max(masses[0,:])])
    plt.title('Generation 10')

    plt.show()
```

A.5 Plots for comparison of performance.py

```
# Lukas Reicht Output data Processing for Bachelor thesis

import matplotlib.pyplot as plt
import numpy as np
from mpl_toolkits.mplot3d import Axes3D
from matplotlib.ticker import LinearLocator, FormatStrFormatter
from matplotlib import cm # colormap
import scipy.interpolate as interpolate  # interpolate
import scipy.sparse.linalg  as  splin

# what this programm does:
# compary improved GA with Wagner's GA

with open("output_data_manyRuns.txt") as outFile:
    dataOut = np.loadtxt(outFile, delimiter=", ", )

mass_real = dataOut[:, 0]
noise_std = dataOut[:, 1]  # gauss standard deviation used to create inputs
number_inputs = dataOut[:, 2].astype(int)
inputs = dataOut[:, 3:19]  # this  is  the correct  index
GAsolution = dataOut[:, 19]
GAsigma = dataOut[:, 20]
inputs_generated_for_each_pair = dataOut[:, 21] # number of times an input is created for
     each mass_real−noise_std−pair
inputs_generated_for_each_pair = inputs_generated_for_each_pair[0].astype(int)
masses_generated = 18
std_generated = masses_generated
totalNumberOfInputs = np.count_nonzero(number_inputs)

with open("output_data_Wagner_Reference.txt") as outFileR:
    dataOutR = np.loadtxt(outFileR, delimiter=", ", )

mass_realR = dataOutR[:, 0]
noise_stdR = dataOutR[:, 1] # gauss standard deviation used to create inputs
number_inputsR = dataOutR[:, 2].astype(int)
inputsR = dataOutR[:, 3:19]  # this  is  the correct  index
GAsolutionR = dataOutR[:, 19]
GAsigmaR = dataOutR[:, 20]
inputs_generated_for_each_pairR = dataOutR[:,
                                           21]  # number of times an input is created for  each
                                                mass_real−noise_std−pair
inputs_generated_for_each_pairR = inputs_generated_for_each_pairR[0].astype(int)

if  mass_real.all() != mass_realR.all(): print("Error: size  of  new run and
     RaphealReproduction do not agree.")
if  inputs. all () != inputsR.all(): print("Error: not  all  inputs are the same.")

#  initialize  Measure of Merits
dp_MM = 18 # datapoints of MM
# dp_MM = dp_MM.astype(int)
MM1_GA = np.zeros((dp_MM, dp_MM))
MM1_RGA = np.zeros((dp_MM, dp_MM))
MM1_lin = np.zeros((dp_MM, dp_MM))
MM2_GA = np.zeros((dp_MM, dp_MM))
MM2_RGA = np.zeros((dp_MM, dp_MM))
MM2_lin = np.zeros((dp_MM, dp_MM))
# for MM calculate mean for each pair
testMatrix = np.zeros((dp_MM, dp_MM))

# define e−function
efun = lambda t, a, b: a ∗ np.exp(−b ∗ t)

minMassOfLin = 10
maxMassOfLin = 0
counterLinFitNegative = 0
i  = −1
```

```
RGACount, GACount, linCount = 0, 0, 0
linSigmaNotEvaluatedCount = 0
for i_mass in range(masses_generated):
    for i_std in range(std_generated):
        for i_inputs in range(inputs_generated_for_each_pair):
            i += 1 # iterator over total number
            if number_inputs[i] == 0: # skip these
                continue

            x_d = np.arange(1, number_inputs[i] + 1) # discrete x array starting from 1 to (
                number_inputs + 1)

            # cut zeros away from input
            inputs_i = inputs[i ][0: number_inputs[i]]

            ### Evaluate least squares linfit

            # y + error = a * exp(-b*x)
            # ln(y + error) = ln(a) - b * x
            # np.log() = natural logarithm
            # error propagation:  (1/y) * deltaY
            linSigmaInput = np.full(number_inputs[i], noise_std[i])  / inputs_i
            linWeight = 1 / linSigmaInput  # according to documentation

