Semester Project in Computational Science II
at the University of Zürich

GAGL - A Genetic Algorithm for Gravitational Lensing

Author:
Simon Schwegler

Supervisor:
Dr. Jonathan Coles

29. August 2011

Inhaltsverzeichnis

1 Introduction

This report describes the implementation of GAGL, a program written in Python for reconstructing gravitational lensing images. I wrote GAGL as a semester project for the course Computational Science II at the University of Zürich. The document has the following structure: In the next section, a short description of the program GLASS is given, a simulation program for gravitational lensing, which was used to produce lensing data for GAGL. In the third section, a detailed explanation of GAGL can be found. The fourth section describes some optimizations to speed up the calculations. In the last section, the results of the algorithm are presented.

2 GLASS

GLASS was written by my assistant for the project at hand Jonathan Coles. Since GAGL is aimed at presenting the power a running genetic algorithm and not to work on actual experimental data, I solely used simulated results from GLASS as a testbed. Therefore, a detailed explanation of GLASS together with the process of gravitational lensing in a nutshell is given.

2.1 Gravitational lensing

Consider a point light source at a distance (x_s, y_s, z_s) relative to earth and a gravitational lens centered at the point $(0, 0, z_l)$ relative to earth. A massive object like a black hole or a galaxy can act as such a gravitational lens. The mass of the lens is modelled as a set of N mass points $(x_i, y_i z_i)$, $i = 0, ..., N$. Assuming that the spatial and mass distributions of the particles are fixed we have 4 free parameters for this situation. If light is coming from the light source to the earth, the light rays are deflected near the massive lens. Thus the point light, as observed from the earth, appears multiple times at different locations on the sky. Also, since the travelling distance for the different light rays is different (because they move on trajectories with different lengths), the rays will appear at different times at the sky.
Figure 2.1 shows an example image generated by GLASS with the parameters $(x_s = 0.026, y_s = 0.0057, z_s = 1.6, z_l = 0.4)$ The impact position of the deflected rays is symbolized by the green transparent circles and the subjacent color distribution shows the mass density of the lensing galaxy projected onto the x-y plane (since the observer is defined to be looking along the positive z-axis). Note that on this picture, only the spatial differences but not the different impact times of the rays are visible.
Let us assume that we have such an image either from observation or from a GLASS simulation (the *reference image*) but we do not know (or pretend not to know) the parameters (x_s, y_s, z_s, z_l) leading to this very image. The goal is now to adjust the parameters such that GLASS produces an outcome which is as similar as possible to the observed or simulated lensing image (what 'similar' exactly means in this context will be defined in subsection 3.1). For such an optimization problem, a number of different algorithms is available, one of them is the so called genetic algorithm, which is the core of GAGL and will be discussed in detail in the next section.

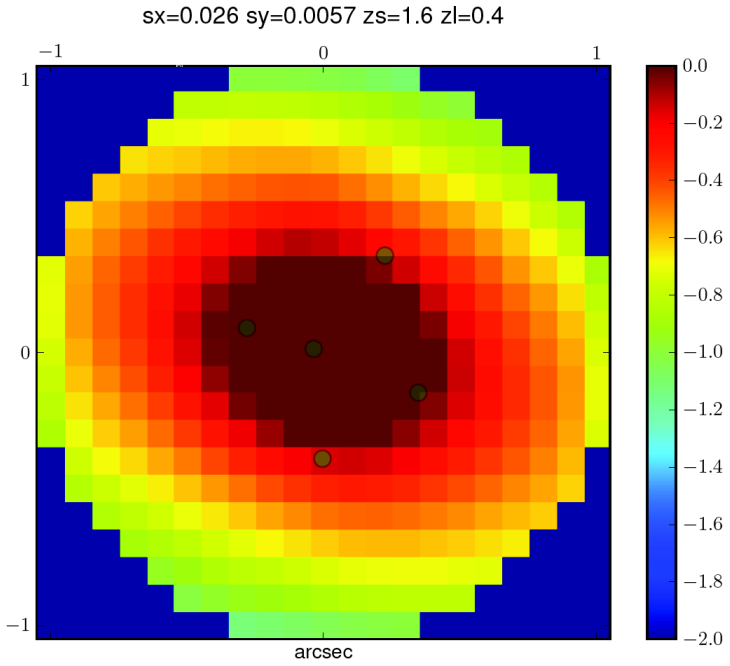

Abbildung 1: Lensing image for $(x_s = 0.026, y_s = 0.0057, z_s = 1.6, z_l = 0.4)$

