Bachelor Thesis (BSc)

Using machine learning for BSM particle identification

by:

Raphael Wagner

Supervisor:

Axel Maas

Abstract

Identifying particles in Beyond the Standard Model (BSM) simulations is a problem due to the high numerical inaccuracy of the data points. Using normal fit methods to compute the mass of the simulated particles do not have satisfying results.

Applying evolutionary algorithms on this problem shows that there can be found better solutions for identifying the particles than normal fit methods. This depends on the value for the mass and the numerical inaccuracy of the data points.

This work shows the concepts I applied on this problem and how the algorithm performs compared to a normal fit method, which is a least-square-fit. Both concepts are compared over a wide range of combinations from simulated masses and numerical inaccuracy of the data points.

Contents:

1. Introduction

The resulting data points from BSM simulations have a big inaccuracy due to numerical inaccuracy and statistical averaging during the calculation.

The information of the data points are needed to identify the simulated particles (more about this problem is written in chapter 2). The big inaccuracy leads to a high uncertainty in the fitted masses from the particles.

This is a nonlinear fit problem. Therefore there is no proof that the normal fit method finds always the best possible solution. This is the motivation to try something different which works with other concepts to find a solution for the fit problem.

The idea is to apply an evolutionary algorithm on the problem. You can read about the basic principles of such algorithms in chapter 3.

The main part of this concept in this case is to create a fitness function which judges the stochastic created solutions for the fit. This problem is detailed in chapter 6.

2. The physical problem [2]

In quantum field theory there are operators which create a particle at position x and destruct them on position y in a state e.g. the state 0. With bra-ket formalism the equation has the form of:

$$\langle 0|O_\pi^+(y)\ O_\pi(x)|0\rangle$$

The time transformation is a unitary transformation and has the form of:

$$\left\langle 0\middle|O_\pi^+(y,0)e^{iHt}e^{-iHt}O_\pi(x,0)\middle|0\right\rangle$$

Applying a Fourier-Transformation with a linear momentum of 0 leads to a problem similar to quantum mechanics, because then the Operator is no longer depending of the space coordinates.

To work on this problem it is easier to substitute the variable $t \rightarrow i\tau$. Then the time transformation has the form of:

$$e^{-\tau m} \tag{1}$$

This exponential function is better for numerical problems than the oscillation function. In a real simulation the operator which creates a π-particel is not known. It is possible to use the Hamilton Operator as a base and to write the original operator as a linear combination of Eigenstates from the Hamilton Operator. Then the Operator can be written as a linear combination from multiple exponential functions.

$$a_1e^{-m_1t} + a_2e^{-m_2t} + a_3e^{-mt} + \cdots \tag{2}$$

Solving this system with numerical methods for integer numbers of τ gives data points as a result which have a numerical error.

Then the next step is to fit the values for m_i and a_i with a normal fit-function. Due to the big numerical error of the data points the fit values also have a big inaccuracy. This is the reason why I applied an evolutionary algorithm on this problem.

For the first application of an evolutionary algorithm I simplified the problem of having a sum of different exponential functions like Eq. (2). Instead I used only data points from a single exponential function like Eq. (1).

3. How an evolutionary algorithm works [1]

3.1 The main routine

An evolutionary algorithm consists of four main parts.

These are:

- Fitness function
- Parenting process
- Mutation
- Termination condition.

The main routine is shown in figure 1.

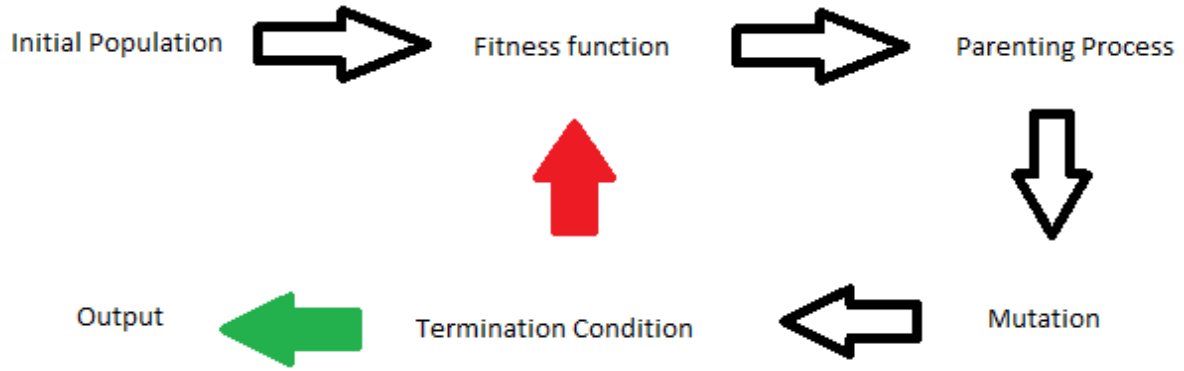

Figure 1: general working steps of an evolutionary algorithm

An evolutionary algorithm starts with an initial setup for the first generation of the population. The algorithm then changes the population by a mix of a deterministic and stochastic processes until the quality of the population satisfies a termination condition.

The stochastic process is the main routine of an evolutionary algorithm. The fitness function is a deterministic process while the parenting and the mutation are stochastic processes.

3.2 Fitness function

The fitness functions judges every element of the population how good it satisfies the problem and gives them a quantitative value. This is the so called fitness value. Creating a good fitness function can be very difficult and depends on the problem that should be solved. The fitness function creates a fitness landscape (Figure 2). The input parameters are the population variables. If the algorithm works how it should, the population elements should then get localized at the global maximum of the fitness landscape after a few generations.

