Report: Shape-constrained Adversarial Robustness

Kellin Pelrine
Department of Economics
Yale University
New Haven, CT 06520
Kellin.Pelrine@Yale.edu

Abstract

Robustness is important in many areas of economics, since models and assumptions are typically only approximations of reality. This paper explores an algorithm for testing model robustness. It generates and optimizes perturbations in a way that accommodates size and shape constraints, without requiring any information about the model beyond a zeroth order oracle. The algorithm is tested experimentally on an RBC model. The results indicate the algorithm can find very effective perturbations and that this RBC model is not robust.

1 Introduction

Economic models are typically based on many assumptions: there's a particular set of agents, with specific objective functions, who value a given set of goods, which are produced in a particular way... These assumptions are necessary, since we cannot create a model that replicates the real world perfectly. But when working with an imperfect model, it is critical to understand how the assumptions and flaws can impact the results of the model. One desirable property is that a model be robust, in the sense that if our modeling assumptions are close to being met in the real world, then we would like our conclusions from the model to match or closely approximate the real world.

A standard way to evaluate robustness is in a sup-norm sense. This has the key property of being agnostic about the model and data generating process, thereby avoiding imposing extra assumptions or modeling decisions that also might not be robust. It is also a natural choice when an error can be very costly - for instance, when implementing the wrong economic policy could cause or prolong a recession.

Formally, given a model $T(X)$ mapping input functions X (e.g. utility functions, transition dynamics, etc.) to output Y (e.g. value/policy functions, correlations, etc.), one might like to solve $\sup_{X \in \Theta(X_0)} L(T(X), Y_0)$, where $T(X_0)$ is the original input (e.g. utility function = Cobb-Douglas or parameter $\alpha = 0.7$) and $\Theta(X_0)$ is a class of allowed perturbations (e.g. the intersection of a ball and cone in function space, where the ball represents "small" perturbations and the cone maintains a shape constraint such as "utility increasing in consumption"). We would like the output Y to be robust, and the loss function L defines how far we are from some baseline output Y_0 (e.g. known real-world data or the original output of the model).

This presents a number of challenges. First, we need to pin down the parameters of the problem. Although we try to be model-agnostic with the $\sup$, clearly not all choices of perturbations will make sense. For example, we may not want to allow perturbations which make a utility function decreasing, because even if the perturbation is small in some standard function space distance, it might represent a massive shift in the real world - a consumer who likes to consume more vs. one that wants to consume nothing. Second, even once the problem is reasonably defined, it may not be obvious how to solve. Often the model $T(X)$ will not have an analytical solution, so there is little hope of an analytical solution to the robustness problem, which adds another layer of optimization.

Preprint. Work in progress.

Further, the problem may be high-dimensional (especially in the case of a functional perturbation that cannot be expressed with a couple parameters) and/or highly non-convex, and the model may be computationally expensive to evaluate, so standard optimization routines may not be viable.

This problem is closely related to adversarial robustness in machine learning, which is an active research topic. There are various ideas and methods one can consider borrowing to facilitate the above computation. Many of these methods though require gradients [3, 4], either from the model itself or from a similar model (for transfer learning). These are not easily available here due to the dynamic optimization structure of macro models.

In this project I create a genetic algorithm inspired by GenAttack of Alzantot et al. [1]. This approach requires only a zeroth order oracle for the model, i.e. it is gradient-free and does not rely on any special structure. It is also highly adaptable to different perturbation specifications, because constraints are enforced in a generative way in steps that are independent of the algorithm's overall structure. Finally, it is highly parallelizable. The most expensive parts of the computation (evaluating the model, and possibly enforcing the perturbation constraints) are repeated in an entirely independent way, and the algorithm benefits from more repetitions.

I test this algorithm using an RBC model with original code from a homework by Fabrizio Zilibotti. I first study a simple one-dimensional case - perturbing the discount rate β, with correlation between output and consumption the quantity of interest - which can be solved with standard optimization routines. This shows my algorithm is indeed solving the optimization problem, and highlights some potential strengths (fast convergence) and weaknesses (tuning parameters can have a big impact in avoiding local extrema). I then examine monotonicity-constrained perturbations of the utility function. Here the algorithm finds small perturbations which produce a massive shift in output-to-consumption correlation, even changing the sign of the correlation in some tests. These results suggest this algorithm can indeed be effective at testing robustness. If they are not due to the numerical implementation of the RBC model, the results also suggest this RBC model may have extremely poor robustness properties - even an MSE of 0.001 in the utility function could produce unrealistic results.

The rest of this report is as follows: first, I discuss the overall structure of the genetic algorithm. Next, I show how to specialize it to the β perturbation case and the experimental results. Then I discuss the (shape-constrained) utility perturbation case and those results. I conclude the main part of the report with an overall summary and questions for future study.

I also include two appendices. The first discusses a technique I previously had in mind and presented for enforcing shape constraints, a Sobolev space projection algorithm. I briefly explain the idea and show experimentally why it did not work in this context. The second appendix explains the contents of the code files and how to reproduce the experimental results.