2.2 Using GLASS

Since GAGL and GLASS are both written in Python, calling the GLASS routines within GAGL is straightforward. Here is a short explanation of the program code used to generate gravitational lensing images:
First, we have to tell GLASS how the mass distribution of the lens looks like and read these mass points into `X`, `Y`, `Z`, `M` :

```
file = 'bigsim.txt.npz'
data = load(file)
X = data['arr_0']
Y = data['arr_1']
Z = data['arr_2']
M = data['arr_3']
```

Then we set the age of the universe `g` and the cosmological constants:

```
g = 13.7
omega(0.3,0.7)
hubble_time(g)
```

Now we generate the lens at the position z_l and a source at z_s, which are both free parameters, as explained in the previous subsection. `pixrad` determines the resolution of the simulation.

```
obj = globject('my lens')
zlens(zl)
pixrad(10)
```

```
maprad(1.0)
src = source(zs)
```

Finally, we create a model with the loaded data and start the simulation process, where we set the source at the position x_s, y_s

```
model(1, mode='particles', data=[X,Y,M, [[sx,sy]], g])
d = raytrace([env().models[0], 0,0])
```

After the simulation, we can get the output:

```
output = observables(env().models[0], 0,0, d)
```

3 GAGL

The core algorithm of GAGL is a genetic algorithm, an instance of an optimization algorithm: Optimization algorithms are aimed at finding the minimum of a given real-valued function $f(\vec{x})$, called the *cost function*. $\vec{x}$ can be a vector of any dimension and is called the *chromosome* in the context of genetic algorithms. The individual entries of the chromosome are called *genes*
A genetic algorithm tries to minimize the test function by mimicking the process of natural evolution: We start with an initial 'population' of M *individuals* $\vec{x}_i$, $i = 1, ..., M$ and let them 'evolve' in a specific way such that better and better solutions (i.e. solutions with smaller and smaller *cost values* $f(\vec{x}_i)$) are generated. This artificial evolution process is driven by the concepts of artificial *reproduction*, *selection*, *mating* and *crossover* which are also important processes in natural evolution. Additionally the purely artificial concept of *elitism* is used. All these ideas will be discussed in detail in the coming subsections.

3.1 The cost function

In the setting of gravitational lensing, the cost function to be optimized is a function of the vector $\vec{x} = (x_s, y_s, z_s, z_l)$ with the values as defined in section 2.1. $f(\vec{x})$ is defined as a measure of the 'similarity' between the lensing image generated by GLASS with the lensing parameters given in $\vec{x}$ and the observered reference image. Of course the definition of 'similarity' is somewhat arbitrary and the only restriction is, that if the reference- and the GLASS-image for an individual $\vec{x}$ are equal, $f(\vec{x})$ has to be zero and that $f(\vec{x}) > 0$ for any other $\vec{x}$.
The outcome of a GLASS simulation is a set of points at different positions with different impact times. Additionally, the points can be of one of three types, either a minimum (`'min'`), a maximum (`'max'`) or a saddle point (`'sad'`). Here is an example output of such a simulation, produced by the functions `lensing` and `extract` with the parameters ($x_s = 0.026, y_s = 0.0057, z_s = 1.6, z_l = 0.4$)

```
[[0.22498714050266264, 0.35474452005406343, 'min', 0.0],
[-0.0037734221456084821, -0.38873981968811877, 'min',
1.0475270667210261],
[0.34730162211175147, -0.14730322372258209, 'sad',
1.5927977751140641],
[-0.27997431830785352, 0.089914948365187153, 'sad',
3.4579593352196563],
```

```
[-0.035920440300767971, 0.013239656740311497, 'max',
1.2968797725800121]]
```