3.3 Parenting Process

After the fitness function the process of drawing a new generation starts. The elements of the population with a higher fitness value should have a higher chance to get into the parenting

process. The parenting process can be asexual e.g. the element of the old generation gets into the new generation unchanged or it can be sexual e.g. a mixture of two elements from the old generation is then a part of the new generation. There are many different possibilities of designing a parenting process. The most important thing about all parenting processes is that the drawing of a new generation is always a stochastic process. Every element of the old generation should have a chance of nonzero to become part of the new generation. Even when an element has a very bad fitness value it is necessary that there is a low chance for that element to get into the new generation.

3.4 Mutation

The next step of the main routine is the mutation. The mutation changes all elements of the new generation by a stochastic process. The mutation should have a significant influence on the population. It should not be too high so the elements with a good fitness value gets changed completely but on the other side a population with too less different elements will decrease in quality after a few generations. The mutation can have different forms. For example if the elements of the population are number values the mutation can be done by adding random numbers but also other methods are possible. That would be converting the values into the binary number system and then changing the digits by a given chance. This can be a good method in some cases but also worse in others, therefore the first digits have a greater influence on the value of the number than the last digits.

3.5 Termination condition

At the end of the main routine is the termination condition. This condition examines if the population or an element of the population satisfies the problem that should be solved by the algorithm. If this is the case the termination condition stops the main routine and the algorithm should generate an output. Termination conditions can be satisfied if the population or an element of the population has a high enough fitness value or the population has no significant changes over a few generations and just oscillate around a global or a local maximum in the fitness landscape.

On figure 2 you can see a fitness landscape with different populations in different colors and symbols on it. The first generation can look like the blue triangles. They are wide spread over the complete range and have good and less good fitness values.

After a few generations the population can be like the orange stars. They are located around the local and global maxima of the landscape. This population should then get more and more concentrated to the global maximum but there is no guarantee that this will happen because the algorithm is a stochastic process. It is possible that the population keeps stuck in the local maxima.

Populations can also develop like the red points. This population is located near the global maximum but has only a very low variety. After a few more generations it can turn into something like the green squares. These points are located mostly at the global maximum and that is exactly what we want. But there is a chance that this might not happen. Due to the

stochastic process the population might not climb on top of the global maximum. It can fall down and climb up at the local maximum beside.

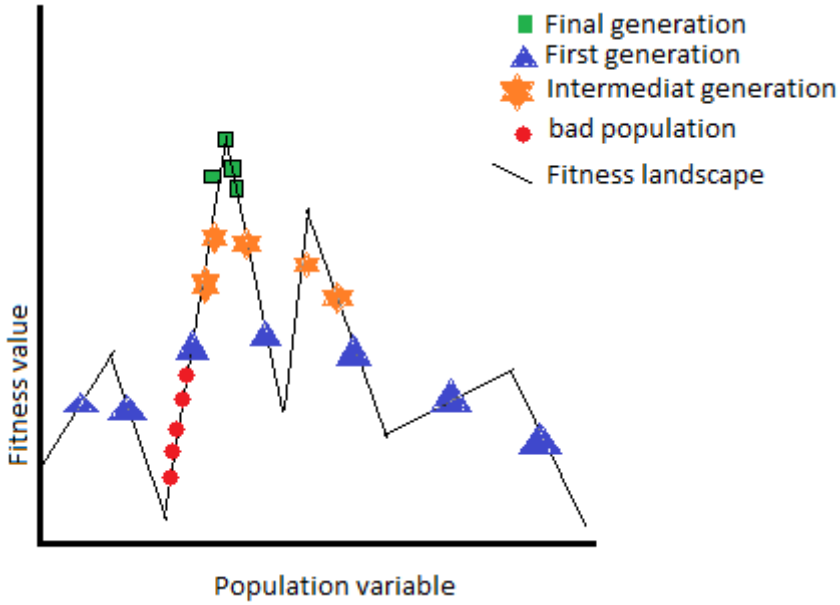

Figure 2: evolution of a population

It is very important that the stochastic processes work how they should. Also an element with a very low fitness value should have a chance to become part of the new generation. Even when the chance is very low it is important that it is not zero. Also the mutation process should have a significant effect on the population. A population with too less variety will decrease in quality after a few generations.

4. Testing my Code

To the test quality of the results from my code, I created sets of the data from given values of the parameters “a” and “m” from Eq. (2). Then the data points were manipulated to simulate the numerical error from real data points.

For every combination of the input parameters and the high of the numerical error were 10 sets of data points created. The results for the value of “m” from Eq. (1) of every set of data points were then compared to the real values which are known to judge the quality of the algorithm.

All sets of data points were tested with the normal fit method from Mathematica too, to compare the results of my code with them. After testing all data sets and plotting the results it is then possible to see in chapter 7 for which values for “m” and which high of the simulated numerical error one of the two different methods work better than the other.

5. Creating the test data

For the test of the algorithm I created data points from an exponential function like Eq. (1) with integer numbers for τ. The value range for τ starts at 1 and has a maximum at 16.

Then pseudo random numbers were added from a normal distribution with mean zero and different values for the standard deviation starting at 0.0035 up to 0.1. This should simulate the numerical inaccuracy which real data points would have.

The value "m" for the test data starts with 0.1, increases in 0.1 steps and ends at a maximum of 1.8. For the test only data points were used which have a greater value than their uncertainty, therefor it is not possible for the data points to have a negative value.

You can see an example for the data in Figure 3 below. These data points are from an "m" value of 0.6 and the standard deviation of the added random numbers is 0.05.