2 Algorithm

Algorithm 1 ShapeAttack

Input: X_0, *Initialize*$(\cdot)$, *Constrain*$(\cdot)$, *Fitness*$(\cdot)$, *Crossover*$(\cdot)$, *Mutate*$(\cdot)$, ρ, P, G
for *p = 1,...,P* in population **do**
    $X_1^p =$ *Initialize*$(X_0;$ *Constrain*$(\cdot))$
for *g = 1,...,G* generations **do**
    for *p = 1,...,P* in population **do**
        $F_g^p =$ *Fitness*(X_g^p)
    $X_{g+1}^1 =$ argmax *Fitness*(X_g)
    $probs =$ *Normalize*(F_g)
    for $p = 2, ..., P$ in population **do**
        Sample two parents from *probs*
        $child =$ *Crossover*$(parent_1, parent_2)$
        if Uniform$[0, 1] < \rho$ **then**
            Mutate$(child)$
        $X_{g+1}^p =$ *Constrain*$(child)$

The structure of the algorithm is given in Algorithm 1. This closely follows GenAttack [1], from which I borrow some of the notation, and more generally follows the structure of a standard genetic algorithm. The key point here is choosing the functional inputs.

The *Constrain* function needs to be constructed to enforce the desired constraints (size, shape, etc.). Using that function and the original model input X_0, the algorithm is initialized with enough perturbations to create diversity in the population (lines 2-3).

Next, the algorithm begins iterating through generations (line 4). For every member of the population one computes the fitness, which measures how effective the perturbation is (lines 5-6). The intuitive choice is the loss function of the problem from the introduction - but that is not saying much, since one also has to choose that loss function. Fortunately, standard loss functions should often be appropriate. In the experiments discussed in this report I use the absolute value.

The "elite" member of the population, the one with the highest fitness, is immediately added to the next generation (line 7). This guarantees that the fitness never goes down, at least if the model output is deterministic.

To construct the rest of the next generation, the algorithm first normalizes the fitness scores into a probability distribution (line 8). Then for each undetermined member of the next generation (line 9) it samples two parents from that distribution (independently; line 10). Their traits are somehow combined by the *Crossover* function (line 11). This function is again the choice of the user. In my experiments I use a simple average weighted by the fitness of the parents.

With probability ρ the child then mutates (lines 12-13). This probability and the mutation itself should not be so large as to completely randomize the next generation (we don't want to just do Monte Carlo) but should be large enough to escape local minima in a reasonable amount of time. Finally, we apply the constraint to the child before adding it to the next generation (line 14).

As noted in the introduction, this algorithm can be parallelized in a trivial way by running the fitness computation (lines 5-6) as a parallel loop. This is likely by far the most expensive part of the computation, so the benefits of parallelization are substantial. If some of the crossover, mutate, or constrain steps are also costly, lines 9-14 can also be parallelized, again by just running loop iterations in parallel. This does not depend at all on the model, so it can be effective whether or not the model is highly parallelized itself and without potentially difficult modifications to the model code.

3 Experiments

The model I use in these experiments is a basic RBC model:

$$\begin{aligned}
& \max E_0 \sum_{t=0}^{\infty} \beta^t U(c_t, n_t) \\
\text{s.t.} \quad & c_t + x_t = zF(k_t, n_t) \\
& k_{t+1} = (1-\delta)k_t + x_t \\
\text{with} \quad & U(c,n) = \log c - bn^{\epsilon}/\epsilon \\
& F(k,n) = k^{\alpha} n^{1-\alpha} \\
& \log z_{t+1} = \rho \log z_t + N(0, \sigma^2)
\end{aligned}$$

The code is a slight modification of code provided by Professor Fabrizio Zilibotti for a homework assignment. I take the output quantity of interest to be the correlation between output and consumption. The original code gave a somewhat unstable result, due to randomness of the generated sequence of shocks. I increase the number of iterations in the simulation part of the model from 100 to 1000 to help stabilize the output. An alternative would be to generate the same shocks every time, but this way shows that a small amount of randomness is not a problem for this algorithm, as well as automatically ensuring that the results here are robust to different draws of shocks.

The model also seems to produce NaN correlations occasionally. This is likely due to a rare set of shocks which push something (capital, I suspect) out of bounds. As a brute-force fix, I discard all NaN results from the simulation. This does not appear to change the correlation compared to

instances where no NANs were generated. Moreover, if anything this gives the initial model a head start in robustness, since some extreme shocks are discarded.

Finally, I convert the whole model into a function with all the parameters as inputs. In the "Perturbing U" case discussed below, this includes a discretized version of the utility function. The initial parameters are all taken from the values provided and solved for in the homework (solutions by TA John Finlay).

The original parameters produce a correlation between output and consumption of .75. As loss/fitness function I use the absolute value of the difference between this and the correlation from the perturbed model. For the crossover, as noted above, I take the average of the parents' attributes (entrywise in the "Perturbing U" case), weighted by the fitness probabilities.

3.1 Perturbing β

The original value of β is 0.964. I begin by searching between over β's between 0.5 and 0.99. This is an extreme range intended for testing the algorithm, not as an actual check on robustness of the model. I first computed the maximal fitness using a standard optimization function fminbnd (Brent's method). This gives β slightly above 0.65, which achieves fitness about .33. Note these numbers are approximate due to the randomness in the model output; a very loose empirical upper bound on the error is that it does not exceed variation in the hundredths place (an more thorough upper bound could be computed but is not meaningful to these results).

