Exploring Different Optimization Techniques Through the Lens of Moser's Worm Problem

James Wenk

April 22, 2022

1 Introduction

In this program we explore several methods concerning computing quantities related to Moser's Worm Problem.

Moser's Worm problem is a notoriously difficult problem in geometry, which asks what is the shape of smallest area that contains every unit length curve after translation and rotation. The difficulty of the problem cannot be overstated, and in general even the question of asking if a set contains every unit length curve is intractable. As such we should not expect any of our methods to converge to or yield the optimal answer.

The first hurdle to solving the question is that the number of unit length curves is infinite so we need to discretize the problem to be able to try to optimize. We do this by discretizing space into squares, and curves into polygonal segments, where angles are a multiple of some base angle. Now in this discretization, everything is finite and so we can describe curves and our containing set, the number of possibilities for both grow exponentionally making brute force search intractable. Thus we developed several methods to attempting to try to optimize, but before we optimize, we need a function.

We can frame the Worm Problem as an optimization problem as follows, minimize over all sets the area of that set plus the maximum over all curves of the minimum over all linear transformations of the lenth of the curve not lying in our set. Indeed, the optimal set will be a minima with value equal to that of the area of the set. To find this minima, we attempt several methods, which we describe below. The most straightforward, which would be to simply nest optimizers is due to the large computational cost of optimizing becoming exponentially larger if we include the function as a sub call. As such this method was deemed to computationally expensive and we opted for a iterative approach where we try to find the min, max, and min iteratively.

2 Nelder-Mead Method

In the following methods we will refer to several variables, as they appear in the program. First is the set S, and this is the quantity we are interested in. We want that S will contain all curves, and want to minimize the size of this set. Secondly we have gamma (γ), which will denote the current curve we are considering. Ideally we will have that gamma lies inside of S, namely $\gamma \subseteq S$. Now since we are only interested in curves up to isometries of the curve, we also consider such linear transformations given by T. We then denote the application of T to gamma by $T(\gamma)$.

In the following methods we will always discretize the initial bounding set by a grid of size $NUM\_POINTS\_SET \times NUM\_POINTS\_SET$ for $NUM\_POINTS\_SET = 25$. In addition curves γ will be discretized by N segments for $N = 10$. The first method we used was to to try the pre-packaged solvers inside of the scipy package. Since optimization can be quite computationally expensive, as well as our functions not having a nice gradient, we started out with the Nelder-Mead method of optimization. The results can be seen in Figure 1. As one can see, there are several main problems. First, and most importantly, is that the resultant set S looks nothing like what we want it to, as we want something that is contiguous. This arises as everytime we update our set S we completely forget the last one and only optimize for the new curve γ. An idea to fix this is presented in *bbopt2*, by damping the output.

The second problem is that we don't have convergence. Now while we have only run for 100 iterations, which is a small number, we can see that the error has several large spikes. This may be due to the local search nature of Nelder-Mead not being able to exploit the true geometry of our function, and most likely is exacerbated by running the optimization calls in sequence, in which the small steps given by Nelder-Mead are able to be accounted for by the resultant change in γ. This is also indicated by the resultant set S, which has the vast majority of the point at 0.5, which is their starting values. As such, to address these problems we can first consider the output using a different method, and then move on to *bbopt*2.