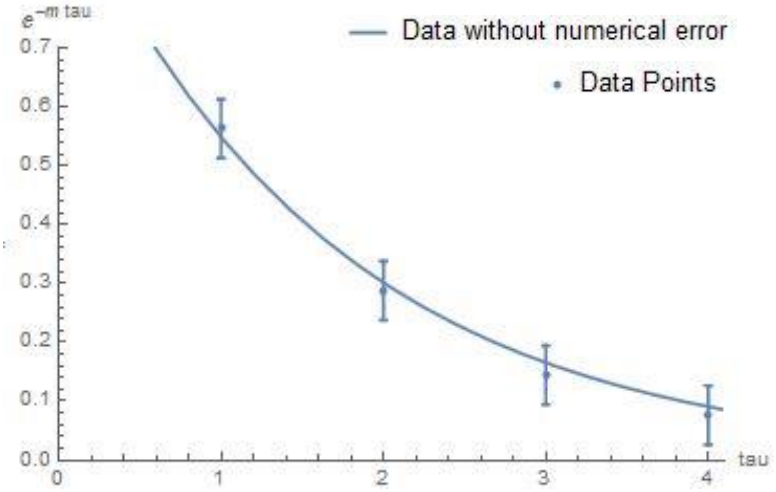

Figure 3: example for a set of data points

6. Working steps of my code

The first version of my code was written in Mathematica but the calculation time was only for very small population sizes short enough to work on the problem. This is why I switched to c++ after the basic concepts worked in Mathematica for small populations. Testing my algorithm on the data in c++ allowed me to create much higher populations. I compared the results of the code for population sizes from 400 up to 3000 for a few different sets of test data. Except a linear increase of the calculation time there was no difference in the quality of the final values for "m" of Eq. (1). This is why I chose to work with a constant population size of 450 to keep the calculation time shorter than one minute for all kind of sets of data points.

6.1 Fitness function

The fitness function calculates the distance between the read in data and the point of the population element for all τ values. Each value gets modified by integrating the function:

$$1 - e^{-(x^2/2\sigma^2)} \tag{3}$$

From 0 to the distance value, where x is the integration value and σ is the numerical error of the data points. This integration value is then the new fitness value. Integrating the function (3) includes the information of the error bars from the data points that it is not exactly known where that point is. It is only known that it should be in the range of the error bars around the point. For a smaller value for σ the fitness value is bigger by the same distance between the populations point and the read in data point.

After the integration the fitness value gets inverted and then the algorithm sums up all inverted values like:

$$f_{Final} = \sum_i^{n_{data}} \frac{1}{f_i} \tag{4}$$

So as you can see the main influence of the final fitness value comes from the data points which are close to the function of the population's element. It would be very different if the fitness values would be summed up like:

$$f_{Final} = \frac{1}{\sum_i^{n_{data}} f_i} \tag{5}$$

In that case the main influence for the final value would come from the data points which are far away from the population´s element.

A normal Least-Square-Fit tries to minimize the big differences between the fit and the data points. With a fitness function similar to Eq. (5) the algorithm would probably create a population similar to the LSQ-Fit.

On the other side a fitness function like Eq. (4) should do the exact opposite and create a population which concentrates on the data points that are close to the exponential function from the elements. This is why I choose to sum up the fitness values for each element like Eq. (4).

Before this final version of the fitness function I worked with other simpler things. One of these was to sum up the inverse of the distance without any manipulations:

$$f_{Final} = \sum_{i}^{n_{data}} \frac{1}{x_i}$$

And to sum up the squared inverse values of the distances like:

$$f_{Final} = \sum_{i}^{n_{data}} \frac{1}{x_i^2}$$

What I wanted was to create is a fitness function which clearly separates the good elements of a population from the bad. This separation should not be too extreme because then there will be only one or two different elements after the first generation left. Cases like this will most likely not lead to a good result at the end.

6.2 Parenting process

The parenting process is responsible for creating a new generation. In this step all final fitness values for each element are in an array and get summed up cumulative. The array has a zero appended as the first element. Then the complete array gets divided with the value of the last element. After that the cumulative fitness array starts with 0 and ends with 1.

Then the algorithm creates a new array with random values from a uniform distribution in the range from 0 to 1. The random numbers from this array are the stochastic part of the parenting process. The algorithm creates a new generation with the elements from the old generation which are chosen from the random uniform numbers. E. g. one part of the array has the value 0.65 then the element of the old generation with the cumulative fitness value which is smaller than 0.65 but bigger than all others is becoming part of the new generation. So this is only an asexual parenting process. All elements of the new generation are from the old generation and no new elements are created.

Including a sexual parenting process and creating some new elements for each generation which have not been in the old generation could lead to a better performance of the algorithm. Also a higher size of the population could lead to better solutions of the code.

The parenting process can be improved by adding crossing. For this method it would be necessary to draw 2 elements of the population and then switching the values for “a” and “m” of Eq. (2) by a given chance p. This would be a Bernoulli experiment because the chance that the crossing happens is unbiased for every new element of the next generation. Therefore when N is the population size, p*N elements of the next generation are generated by crossing and N*(1-p) are created with the asexual process.

The working step which normalizes the cumulative array from 0 to 1 is basically not needed. It would work the same way if the random numbers were created from a uniform distribution from 0 to a maximum of the last number from the cumulative fitness array. I added this because with this normalization the distance between one element and the next is equal to the chance to get into the next generation. With this information it is possible to see how high the chance for a bad element is to get into the next generation compared to a good one.

6.3 Mutation

The mutation of the algorithm is very simple. All elements of the generation are modified by adding random number from a gauss distribution. The standard deviation for the added random numbers for “a” and “m” of Eq. (2) are different while the mean of the random numbers is always zero. The standard deviation for the values of “a” is 0.001 and the standard deviation for the values of “m” is 0.005. The mutation is smaller for the values of “a”, because of the smaller range for the initialized values of the first generation. A strong mutation would not improve the fitness value of the elements. Already small changes for both values have a big influence of the fitness value for the next generation.

Beside this mutation with constant values for the standard deviation of the added random numbers, I worked with a different method before. The standard deviation depended on the standard deviation from the current population. This means with a wider spread generation the mutation has a bigger influence. When the populations is concentrated around a maximum of the fitness landscape the mutation becomes weaker.