I begin by initializing the algorithm with a population size of 12 on an equispaced grid. This is viable in this simple one-dimensional case, but not in high dimensions. I set the mutation probability to 0.5, and mutate by adding a uniformly random amount between -.05 and .05. Results from 10 generations are shown in Figure 1.

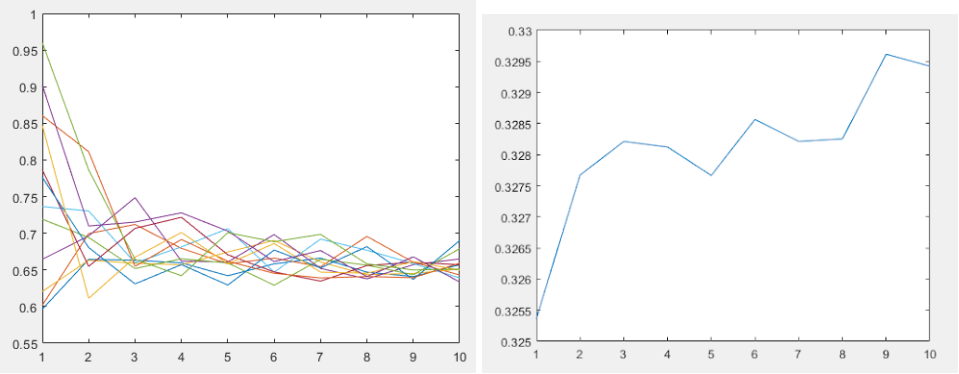

Figure 1: Test 1: grid start. Left: population. Right: fitness.

As can be seen, the algorithm converges to the correct solution, and quite quickly too. However, it is practically starting on top of the solution here. Next I tested with an equispaced start between .8 and .95 (population size = 8 here), shown in Figure 2.

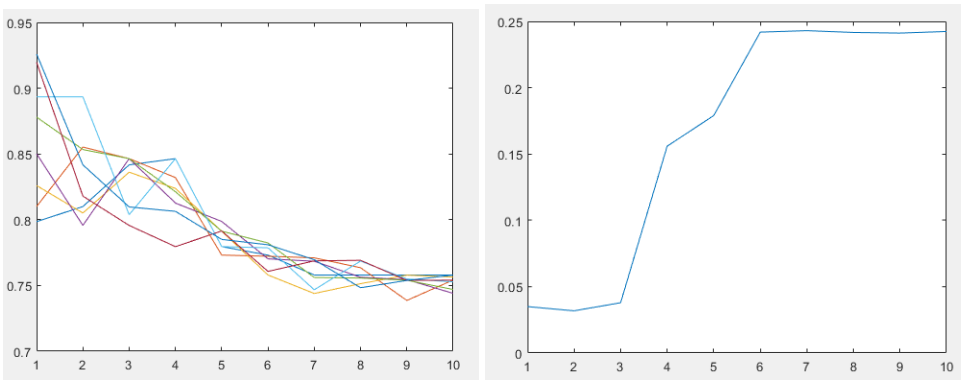

Figure 2: Test 2: start in wrong area. Left: population. Right: fitness.

This again converges, but not to the right place. In fact, an error in the code meant all the mutations were downwards, but even that was not sufficient to get to the true minimum. 40 generations (not pictured here) were also insufficient to escape this local minimum. The issue is that the geometry of the problem looks like Figure 3 (negative fitness here - minimization instead of maximization).

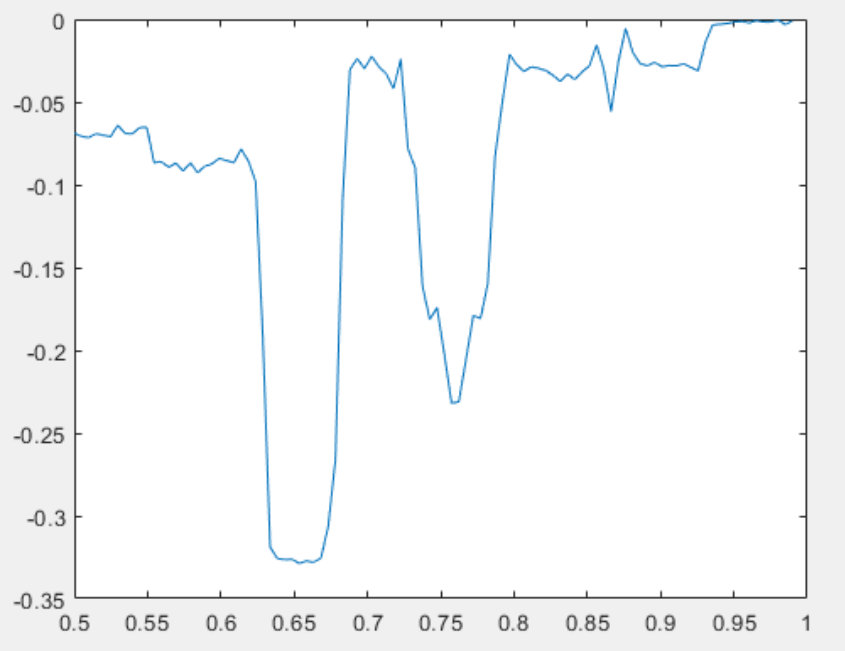

Figure 3: Problem geometry: fitness as function of β. Negative here (minimization).