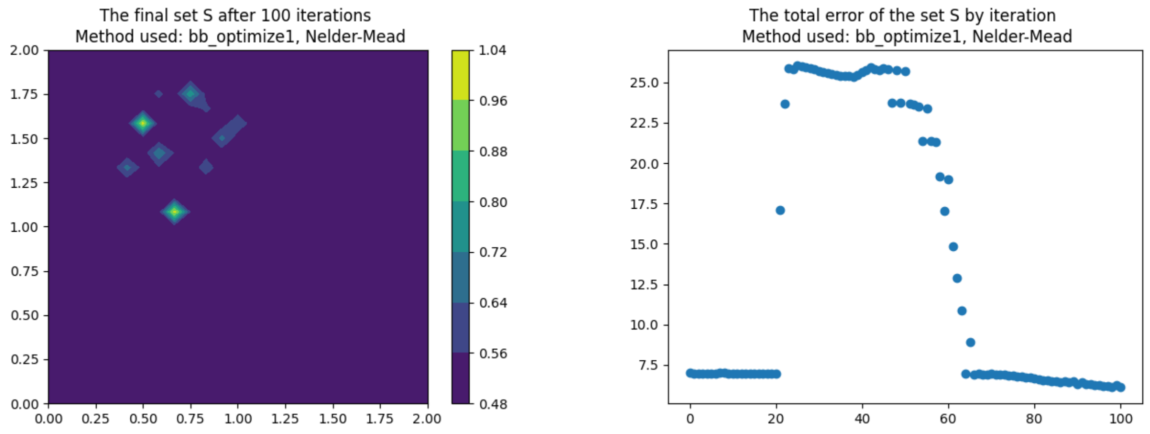

Figure 1: This shows the output of *bbopt*1 for 100 iterations with Nelder-Mead method

Further, one might ask why we are starting at the set S having an initial choice of all values 1/2. This is due to the nature of the Nelder-Mead algorithm not playing nicely with boundaries. Namely we want our curve γ to stay inside our bounding box of $[0, 2] \times [0, 2]$, and we want the values of S to be between 0 and 1. To do this we have to apply a penalty to terms outside of the bounding box to nudge our solution back in if we ever go outside. However, starting at the all 0's solution means that we are at a vertex of our solution space, which looks like a cube in some high dimensional space. Thus we are quite likely to move out, and when we do it will take time to move back. This can be seen in Figure 2 where the moment we take a step out of bounds around iteration 23 we then have to use the next 40 steps to be able to find our way back into the desired solution space.

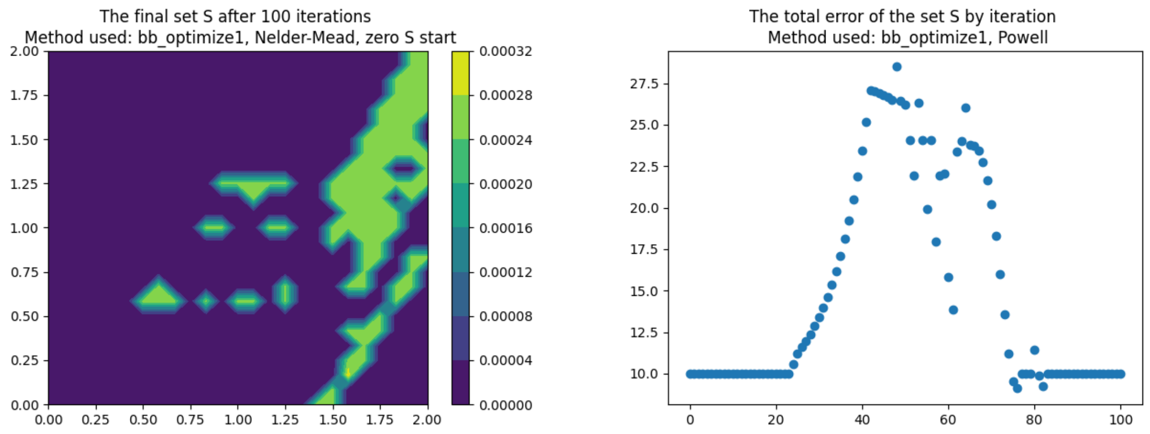

Figure 2: This shows the output of *bbopt*1 for 100 iterations with Nelder-Mead method, using the initial choice for S to be all 0's

Now one might say that the overall set from the zero start looks better, as it seems to retain the history of the previous iterations of gamma better. So to test this we ran it for 1000 iterations. From Figure 3 we can see that after the first 80 iterations it appears that gamma would get stuck at the edge and it fails to make much appreciable progress. However, it can be noted that the largest value of S is on the order of 0.2 as opposed to after 100 iterations that the largest value of S is on the order of 0.0003. Thus there was an increase in the largest value by a 1000 times. It would be interesting to see how it compares to 10,000 iterations, but this was omitted for lengthy runtime.