For this problem both methods had no significant differences in the solutions.

Another concept would be the opposite of the second version. With a stronger mutation for a generation located at a small area and a weak mutation for a wider spread generation. This could create elements located near a higher maximum of the fitness landscape when the generation is located in a lower maximum.

For a more complex problem like a sum of exponential functions the more complex method might perform better but has a longer calculation time.

6.4 Termination condition

The main routine of the code should not need a too long calculation time, therefore I limited the number of generations to a constant value. The algorithm stops after 15 generations. Watching the evolution after every generation showed that more generations are not needed. Other possibilities for a termination condition could be stopping the algorithm automatically if there is no significant change from one generation to another anymore.

6.5 The results of a set of data points

The complete main routine is always done 50 times. At the end there are 50 different values for “m” for the same set of data points. Then the code calculates the mean and standard deviation of this values and prints them on the screen as a final value.

7. The Results

The algorithm I created gives a value for the mass and a statistical inaccuracy as result. To compare the results with the normal fit method I did 3 difference Plots. In the x-y plane is always the mass value and the simulated numerical error. On the z axes are 3 different values which are:

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

$$\frac{\sigma}{m_{real}} \tag{7}$$

$$\frac{|m-m_{real}|}{\sigma} \tag{8}$$

All values from Eq. (6), (7) and (8) are better the closer they are to zero.

In all Plots the value from my algorithm is subtracted from the value of the normal fit method. Therefore whenever the curve is higher than zero means that my algorithm found a better value than the normal fit method. Whenever the curve is below zero the normal fit method was better. The plots are colorized with dark blue for small values of the z axes and strong red for high values.

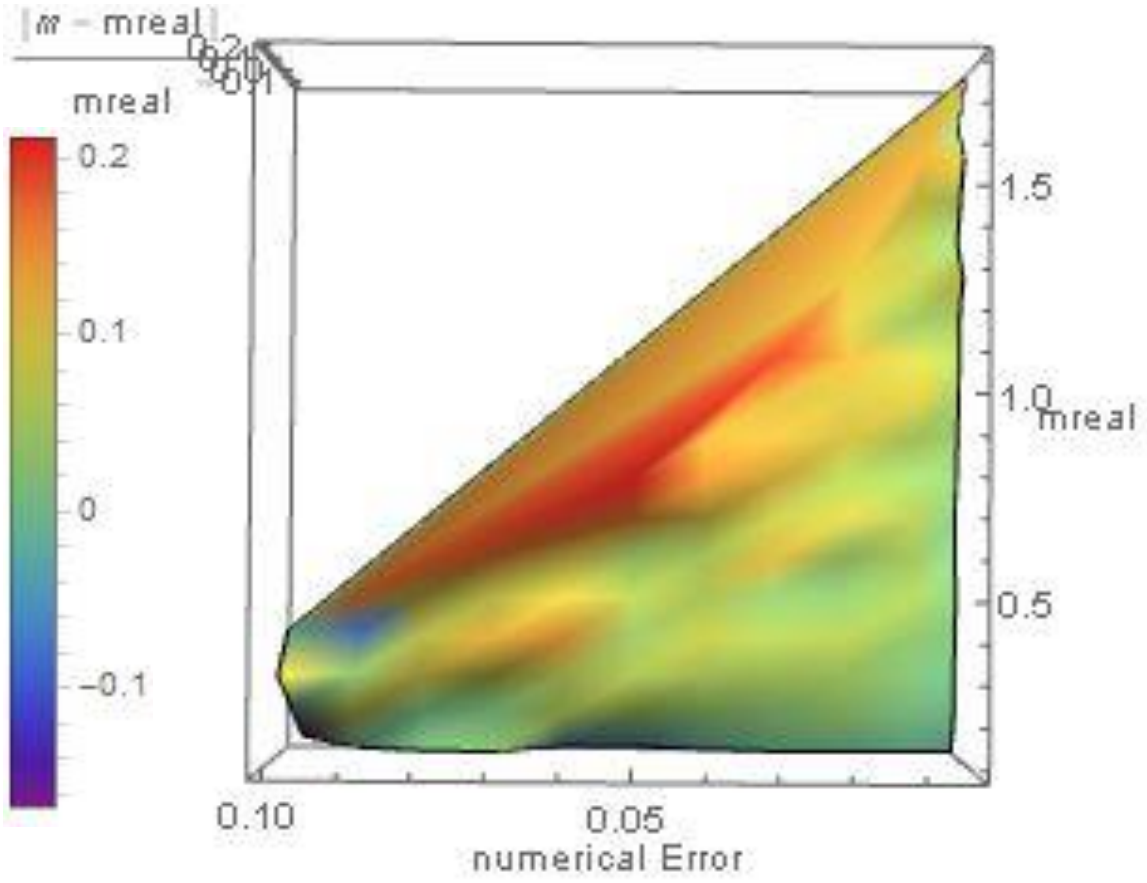

Figure 4 a)