This picture was made by directly evaluating the model at 100 evenly spaced β's. The small jaggedness is due to the randomness in the model, but there is a large local minimum between .75 and .8, exactly where the algorithm is getting stuck in Figure 2. Although eventually the algorithm will escape (it is straightforward to show that it explores every point with probability 1 given infinite generations), the above inputs, especially the mutation scheme, do not do so efficiently.

This can be fixed with a different mutation. In figure 4 I show a random normal mutation with variance .02 (population size 12). This is sufficient and even overkill, producing some wild fluctuations in the population even after convergence of the elite member (and thus the algorithm overall). I later tested using the uniform random mutation again, but larger (.3 instead of .05) and that also worked (not shown here, but included in the accompanying code).

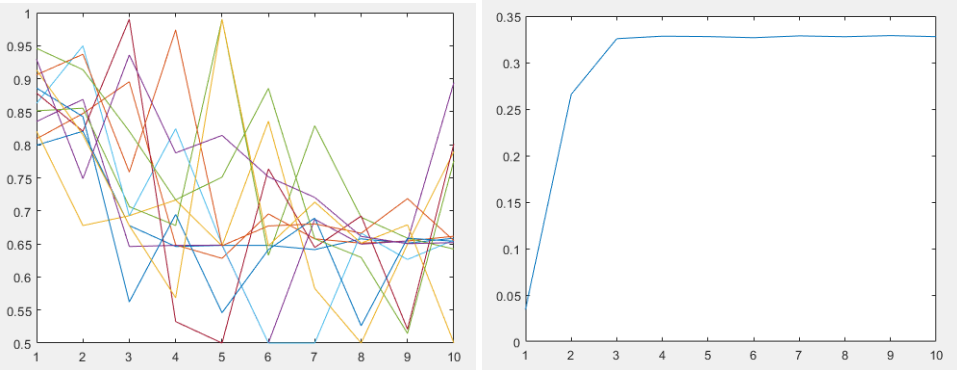

Figure 4: Test 3: random normal mutation. Left: population. Right: fitness.

These tests indicate that the algorithm is indeed solving the optimization problems, but careful tuning may be needed to achieve the best results and avoid local minima.

In this case the constraints were not binding in the sense that the global optimum was interior. However, one can see members of the population hitting the constraints in Figure 4. The constraint was enforced in the simplest way possible, simply setting any member of the population outside the bounds back to the edge. This easily gives conventional perturbations which are bounded in absolute value or some other simple (that is, easily calculated) norm. Considerably more effort is required for shape constraints, which we turn to next.

3.2 Perturbing U

In this section I perturb the consumption part of the utility function. The most important shape constraint here is monotonicity. If we could perturb utility to be decreasing in consumption, we can trivially destroy any conclusions from the model, since the agent will always consume 0, but this is clearly not meaningful.

Beyond monotonicity, in this model a constraint on the level of utility as function of consumption is necessary, because there is also the additive penalty for labor. If we change the level drastically, it will completely dominate or be dominated by the labor decision.

In order to satisfy these requirements, I first define a parameter "gridsize" and discretize the model's original utility function. From this I calculate the increments in the function between grid points and save these in the variable "increase." I then implement the following algorithm:

Algorithm 2 MonotoneMutate

Input: $X_0,\ \alpha,\ \delta,\ gridsize,\ increase,\ P$
for *p = 1,...,P* in population **do**
  for $i = 1, ...gridsize - 1$ **do**
    $candidate = \alpha * \text{Uniform}[-1, 1]$
    $maxdecrease = increase_i$
    if $\alpha < maxdecrease$ **then**
      $perturbation_i = \alpha * candidate$
    else
      $perturbation_i = maxdecrease * candidate$
  $norm = \sqrt{\frac{1}{gridsize-1} \sum_{j=1,...,gridsize-1} (perturbation_j / increase_j)^2}$
  if $Norm > \delta$ **then**
    $perturbation = \delta * perturbation / norm$
  $X^p_{1,1} = X_{0,1}$
  for $i = 2, ..., gridsize$ **do**
    $X^p_{1,i} = X^p_{1,i-1} + increase_{i-1} + perturbation_{i-1}$

The key to this algorithm is to work directly on the increments of the function. For each increment (line 3) the algorithm creates a candidate mutation. We compare it with the original increment (lines 5-6) and guarantee it is smaller than the original increment, and therefore will not break monotonicity (lines 7 and 9). Note there is something weird here - more explanation below.

The algorithm then calculates a special norm (line 10), a two-norm that is weighted by the original increase. This is designed to keep the perturbation small compared to the original increment. If the norm is too big, the perturbation is shrunk by renormalizing it (lines 11 and 12).

This produces a small monotonicity-constrained perturbation. To apply it, we begin with the same initial point (line 13), then reconstruct the function point by point, adding the original increase and the perturbation.