We have a list of lists, where each inner list contains the data of one image point: The x and y coordinates of the image points in the first and second field, the type of the point in the third field and the difference between the impact time of this point and the impact time of the point impacting first of all in the fourth fild. Note that the first point in the list is always the one with the first impact, such that its time difference will always be zero.
The cost function is now defined as follows: If we have a reference image with N points at positions $(x_i, y_i) = \vec{r}_i$ and time differences t_i, $i = 1, ..., N$ and an image generated from an individual with M points at positions $(x_j\prime, y_j\prime) = \vec{r}_j\prime$ and time differences $t_j\prime$, $j = 1, ..., M$ the cost is defined as:

$$f(\vec{r}_i\prime) = (\sum_{n=0}^{L} |\vec{r}_n - \vec{r}_n\prime| + |t_n - t_n\prime|) + \Delta_{num}^3 \tag{1}$$

where $L = min(M, N)$, because it can happen that the reference and test image have not the same number of points and $\Delta_{num} = M - N$. Δ_{num} is taken to the third power because a different number of lensing points indicates a very bad solution and is thus punished by a high cost value.

3.2 Selection

Now that we have defined a cost function, the genetic algorithm can be started: We begin with an initial *population* of M individuals $\vec{x}_i$, $i = 1, ..., M$. We can then compute the cost of each individual $c_i := f(\vec{x}_i)$, resulting in a set of tuples $(\vec{x}_i, c_i)$. We now order these tuples in descending order of c_i. A certain ratio `xrate` of this ordered population is then chosen to come into a *mating pool*, meaning that only the first `xrate` $* M := N_{keep}$ individuals are kept and used to produce *offspring* for the next generation.
As in nature, the lower the cost (the higher the fitness) of the indivual, the higher the chance should be to reproduce. In GAGL this is modelled by rank weighting [1]: Choose the N_{keep} best individuals and compute the values

$$P_n = \frac{N_{keep} - n + 1}{\sum_{n=1}^{N_{keep}} n} \tag{2}$$

Then compute the cumulative sum

$$C_n = \sum_{i=1}^{n} P_i \tag{3}$$

Note that

$$\sum_{i=1}^{N_{keep}} C_i = 1 \tag{4}$$

The selection now works as follows: We choose a random number $r \in (0, 1)$ and pick the first individual $\vec{x}_i$ in the mating pool with $C_i > r$ (beginning at the

individual with the smallest c_i). With this method, individuals with a smaller cost are more likely to be selected.
The N_{keep} individuals in the mating pool will now produce in total $M - N_{keep}$ offspring which replace the $M - N_{keep}$ indivduals which did not make it into the mating pool.
To this end, we choose $\frac{M-N_{keep}}{2}$ *couples* from the mating pool and let them mate. One individual of a couple is called the *mother*, where the other one is the *father*. Each couple produces two offspring, which replace individuals that are not in the mating pool.
To choose one couple, we generate two random numbers r_{mother} and r_{father} and perform the rank weighting selection with both of these.

3.3 Mating and crossover

Mating is the process of combining the chromosomes of both mother and father in such a way, that the features of both parents are mixed and that also an additional random information is introduced. This process is called *crossover*: We choose a random integer number $x \in [1, 5]$ which indicates the *crossover gene* in the chromosomes of the mother and the father.
Let us denote the k'th gene in the chromosome as g_k, meaning $(x_s, y_s, z_s, z_l) \equiv (g_1, g_2, g_3, g_4)$ in our case. Then the two offspring $\vec{o_1} = (o_1, o_2, o_3, o_4)$ and $\vec{o_2} = (o'_1, o'_2, o'_3, o'_4)$ are generated from the mother $\vec{m} = (m_1, m_2, m_3, m_4)$ and the father $\vec{f} = (f_1, f_2, f_3, f_4)$ as follows:

1. All the chromosomes of offspring 1 to the left of the crossover point are taken (copied) from the mother, whereas the chromosomes the the left of the crossover point of offspring 2 are taken from the father

2. All the chromosomes of offspring 1 to the right of the crossover point are taken from the mother, whereas the chromosomes to the left of the crossover point of offspring 2 are taken from the father.