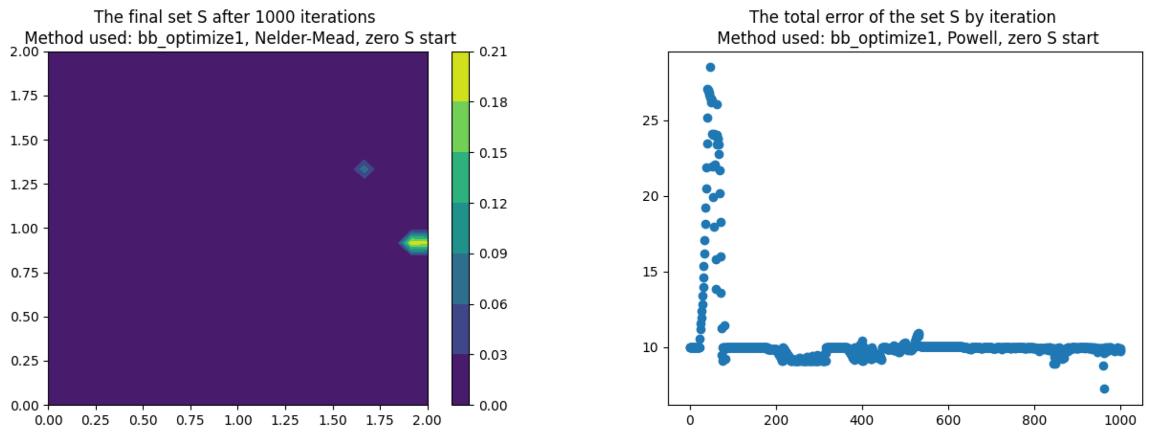

Figure 3: This shows the output of *bbopt*2 for 1000 iterations with Nelder-Mead method, using the initial choice for S to be all 0's

Now we proceed onto *bbopt*2, where we try to use a damping factor to make it where S cannot change quite so radically per step. What this does is that when it updates the set S, it takes the maximum of the new set and 0.95 times the value of the old set. The reasoning is that it should after each update it will still have some memory of the previous steps. The disadvantage to this is that the interior part of the optimization can still optimize while forgetting the old S, as we apply the damping after the optimization. We first applied this method with Nelder-Mead and curiously find that the damping has no change on the output, as seen in Figures 1 and 4.

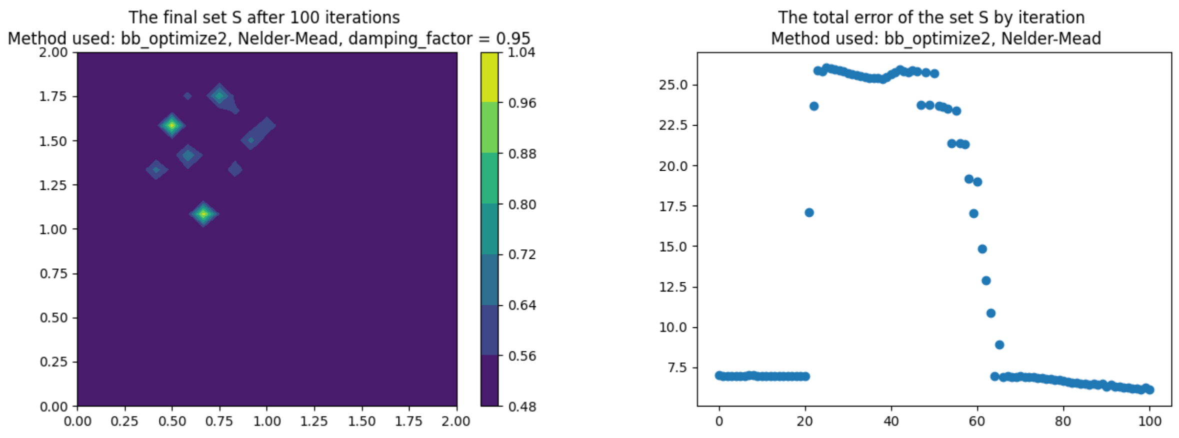

Figure 4: This shows the output of *bbopt*2 for 100 iterations with Nelder-Mead method, using a damping factor of 0.95

What we can begin to see is that in Figures 1, 4, 5 the Nelder-Mead algorithm behavior proceeds almost identically. As such it seems that Nelder-Mead is simply not a good choice for this problem without many more iterations. Surprisingly the damping and the change of where the damping was performed had no apparent change of the output.