A careful reader may note that the stepsize α is applied once in line 4, then another factor is applied in line 7, or else the other scaling factor *maxdecrease* is applied in line 9. This reader is more careful than I was, because I intended to only apply the scaling once. There is no longer enough time before this report is due to test the intended version (sorry about that), but actually this only shrinks the perturbation. Therefore it does not negatively impact the results, except that it shrinks the perturbation especially strongly as the gridsize increases (because *maxdecrease* gets smaller).

This algorithm covers both initialization and mutation. In the initialization case, X_0 is the original utility function, whereas in the mutation case it is the original child function. In the latter case, one separately uses the child's differences as *increase* in the perturbation generation part, while using the differences of the original utility function when enforcing the norm.

Note that the norm here has nothing to do with monotonicity but is critical for enforcing level constraints. Further testing is needed, but my preliminary experience indicated the special weighting is far superior to a more standard norm in producing good perturbations.

The following experiments were done with population size 20 over 300 generations (using Grace - this takes a bit under 6 hours with 20 cores, not too crazy). As noted before, the original utility function is log. I set $\delta = 1$ and $\alpha = .2$ in the initialization, then reduce α to .1 in the main algorithm. The mutation probability is .2.

In the first test I discretize the utility function with 50 equally spaced points (0.00005 to 1.0). These grid points are linearly interpolated when the model calls the utility function. The initialization produces perturbations like those shown in Figure 5.

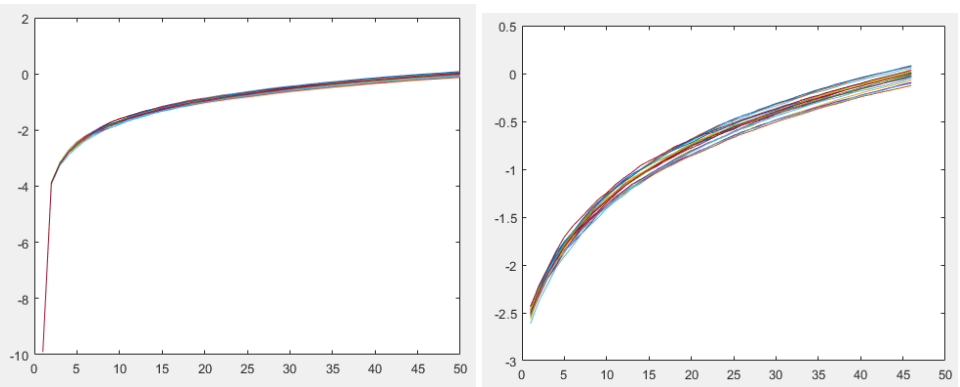

Figure 5: Test 1: 50 grid points. Initialization. Left: full. Right: 4 leftmost points omitted from graph.

As can be seen, the shape looks reasonable. The original (discretized) function is included in these pictures, but is indistinguishable, illustrating that the fit is quite close. On the right I omit the first 4 points of the graph, where there is the most variation in the function, to better show the perturbations. They do not deviate from the original function radically but have high diversity, exactly like we want for the initialization.

The next figure shows an example perturbation after running the algorithm which achieves the maximum fitness, and the graph of the fitness. The picture is large to make the perturbation more visible. It is shown in red, while the original function is shown in blue.