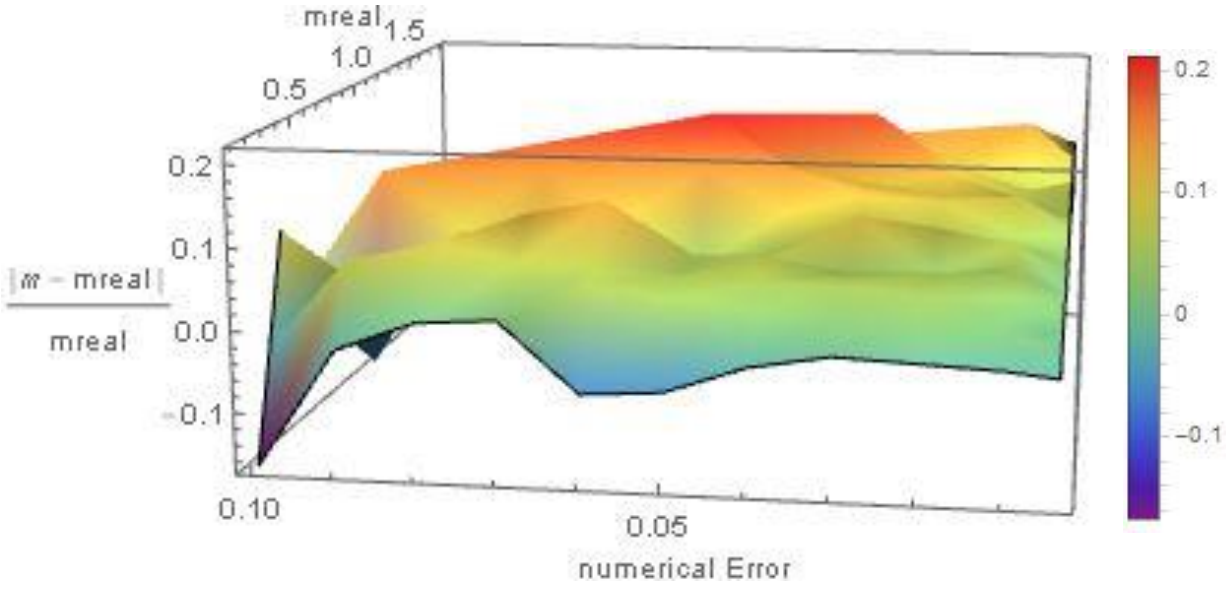

Figure 4 b)

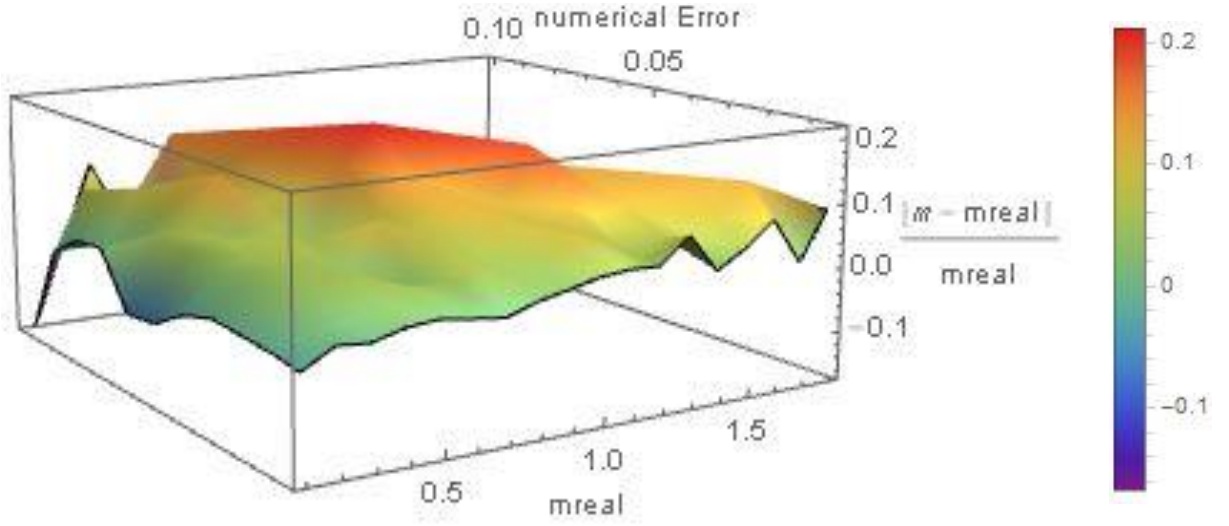

Figure 4 c)

Figure 4 a), b) and c) shows the results of Eq. (6). These Plots compare the distance between the real mass and the fitted mass relative to the real mass. Except for a small mass the algorithm finds better solutions independent of the numerical error.

For a simulated mass higher than 0.3 the curve of the plot for Eq. (6) is above 0. For a very small mass the normal fit method finds better solutions. For small masses there is always the maximum of 16 data points. So for the case of many data points the normal fit is the better option.