3. The gene at the crossover point in the offspring is chosen to be in between both the crossover gene of the mother and the one of the father:
   Take a random number $r \in (0, 1)$. If x denotes the index of the crossover gene:
   $$o_x = m_x - r * [m_x - f_x] \tag{5}$$
   $$o'_x = f_x + r * [m_x - f_x] \tag{6}$$

3.4 Mutation

The infimum and supremum with respect to the random number $r \in (0, 1)$ for the offspring's crossover genes are (assuming without loss of generality $m_x > f_x$):

$$\sup_r o_x = m_x - 0 \cdot [m_x - f_x] = m_x \tag{7}$$

$$\inf_r o_x = m_x - 1 \cdot [m_x - f_x] = f_x \tag{8}$$

$$\inf_r o'_x = f_x + 0 \cdot [m_x - f_x] = f_x \tag{9}$$

$$\sup_r o'_x = f_x + 1 \cdot [m_x - f_x] = m_x \tag{10}$$

Meaning that the offspring can, for a fixed crossover gene index i, never attain values bigger than the biggest corresponding gene of all the parents in the mating pool and never attain values smaller than the smallest corresponding gene of all the parents in the mating pool, which will quickly lead the algorithm into local minima. A possible solution to this problem is the introduction of *mutations*: We first fix a parameter $R_{mutations} \in (0, 1)$ which denotes the ratio of the total number of genes in the population that are going to be mutated. If we have N_{gene} genes per individual and M individuals, we then randomly choose $N_{mutations} = R_{mutations} \cdot N_{gene} \cdot M$ genes and set the value of each of them randomly within the allowed interval of this gene. This method prevents the algorithm from converging towards local minima, but can also introduce too much randomness, so a careful fine tuning of the parameter $R_{mutations}$ is essential.

3.5 Elitism

It can happen that one single offspring attains a very good cost value which is then immediately destroyed by the random mutation process before the offspring can pass its genetical information to the next generation. To prevent this situation, the mutation takes place such that the best N_{elite} individuals simply cannot be mutated. In GAGL we set $N_{elite} = 1$.

3.6 Overall algorithm

Now we have all the ingredients to assemble the full genetic algorithm:

1. Create an initial random population of M individuals
2. determine the cost c_i of each individual $\vec{x}_i$
3. sort the population in descending order of costs
4. (as long as the smallest cost is above a fixed threshold:)
   `while smallest_cost > threshold`:
   (a) Take the first N_{keep} best (first) individuals in the population and compute their C_i (cumulative sum) values
   (b) (Perform $\frac{M-N_{keep}}{2}$ matings:)
   `for j in n_matings`:
      i. Select a mother and a father from the first N_{keep} individuals using ranked weighting selection
      ii. Create the two children using crossover
      iii. Replace the individual at $N_{keep} + 2 * j$ with the first child and the individual at $N_{keep} + 2 * j + 1$ with the second child
   (c) Mutate $N_{mutations}$ randomly chosen genes in the population
   (d) determine the cost c_i of each individual $\vec{x}_i$
   (e) sort the population in descending order of costs
5. Output the best individual as the solution of the algorithm

4 Optimizations

In order to calculate the cost function of an individual $\vec{x}_i$, one has to start GLASS to comupute the lensing image. Since this simulation is computationally heavy and has to be started for *every* individual, it is crucial to minimize the number of cost function evaluations until an acceptable minimal cost is found. To this end, GAGL uses a number of optimizations techniques which are described in this section

4.1 Parallelization

Parallelization does not reduce the number of cost evaluations but makes them faster: Instead of running only one cost evaluation at once, as many cost evaluations are started as there are processors on the computer. Since most office machines have already at least four processor cores, this means approximately a speedup of a factor 4!
Assuming that we have n_{proc} processors and M cost values of the M individuals to evaluate, each processor gets

$$f(n) = \lfloor \frac{j}{p} \rfloor + \begin{cases} 1, \text{if } i < j \bmod p \\ 0, \text{if } i \geq j \bmod p \end{cases} \tag{11}$$

chromosomes to evaluate, where $\lfloor \cdot \rfloor$ is the floor function. This guarantees an optimal load distribution over the processors.

4.2 Avoiding unnecessary cost evaluations

The N_{keep} best ranked individuals (at the beginning) of a generation do not always have to be reevaluated at the end of this generation, since they are not changed during the mating process. However it is still possible that they will be changed during the mutation process (except for the first ranked, elitized individual). We therefore keep track of all the individuals that are either replaced by offspring during the mating or are changed during the mutation process. All the other individuals need not be reevaluated.