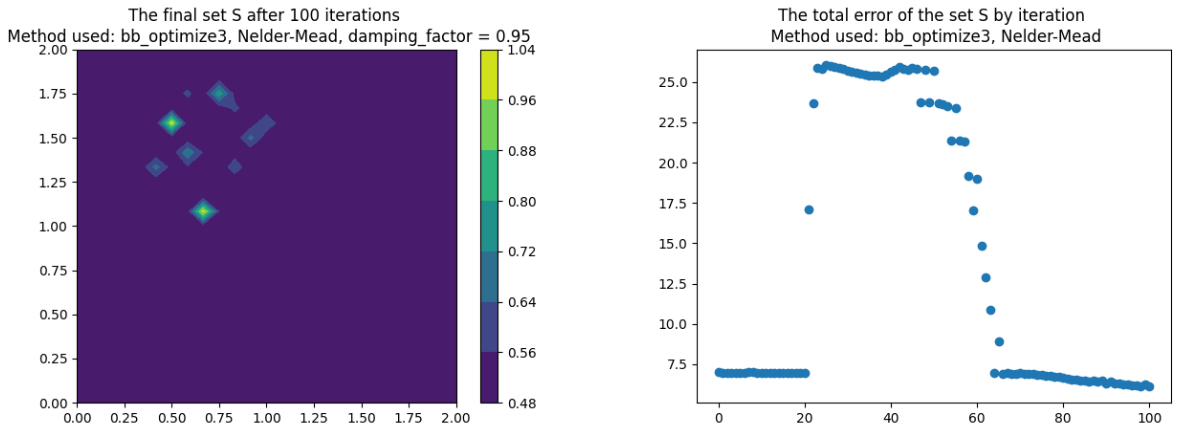

Figure 5: This shows the output of *bbopt*3 for 100 iterations with Nelder-Mead method, using a damping factor of 0.95

One last idea we can try with Nelder-Mead is to see if performance would improve if we let it run for more iterations. As we can see there is a slow progression on the error over time with a distinct improvement in the last 100 iterations. In addition there are several properties we desired of S that this process has yielded. First off, S doesn't look just like a single curve, namely it has an actual thickness to it. This is necessary as S is supposed to contain all curves, not just the current one. Secondly, many of the values of S are at 1, which is something else we want as we want the values of S to be in $\{0, 1\}$. Unfortunately, Nelder-Mead seems to have a hard time decreasing the values of points of S, and seems to have a bias towards increase. This can also be seen in the set S, where it tries to increase the x and y coordinates of γ where it can. I think perhaps with more iterations it might start to decrease the values of the non-desirable points of S, but this would be for future studies.