            [mass_lin, a_lin] = np.polyfit(-x_d, np.log(inputs_i), 1, w=linWeight)
            a_lin = np.exp(a_lin)

            for k in inputs_i:
                if k < noise_std[i]:
                    print("error1", i + 1, k, noise_std[i])

            if (mass_lin < minMassOfLin) and (mass_lin > 0):
                minMassOfLin = mass_lin
            if mass_lin > maxMassOfLin: maxMassOfLin = mass_lin
            if mass_lin < 0:
                mass_lin = mass_real[i]
                counterLinFitNegative = counterLinFitNegative + 1
            # if mass_lin < 0.1: # add unfair advantage to Lin Fit
            # mass_lin = mass_real[i]

            # calculate error by making linfit through upper and lower point of errorbar and
                taking that difference
            [lowerMass, _] = np.polyfit(-x_d, np.log(inputs_i + np.full(number_inputs[i],
                noise_std[i])), 1,
                                    w=linWeight)
            [upperMass, _] = np.polyfit(-x_d, np.log(inputs_i - np.full(number_inputs[i],
                noise_std[i])), 1,
                                    w=linWeight)
            linSigma = np.mean(abs(upperMass - lowerMass))

            if i % 1000 == 0: print(i)

            # calculate MM and its deviation for each mass
            MM1_GA[i_mass, i_std] += abs(GAsolution[i] - mass_real[i]) / mass_real[i]
            MM1_RGA[i_mass, i_std] += abs(GAsolutionR[i] - mass_real[i]) / mass_real[i]
            MM1_lin[i_mass, i_std] += abs(mass_lin - mass_real[i]) / mass_real[i]

            MM2_GA[i_mass, i_std] += GAsigma[i] / mass_real[i]
            MM2_RGA[i_mass, i_std] += GAsigmaR[i] / mass_real[i]
            MM2_lin[i_mass, i_std] += linSigma / mass_real[i]

            if abs(mass_real[i] − GAsolutionR[i]) < GAsigmaR[i]:
                RGACount = RGACount + 1
            if abs(mass_real[i] − GAsolution[i]) < GAsigma[i]:
                GACount = GACount + 1
            if abs(mass_real[i] − mass_lin) < linSigma:
                linCount = linCount + 1

MM1_GA = MM1_GA / inputs_generated_for_each_pair
MM1_RGA = MM1_RGA / inputs_generated_for_each_pair
MM1_lin = MM1_lin / inputs_generated_for_each_pair
MM2_GA = MM2_GA / inputs_generated_for_each_pair
MM2_RGA = MM2_RGA / inputs_generated_for_each_pair
MM2_lin = MM2_lin / inputs_generated_for_each_pair

print("solution within +−GAsigma (should be > 67%): ", round(100 ∗ GACount /
    totalNumberOfInputs, 4), "%")
print("solution within +−RGAsigma (should be > 67%): ", round(100 ∗ RGACount /
    totalNumberOfInputs, 4), "%")
print("solution within +−linsigma (should be > 67%): ", round(100 ∗ linCount /
    totalNumberOfInputs, 4), "%")

# plot results
x = np.linspace(np.amin(noise_std), np.amax(noise_std), dp_MM) # standard deviation
y = np.linspace(np.amin(mass_real), np.amax(mass_real), dp_MM) # mass
X, Y = np.meshgrid(x, y)

# remove points from final plot for which the input is not meaningful
Remove = np.exp(−3 ∗ Y) < X
X[Remove] = None
Y[Remove] = None

def my3Dplot(Z, title):
    interpZ = interpolate.interp2d(x, y, Z, kind='cubic')

    xfine = np.linspace(np.amin(noise_std), np.amax(noise_std), 3 ∗ dp_MM) # standard
        deviation
    yfine = np.linspace(np.amin(mass_real), np.amax(mass_real), 3 ∗ dp_MM) # mass
    Zfine = interpZ(xfine, yfine)
    X, Y = np.meshgrid(xfine, yfine)