4.3 Meta Optimization

A number of parameters of the genetic algorithm can be chosen more or less arbitrarily, among them are the mutation rate $R_{mutation} \in [0, 1]$, the population size M, the number of individuals N_{keep} that make it into the mating pool and the number of elitized individuals during the mutation. Since all these parameters can have an impact on the convergence rate of the algorithm, it is indicated to optimize these parameters for maximal convergence rate. So we are optimizing the parameters of the optimization algorithm itself, hence the term meta optimization.
We define the convergence rate to be the number of cost evaluations until the best individual has a cost value below a certain threshold. In our case, we set the threshold to 0.2, which is equivalent to a relative error of about 1% for each of the points in the image.
Note that other definitions are possible, for example that the average cost of all

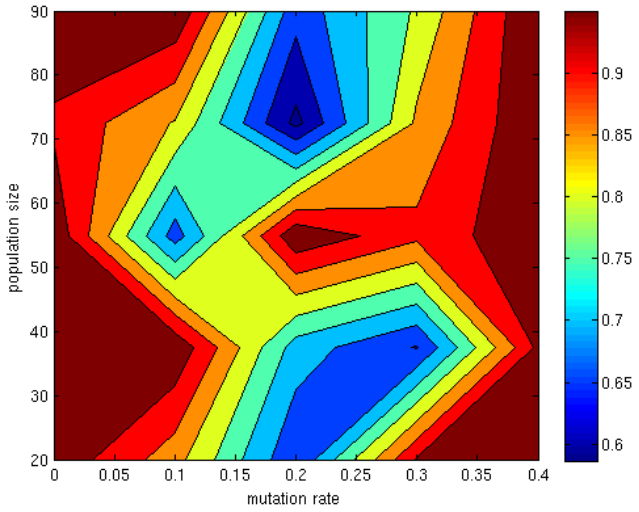

Abbildung 2: Number of evaluations to reach a fitness below 0.2 as a function of $R_{mutation}$ and M (normalized to 1)

the individuals in a population drops below a threshold.
Instead of varying all the four parameters $R_{mutation}$, M, N_{elite} and N_{keep}, we restrict ourselves to $R_{mutation}$ and M. First because they are expected to have the most impact on the performance of the algorithm [1] and second because already optimizing these two parameters is an extremely computationally expensive task. Therefore we use only a very coarse grained exhaustive search where we take 5 values within $M \in [20, 90]$ and $R_{mutation} \in [0.0, 0.4]$ respecitvely. GAGL is then run for each of the sample points until either the minimal cost drops below 0.2 or a maximum of 5000 cost function evaluations is reached. To smooth statistical noise, we sample each point 5 times and take the average. Figure 2 shows the resulting landscape $N_{evals}(R_{mutation}, M)$ where N_{evals} is the number of evaluations until the minimal cost drops below the threshold. Figure 3 shows the landscape $cost_{min}(R_{mutation}, M)$ where $cost_{min}$ is the minimal cost in the population at the termination of the algorithm. Figure 4 is the addition of both landscapes which is used to determine the optimal meta parameters. For these figures, $(x_s = 0.0160, y_s = 0.0047, z_s = 1.5, z_l = 0.5)$ was used as the reference, and we see from 4 that $M = 74$ and $R_{mutation} = 0.2$ are the parameters with the best performance rate. Therefore, they are used in GAGL. Note that because of the huge effort to compute the meta-cost values, these figures are extremely crude and certainly a field of further investigation.

4.4 Twin ellimination

After several generations and several mating processes, twins can appear in the population: Individuals with all the same genes. Either this case can be detected and the cost function for these twins is only evaluated once, or twins can be elliminated after the mating. To this end, we go through the list of indiviuals and check for each of them, if one or more twins exist in the list. The genes of

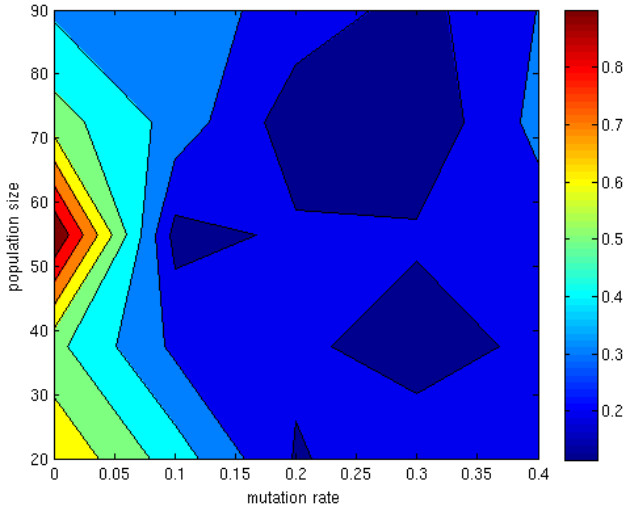

Abbildung 3: Lowest cost at termination as a function of $R_{mutation}$ and M (normalized to 1)