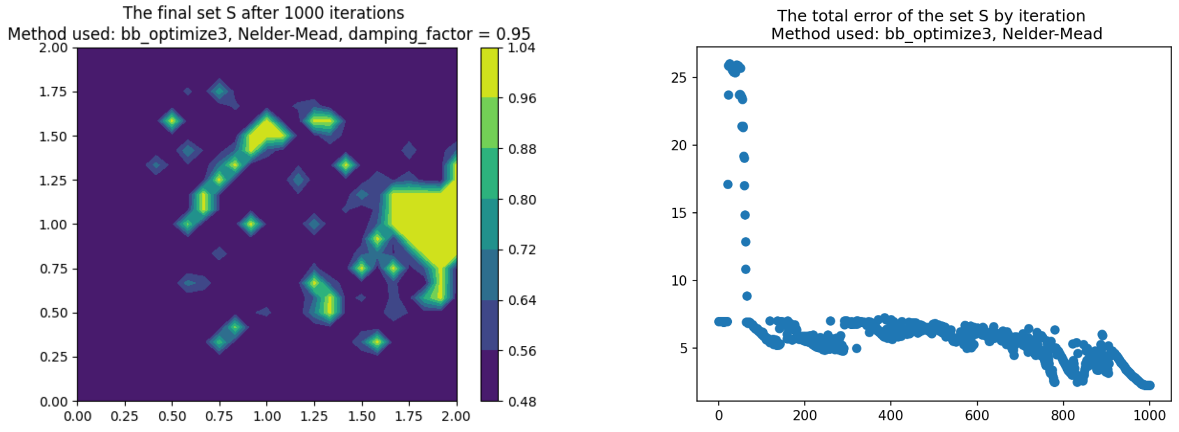

Figure 6: This shows the output of *bbopt*2 for 100 iterations with Nelder-Mead method, using a damping factor of 0.95

3 Powell Method

Now we consider the Powell method of minimization. In principle this should work better about finding local saddles as place as Nelder-Mead as it actually finds the minima along our basis directions, which allows for greater movement. Secondly it works well for finding local minima or saddle like points. One of the main downsides is that the computation time is much longer, and for my machine was 2 hours for the Powell method vs 1 minute for the Nelder-Mead.

Now as we expected, since the Powell method allowed for larger steps each iteration, we have a much jumpier error per each iteration. However, we can also notice that unlike in the Nelder-Mead method, very quickly, almost all of the errors were around 0, whereas the Nelder-Mead method failed to ever get below 7.5 in the first 100 iterations.

A second major benefit to the Powell method is that it created a more contiguous set as apposed to Nelder-Mead, however, due to the ability to make such large jumps, it just optimized S for the current γ. Now we can consider the similar cases as before, but without the zero start, as the increased movement makes the initial conditions change irrelevant.

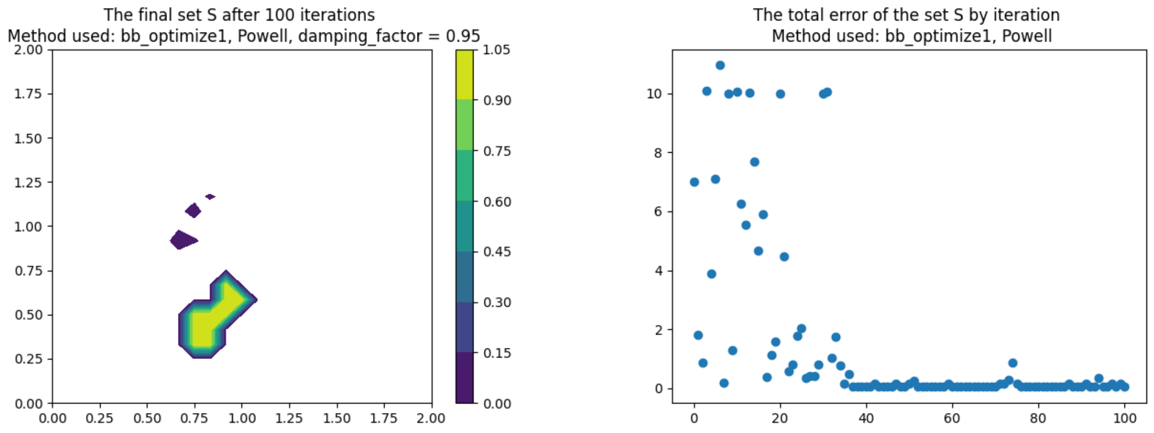

Figure 7: This shows the output of *bbopt*1 for 100 iterations with Powell method

Since as we can see in the first case, we have that S is contiguous, however, we want to have two things. First we want the error to be less jittery and secondly we want S to optimize for more than just the current gamma. As we can see in figure 8, damping outside of the optimizer accomplished the first goal. Namely that the error proceeds in a much more continuous and exponentially decreasing manner. The problem is that this still is optimizing only for one gamma. This is due to when we damp, we damp outside of the optimizer, so we don't take the history into account. To fix this we look at *bbopt*3.