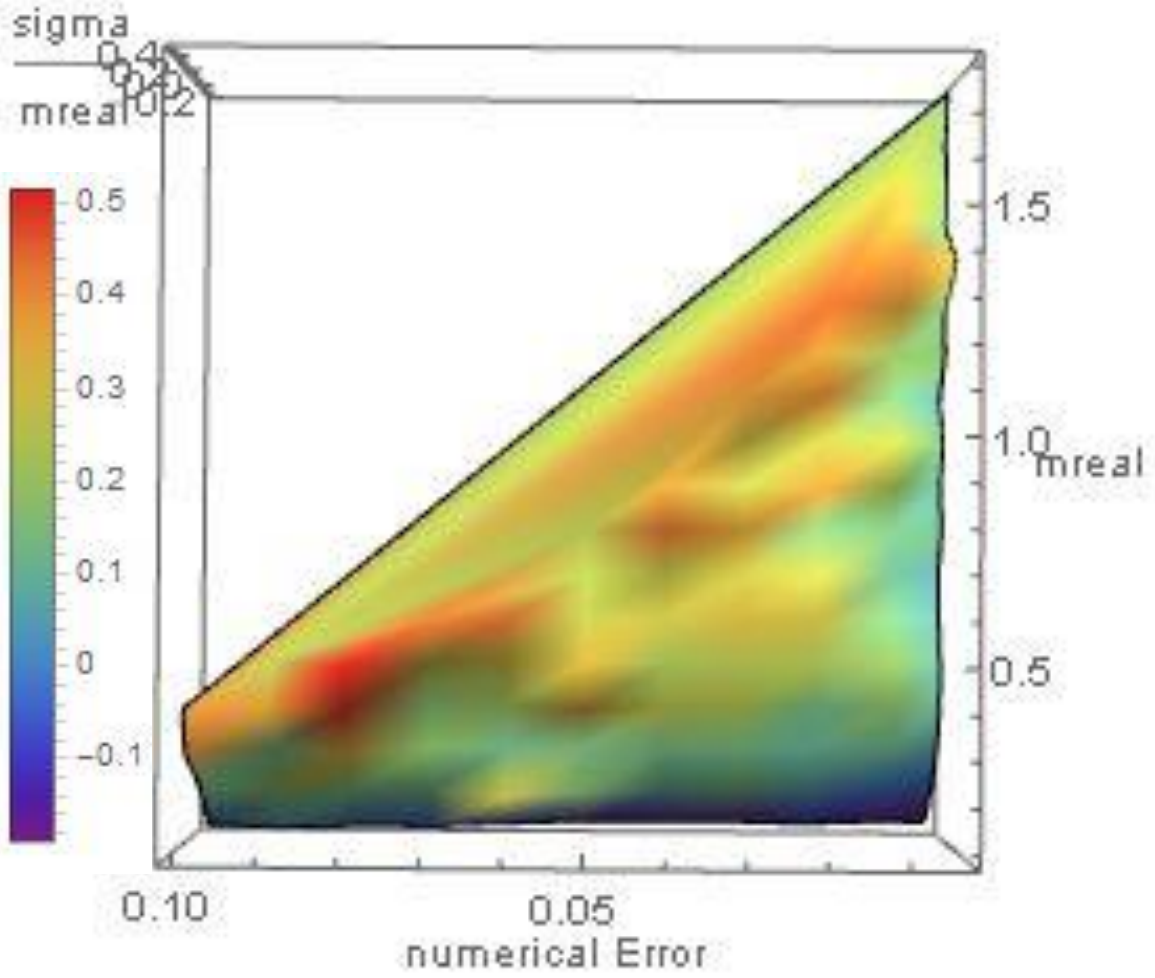

Figure 5 a)

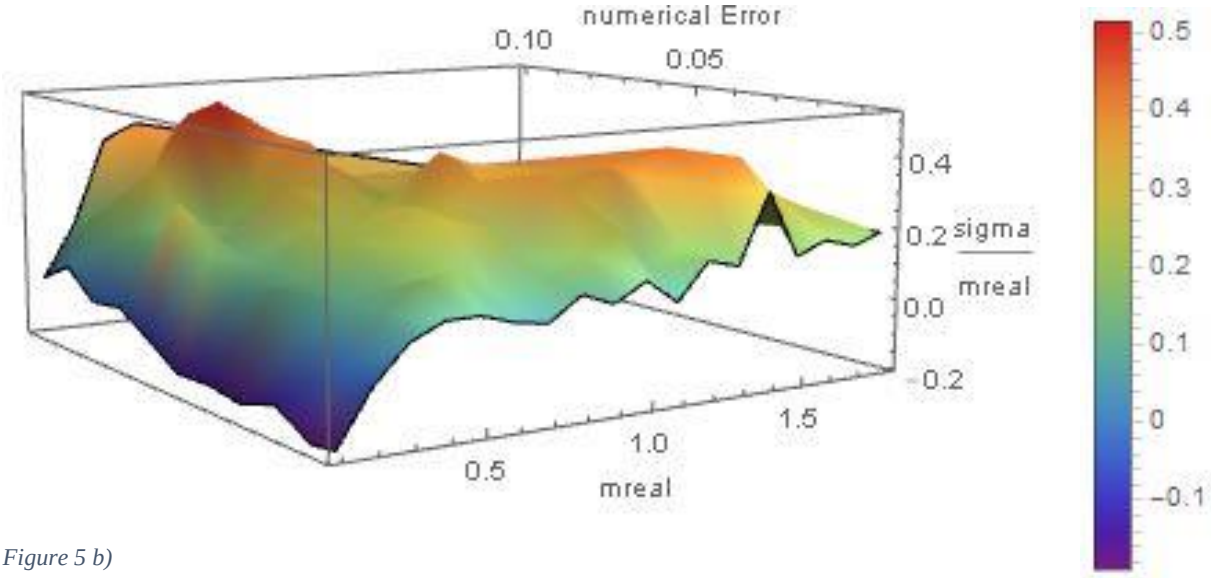

Figure 5 b)