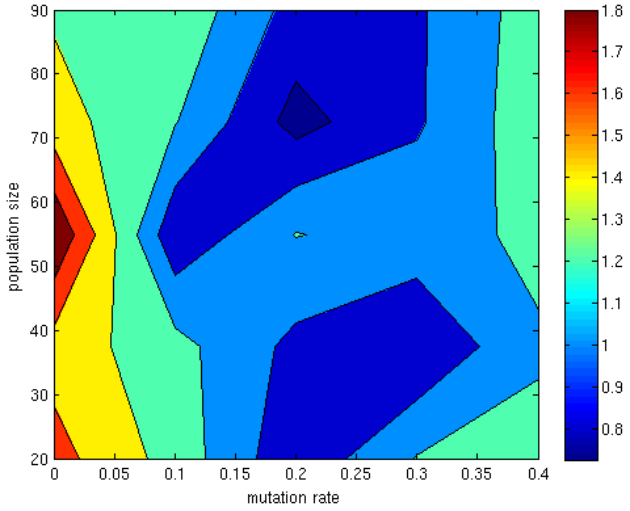

Abbildung 4: Addition of Figure 2 and Figure 3

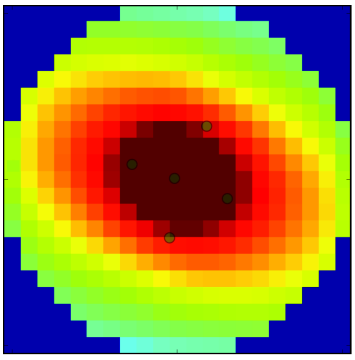

Abbildung 5: Lensing image for ($x_s = 0.016, y_s = 0.0047, z_s = 6,0, z_l = 1.0$)

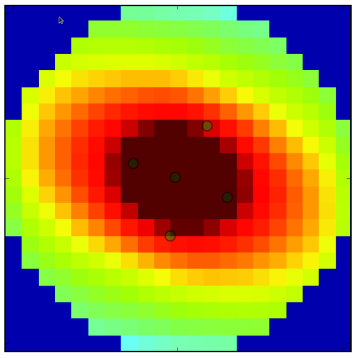

Abbildung 6: GAGL's reconstruction of the reference ($x_s = 0.016, y_s = 0.0047, z_s = 6,0, z_l = 1.0$)

the twins are then all reset to random values.

5 Results

In this section, the results achieved by GAGL for the reference ($x_s = 0.016, y_s = 0.0047, z_s = 6.0, z_l = 1.0$) are presented.
A picture of the reference image is given in Figure 5. Figure 6 shows a the corresponding reconstruction found by GAGL after 4623 cost evaluations with a cost value of 0.10273900079447972. The search space was restricted to $x_s \in [0.0, 0.16]$, $y_s \in [0.0, 0.047]$ $z_s \in [0.0, 10.5]$, $z_l \in [0.0, 5.0]$. Figure 7 shows the minimum cost in the population as a function of the number of cost function evaluations for five different runs of the algorithm. To illustrate the evolution of the population towards the correct solution, I sampled the cost function for fixed ($z_s = 6.0, z_l = 1.0$) and varying (x_s, y_s) with 30 times 30 steps each. A contour plot of the resulting cost landscape is shown in Figure 8 . The same sampling can be performed vice versa, where (z_s, z_l) are varied and (x_s, y_s) fixed which leads to the landscape shown in Figure 9. The light grey area at the right bottom of the image is the physically invalid region where $z_l \geq z_s$ (i.e. the lens is *behind* the source as seen from earth). All individuals in this region are punished with a very high constant cost value.