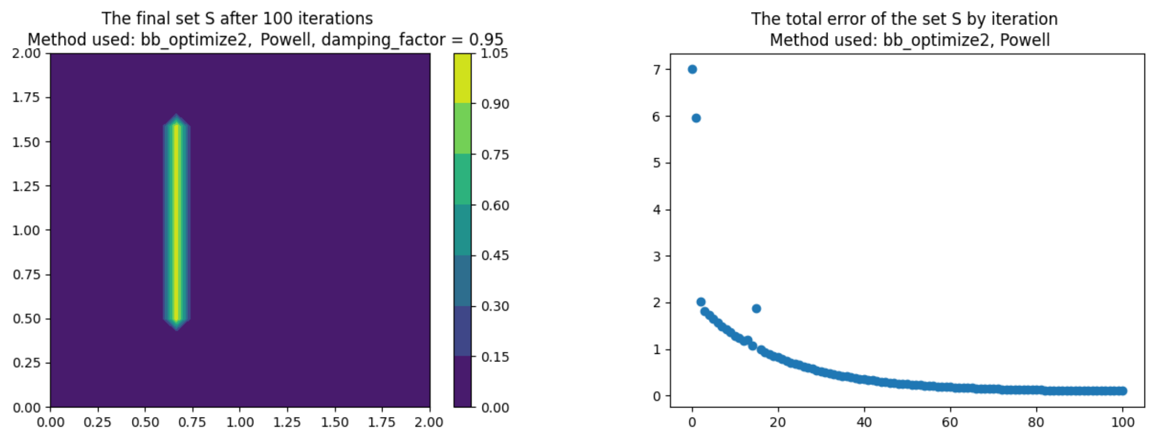

Figure 8: This shows the output of *bbopt*2 for 100 iterations with Powell method

In *bbopt*3 we use the damping in the minimizer with hopes to create a set S that will try to optimize for more than just the current gamma. This seems to be the case, as the final S set is the largest of all of the S sets we have compared so far. In fact it seems as if this set S would be a valid set for what we are considering. However, we do have a curiously large oscillating error that we did not have with the previous case that we damped outside.

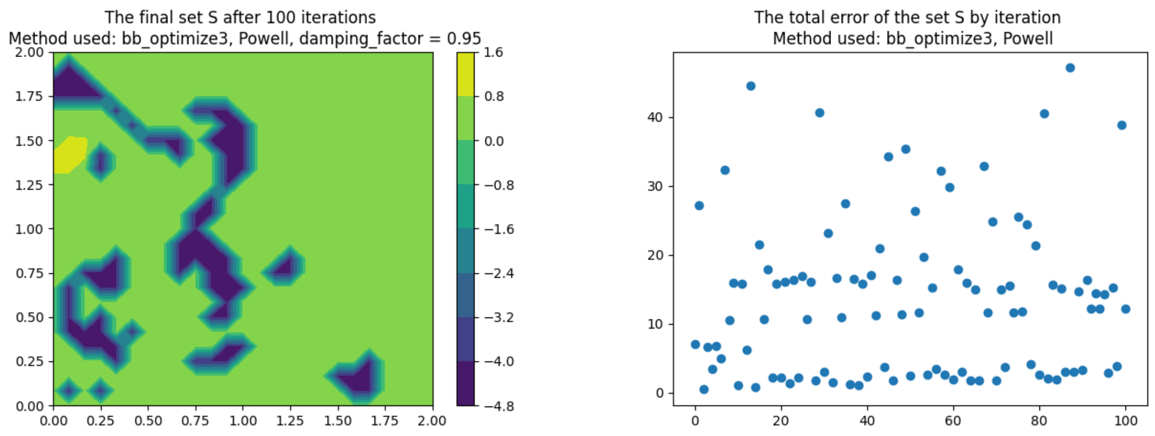

Figure 9: This shows the output of *bbopt*3 for 100 iterations with Powell method