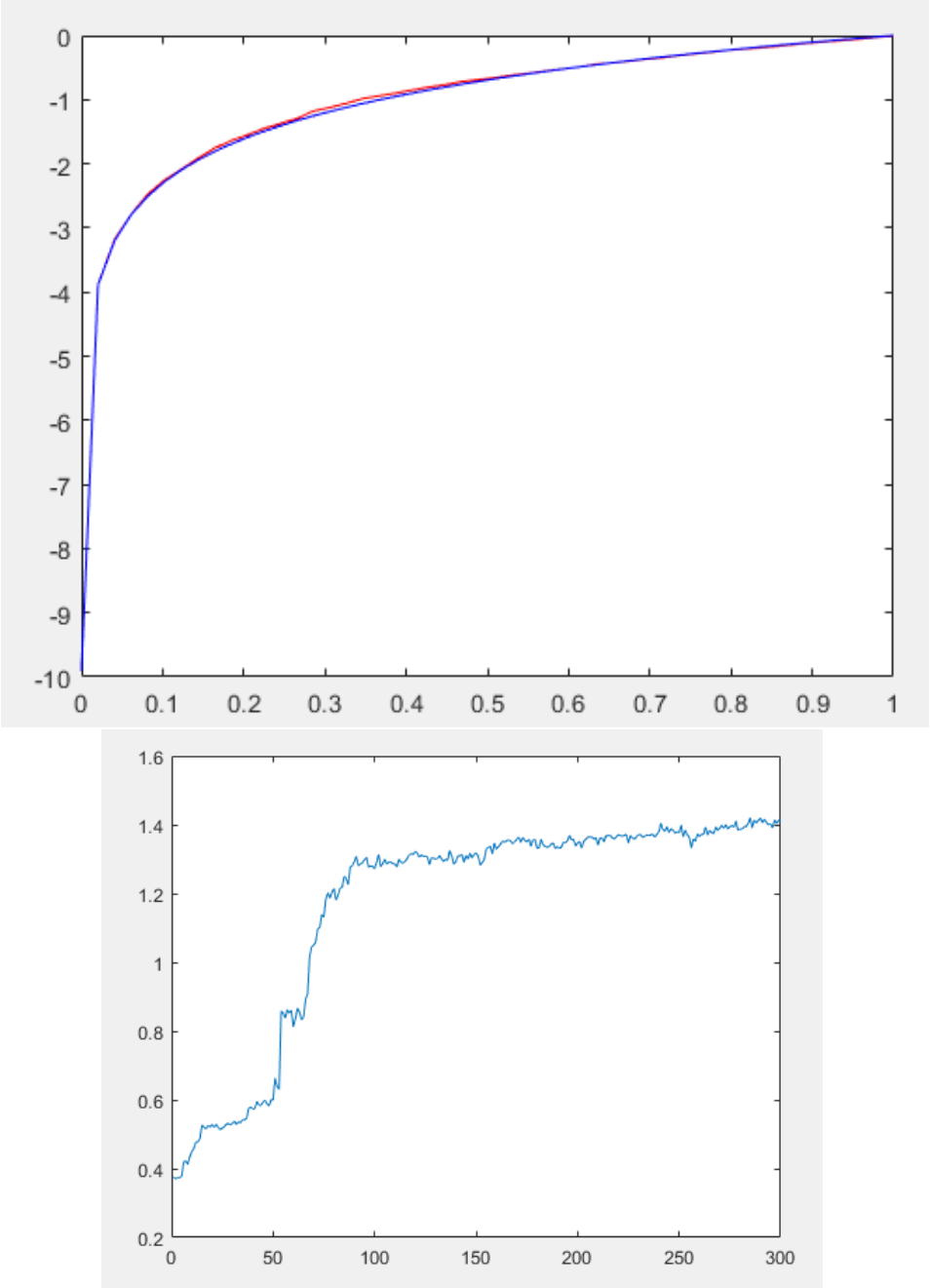

Figure 6: Test 1: 50 grid points. Above: example perturbation achieving fitness 1.42. Below: Fitness.

These fitness values are not scaled. The perturbation shown (which was selected arbitrarily from the final population) reverses the correlation between output and consumption entirely. With the perturbed utility function, the model says that output and consumption are negatively correlated, changing from the original value of +.75 with the blue function to -.67 with the red one.

The MSE between the true function and this perturbation is 0.00097. This suggests one may need to estimate the utility function with accuracy greater than that in order to rule out a result like this with this model.

In addition, the fitness in the first generation is just under 0.4. This is already a significant amount, and is generated without optimization, suggesting that perturbations which generate significant changes in the correlation are not just worst case but also occur with substantial probability.

Next, I tested increasing the grid size in the discretization of the utility function. As mentioned above, due to the unintentional structure of the algorithm, this has the side effect of reducing the size of the perturbations.