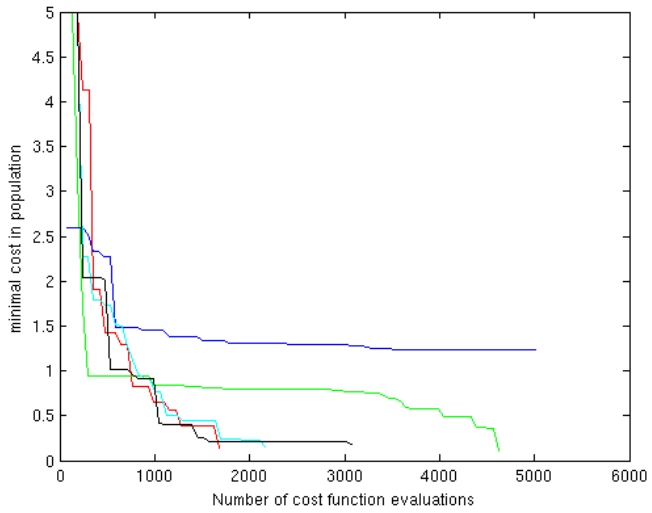

Abbildung 7: Minimal cost as a function of the number of cost function evaluations

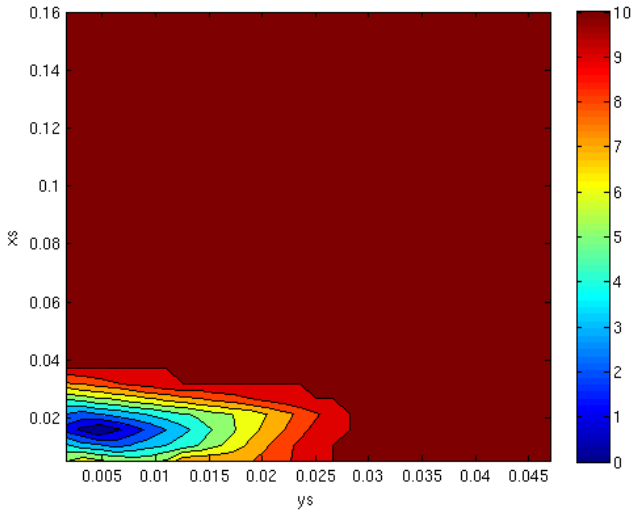

Abbildung 8: Cost landscape for $(z_s = 6.0, z_l = 1.0)$ and varying (x_s, y_s) (values truncated to 10.0 for better visual appearance)

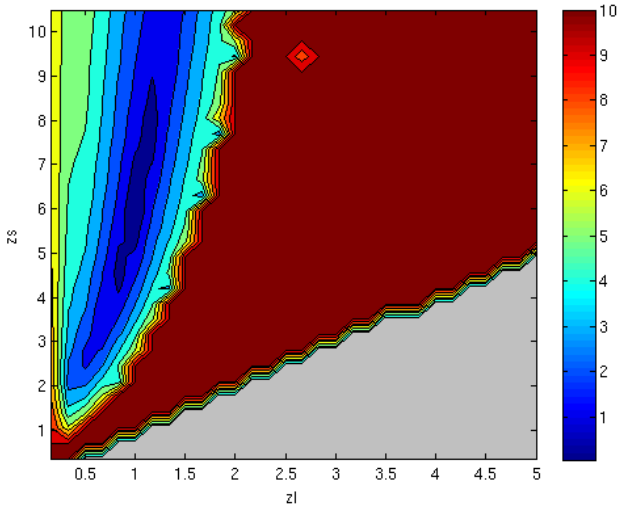

Abbildung 9: Cost landscape for $(x_s = 0.016, y_s = 0.0047)$ and varying (x_s, y_s) (values truncated to 10.0 for better visual appearance)

We can now take the chromosomes of all the members of the population and project them onto the landscape with fixed $(z_s = 6.0, z_l = 1.0)$ and fixed $(x_s = 0.016, y_s = 0.0047)$ respectively. The resulting plots 10 to 16 are an illustration of the evolution process as well as a visual check whether the individuals are converging to the reference point. The position of each individual is marked as a red circle and the reference solution as a green cross. Note that in Figure 15, the algorithm approached a local minimum, but finally managed to switch to a better minimum in Figure 16.