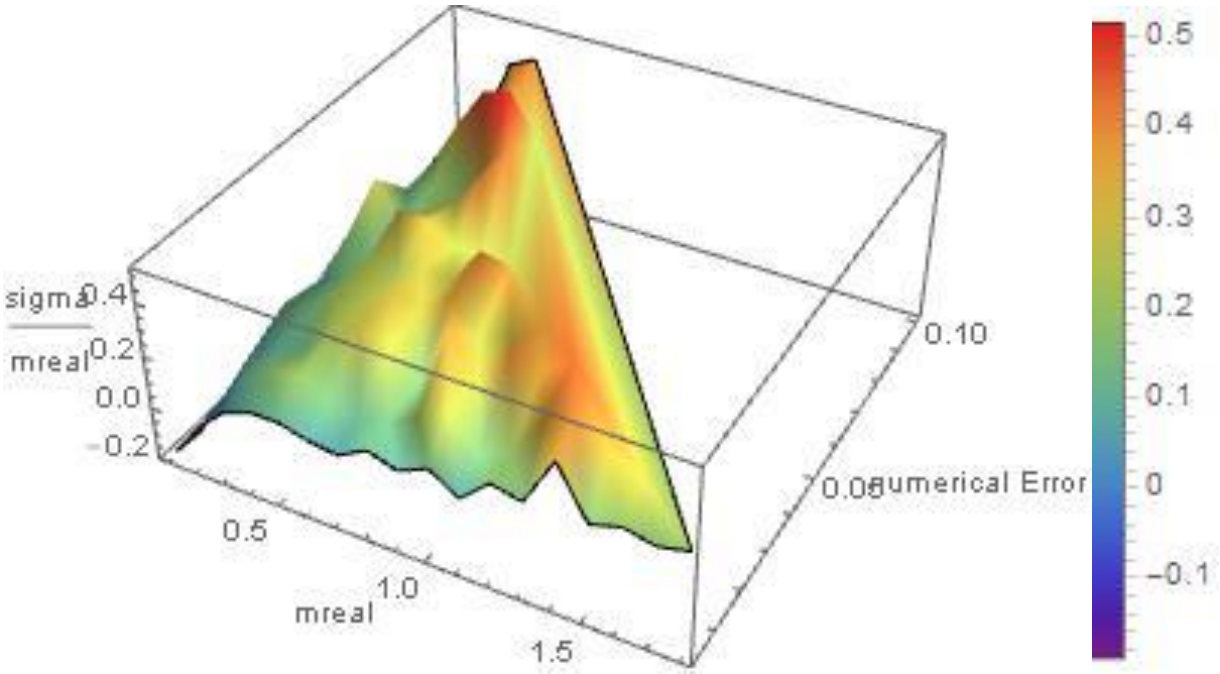

Figure 5 c)

Figure 5 a), b) and c) shows the results for Eq. (7). These plots compare the size of the inaccuracy of the fit compared to the high of the numerical error. Same as for Eq. (6), for a small mass the normal fit is better independent of the numerical error.

Same as for Eq. (6) the algorithm finds better solutions for Eq. (7) if the mass is higher than 0.3 independent of the numerical error. For a small mass the inaccuracy of the normal fit becomes very low. The inaccuracy of the algorithm does not get much smaller for low masses compared to higher masses.

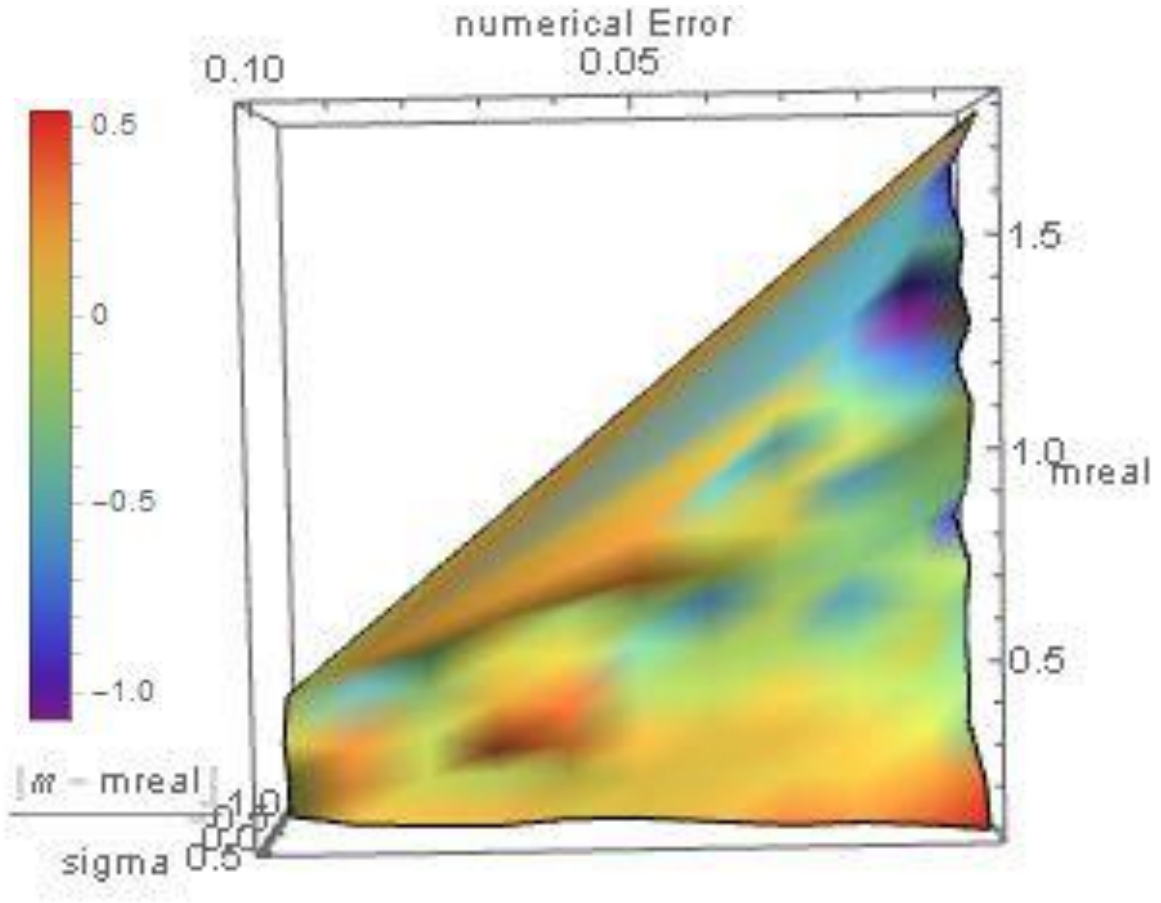

Figure 6 a)