    Remove = np.exp(−3 ∗ Y) < X
    X[Remove] = None
    Y[Remove] = None

    fig = plt.figure()
    ax = fig.gca(projection='3d')
    surf = ax.plot_surface(X, Y, Zfine, cmap=cm.viridis) # good colormap: coolwarm
    fig.colorbar(surf, shrink=0.5, aspect=5) # add a color bar
    plt.title(title)
    ax.set_xlabel("Standard deviation")
    ax.set_ylabel("Masses")
    ax.invert_xaxis() # make it look like Raphael's plot


def my2Dplot(Z, title):
    interpZ = interpolate.interp2d(x, y, Z, kind='cubic')

    xfine = np.linspace(np.amin(noise_std), np.amax(noise_std), 10 * dp_MM) # standard
        deviation
    yfine = np.linspace(np.amin(mass_real), np.amax(mass_real), 10 * dp_MM) # mass
    Zfine = interpZ(xfine, yfine)
    X, Y = np.meshgrid(xfine, yfine)

    Remove = np.exp(-3 * Y) < X
    X[Remove] = None
    Y[Remove] = None

    fig = plt.figure()
    cf = plt.contourf(X, Y, Zfine, 500)
    plt.colorbar(cf)
    CS = plt.contour(X, Y, Zfine, levels=[0], cmap='Reds') #white line at zero
    plt.xlim([max(x), min(x)]) # invert axis
    plt.ylim([min(y), max(y)])
    plt.xlabel("Standard deviation")
    plt.ylabel("Masses")
    plt.title(title)


Z1 = MM1_RGA - MM1_GA # if Z is larger than 0, my GA is better than Raphael's
my3Dplot(Z1, "MM1: GA without vs. Ga with factor")
my2Dplot(Z1, "MM1: GA without vs. Ga with factor")

Z2 = MM2_RGA - MM2_GA
my3Dplot(Z2, "MM2: GA without vs. Ga with factor")
my2Dplot(Z2, "MM2: GA without vs. Ga with factor")

Z3 = MM1_lin - MM1_GA
#my3Dplot(Z3, "MM1: linear Fit vs. GA")
#my2Dplot(Z3, "MM1: linear Fit vs. GA")

Z4 = MM2_lin - MM2_GA
#my3Dplot(Z4, "MM2: linear Fit vs. GA")
#my2Dplot(Z4, "MM2: linear Fit vs. GA")


#my3Dplot(MM2_GA, "MM2 Wagner's Ga with bad error estimate")
#my2Dplot(MM2_GA, "MM2 Wagner's Ga with bad error estimate")
#my3Dplot(MM2_RGA, "MM2 Wagner's Ga with good error estimate")
#my2Dplot(MM2_RGA, "MM2 Wagner's Ga with good error estimate")
#my3Dplot(MM2_lin, "MM2 linear least squares Fit")
#my2Dplot(MM2_lin, "MM2 linear least squares Fit")

GAsigma = GAsigma[np.nonzero(number_inputs)]
GAsigmaNonzero = GAsigma[np.nonzero(GAsigma)]
counter = 0
for i in GAsigmaNonzero:
    if i < 0: counter = counter + 1
test1 = number_inputs[np.nonzero(number_inputs)]
if counter > 0: print("# GAsigma ist negativ:", counter)
if (test1.size - GAsigmaNonzero.size) > 0: print("# GA sigma ist 0:", test1.size -
    GAsigmaNonzero.size)

print("inputs created: ", GAsigmaNonzero.size, GAsigma.size)
print("Max mass of Lin Fit:", maxMassOfLin)
print("Min mass of Lin Fit:", minMassOfLin)
print("Times mass of Lin Fit is negative:", counterLinFitNegative)

print("Mean of MM1 RGA vs. me: ", np.round(np.mean(Z1), 5))
print("Mean of MM2 RGA vs. me: ", np.round(np.mean(Z2), 5))

plt.show()
```

A.6 CMakeList.txt

```
cmake_minimum_required(VERSION 3.10)
project(pretty_Bachelor_thesis)

set(CMAKE_CXX_STANDARD 11)

set(SOURCE_FILES mainGA.cpp generateInput.cpp)

add_executable(pretty_Bachelor_thesis ${SOURCE_FILES})
```