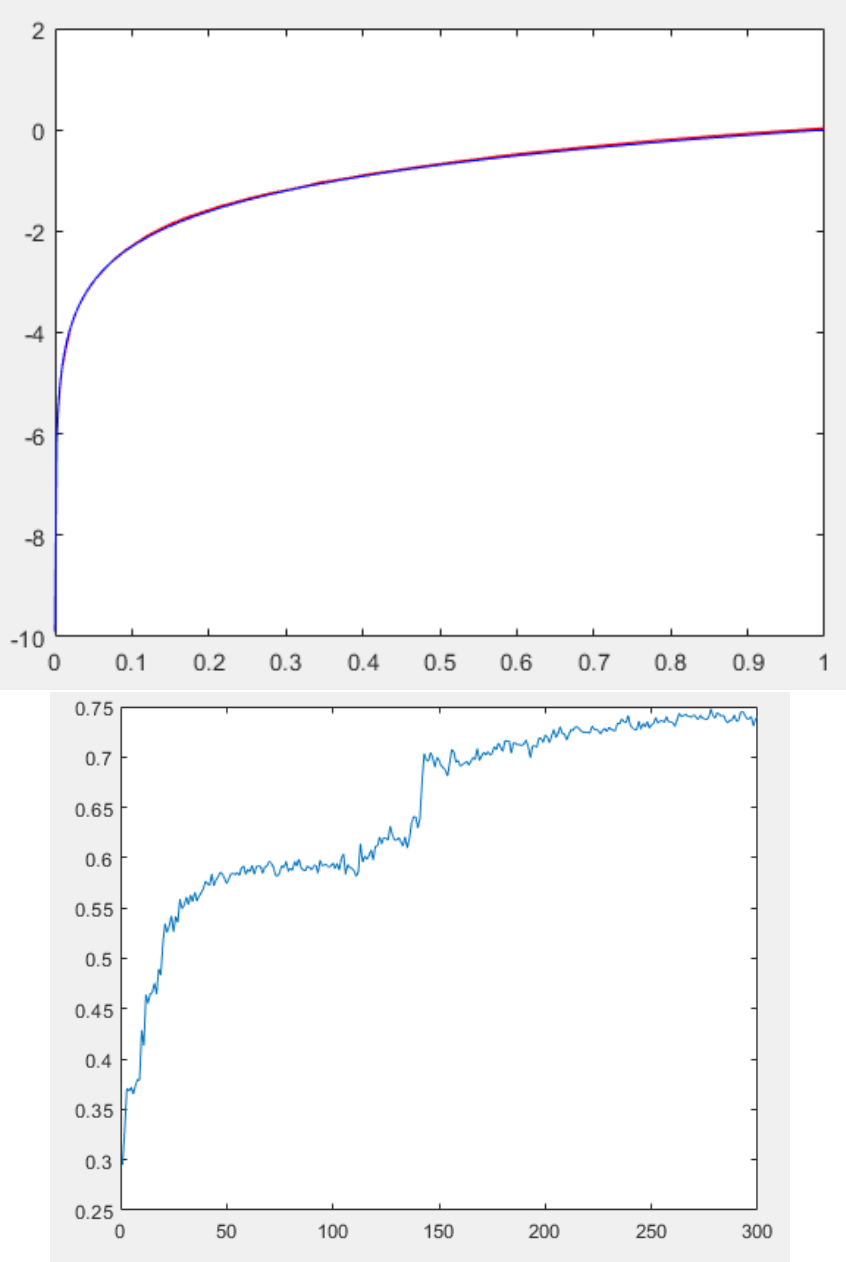

Figure 7: Test 2: 500 grid points. Above: example perturbation achieving fitness 0.74. Below: Fitness.

As before, there are two functions in the upper graph, the original in blue and the perturbed one in red. The perturbation is smaller, almost imperceptible. The change in the correlation is also smaller, but still massive, from .745 to near 0.

To verify that the difference here is indeed because the perturbation is smaller and not because the grid size is larger, I transfer the 50 grid point perturbation of Figure 6 to the 500 point grid by linear interpolation. This reveals a potential issue: the leftmost part of the perturbation diverges substantially from the 500 grid point true utility function, because it changes too quickly there for the low grid-size linear interpolation to keep up. This is shown in Figure 8.

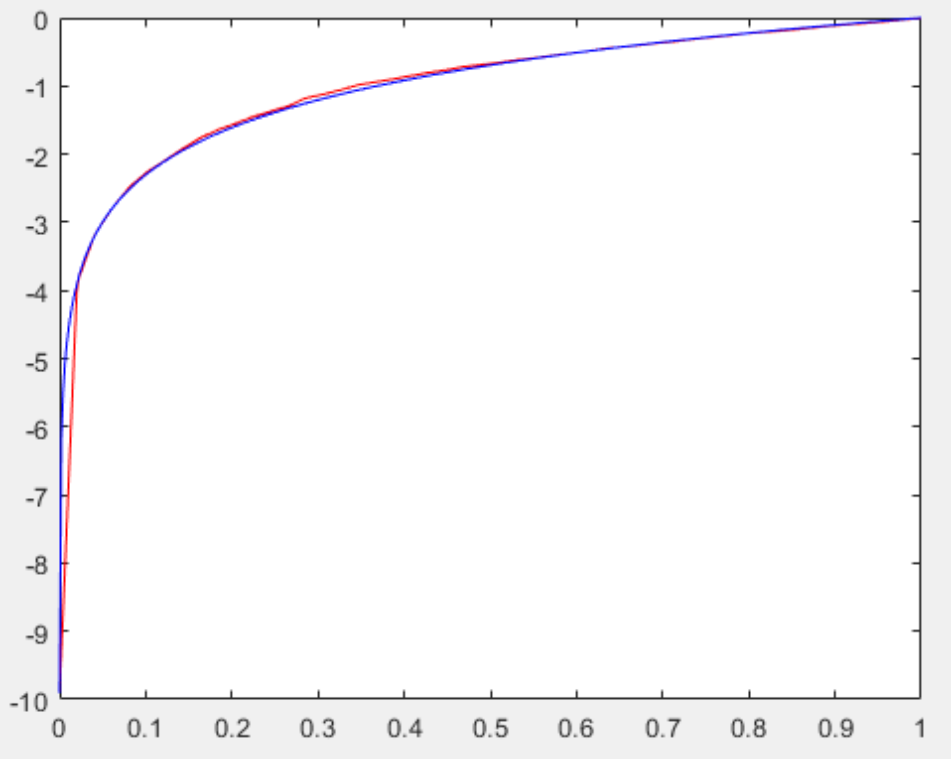

Figure 8: Test 3: 50 to 500 grid points. True discretization in blue, perturbation in red.

To fix this, I replace the first 50 points of the 50-to-500 perturbed function with the first 50 points from the 500 point perturbed function shown in figure 7. This gives the perturbation shown in Figure 9. The left side of the graph looks much more reasonable.