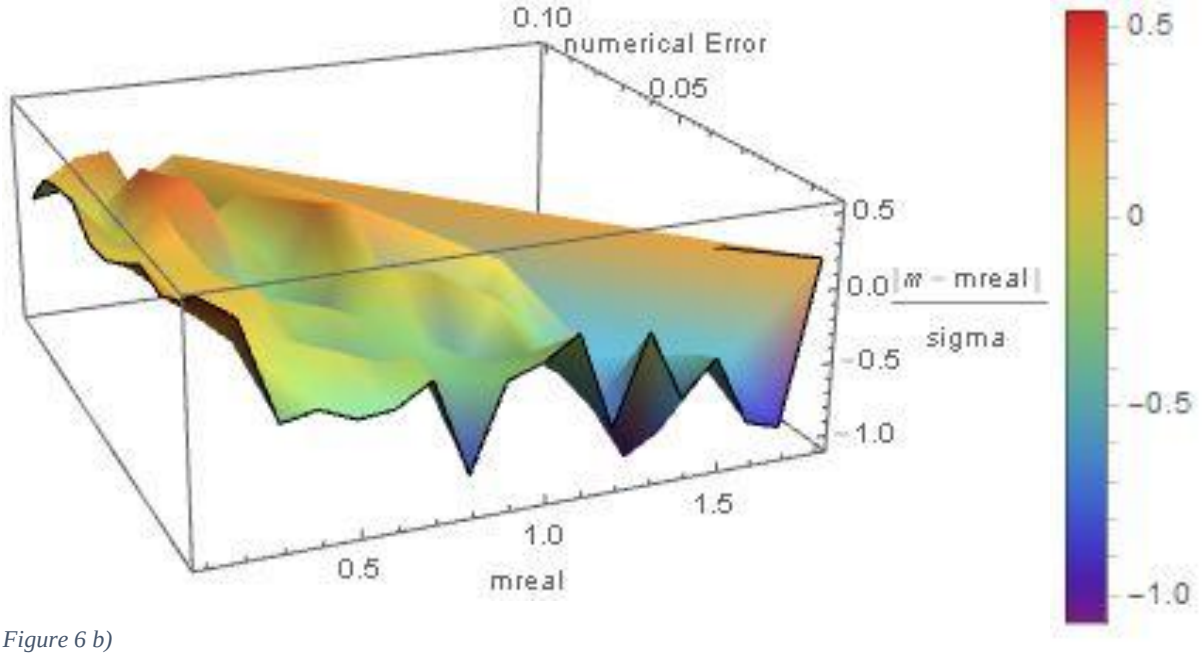

Figure 6 b)

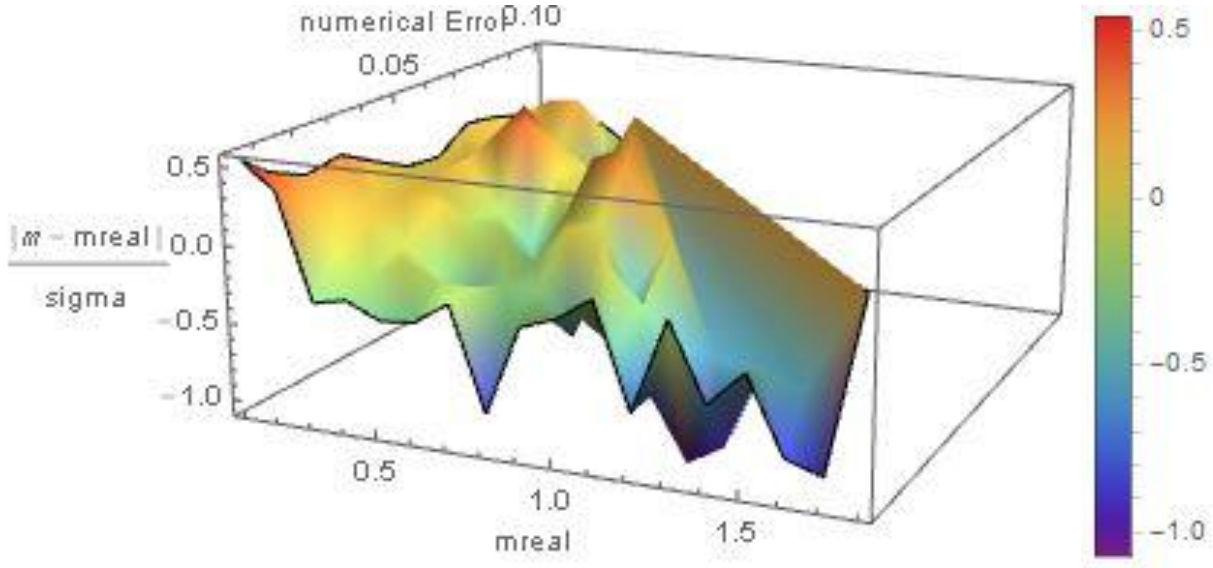

Figure 6 c)

Figuer 6 a), b) and c) shows the results for Eq. (8). These plots show how big the distance between the fitted solution and the real mass is compared to the inaccuracy of the fitted solution. For this Eq. the algorithm performs worse than the normal fit method.

For Eq. (8) the normal fit method is better than the algorithm. For nearly all combinations of mass and numerical error the plot values are below zero. Especially for high masses the algorithm performance gets worse.

8. Summary and Outlook

Figure 4, 5 and 6 from chapter 7 show that for some cases the algorithm performs well and in other cases the normal fit method is still better. For the fitted value the algorithm performs better for masses greater than 0.3. Independent for the mass and the numerical error the algorithm performs worse for Eq. (8).

This was only a first test of applying an evolutionary algorithm and I implemented just the basic concepts. Including more ideas and optimize existing parts may lead to a better general performance or a better performance where the algorithm performs worse than the normal fit at the moment. Especially the bad performance for high masses that you can see in Figure 7 is a problem. Nevertheless I think this concept has a potential to be a good solution for cases with only poor information about the real mass value. It would also be interesting to apply this concept on a more real situation with a sum of exponential functions like Eq. (2) and see how it performs compared to the normal fit method.

References

[1] A. E. Eiben and J. E. Smith, Introduction into evolutionary computing, Springer Science and Business Media, 2007

[2] David J. Griffiths, Introduction to Quantum Mechanics, Second Edition, Pearson, 2014