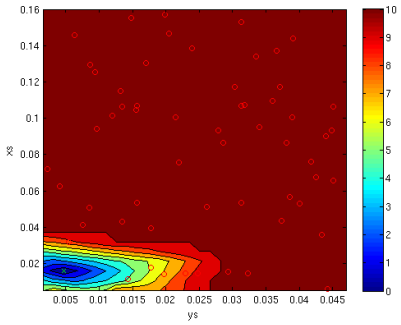

Abbildung 10: Population after 0 generations projected onto $(z_s = 6.0, z_l = 1.0)$ plane

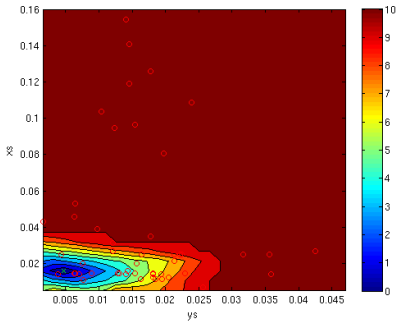

Abbildung 11: Population after 5 generations projected onto $(z_s = 6.0, z_l = 1.0)$ plane

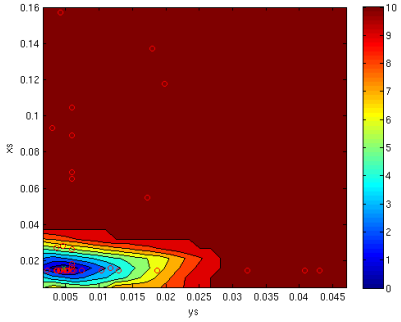

Abbildung 12: Population after 46 generations projected onto $(z_s = 6.0, z_l = 1.0)$ plane

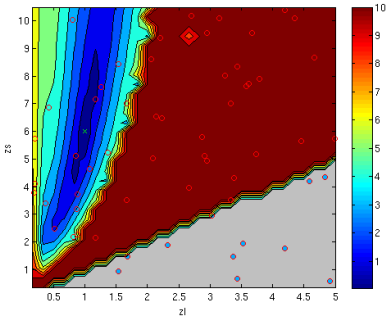

Abbildung 13: Population after 0 generations projected onto $(z_s = 6.0, z_l = 1.0)$ plane

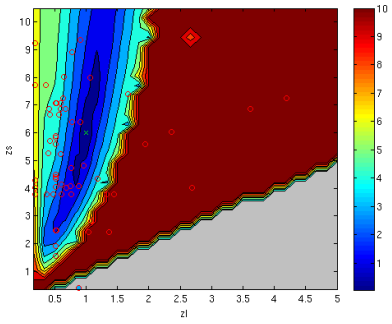

Abbildung 14: Population after 5 generations projected onto $(z_s = 6.0, z_l = 1.0)$ plane

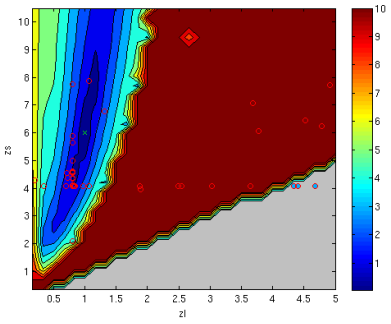

Abbildung 15: Population after 60 generations projected onto $(z_s = 6.0, z_l = 1.0)$ plane

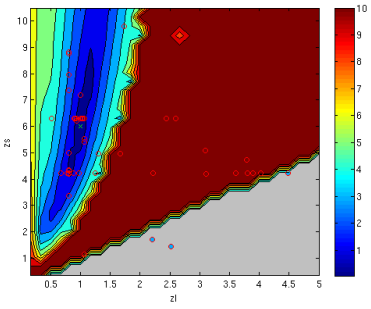

Abbildung 16: Population after 79 generations projected onto $(z_s = 6.0, z_l = 1.0)$ plane

6 Acknowledgements

I would like to thank Jonathan Coles for a lot of helpful discussions and advice during the project and plenty of hints about Python programming.
I would also like to thank my fellow student Simon Obrecht for a timesaving hint about the processor load distribution in GAGL.

Literatur

[1] Randy L. Haupt, Sue Ellen Haupt, *Practical Genetic Algorithms*, John Wiley & Sons, 2004.

[2] H. J. Herrmann, *Physik auf dem Computer I & II*, Lecture Script, ETH, WS 06/07.

[3] Brendon J. Brewer, Geraint F. Lewis, *When Darwin Met Einstein: Gravitational Lens Inversion with Genetic Algorithms*, University of Sydney, 2006.