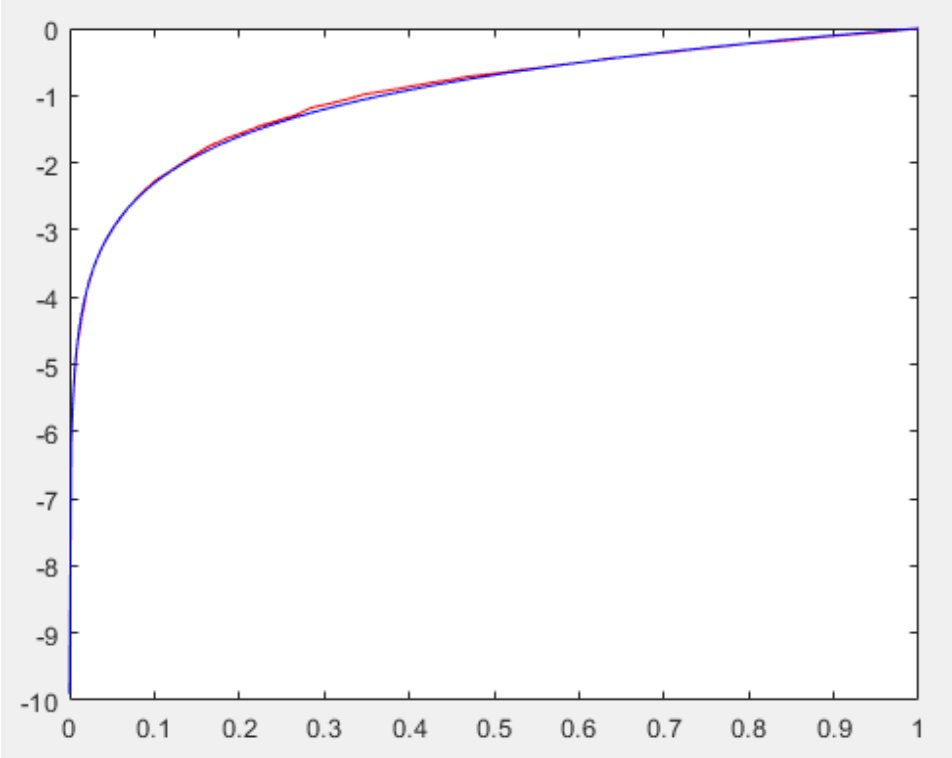

Figure 9: Test 3: 50 to 500 grid points. First 50 perturbation points from Figure 7 perturbation.

This produces fitness 1.42, matching the result with 50 points. So it is not an artifact of a small discretization. This also suggests that there may be some potential for transfer learning or some sort of (parallel?) tempering here. There may also be potential for "feature-preserving" crossover functions, i.e. ones which preserve entire parts of parent functions, which I thought were a bad idea in this context when I presented two weeks ago.

4 Conclusion

The aim of this project was to develop tools to better understand when and how economic models might fail. The tests so far are somewhat limited in that they only cover a single model and single output variable, but the results are strong and suggest the proposed algorithms can be effective.

They also suggest this RBC model has very poor robustness properties. These results may be qualified in two ways. First, there is insufficient testing to completely rule out that this outcome is some numerical artifact or outright error. This can only be resolved with further testing and review of the code. Second, one may object that although monotonicity has been enforced, and the perturbations are close to the original function, they are not concave (in general, and it's possible to see visually that the function in figure 8 is not concave, though in figure 7 it's hard to tell). This is a legitimate concern, but "perfect" concavity may be a very strong assumption. It is one thing to say consumers are risk-averse in general or that a fully risk averse consumer is a good approximation, but another to say they are *always* risk averse. The perturbation in Figure 7 would seem to require a very strong concavity assumption, close to perfect concavity, in order to exclude it.

Nonetheless, both these potential criticisms suggest topics for further research. Along the lines of the former, it would also be worth testing this algorithm on other models. Along the lines of the latter, it would also be interesting to extend this to multidimensional utility functions (that aren't additively separable) and to perturb the utility function and other parts of the model simultaneously. Finally, another obvious idea is to try correcting the unintentional shrinking of the perturbation in the MonotoneMutate algorithm.

This report is also notably lacking a literature review. To my knowledge, no one has examined shape-constrained perturbations in a computational way before, but a thorough review of the robustness literature is needed.

References

[1] M. Alzantot, Y. Sharma, S. Chakraborty, and M. B. Srivastava. Genattack: Practical black-box attacks with gradient-free optimization. 2018. URL `https://arxiv.org/pdf/1805.11090.pdf`.

[2] M. Delecroix, M. Simioni, and C. Thomas-Agnan. Functional estimation under shape constraints. 1996. URL `https://www.tandfonline.com/doi/pdf/10.1080/10485259608832664?needAccess=true`.

[3] N. Papernot, P. McDaniel, I. Goodfellow, S. Jha, B. Z. Celik, and A. Swami. Practical black-box attacks against machine learning. 2017. URL `https://arxiv.org/pdf/1602.02697.pdf`.

[4] F. Tramèr, A. Kurakin, N. Papernot, I. Goodfellow, D. Boneh, and P. McDaniel. Ensemble adversarial training: attacks and defenses. 2018. URL `https://arxiv.org/pdf/1705.07204.pdf`.

5 Appendix A: Sobolev space projection constaints

Monotone increasing functions form a closed, convex cone in Sobolev space. My original idea for enforcing this constraint was to take an unconstrained perturbation and project it to the cone. This strategy was used effectively in Delecroix et al. [2] to enforce monotonicity constraints for nonparametric (kernel) regression.

Their algorithm is based on the following. An increasing function has nonnegative derivative at every point. In a Hilbert space with continuous first order differentiation functionals, that is $D_x^1(u) := u'(x)$, there exist Riesz representers d_x^1 for which $\langle d_x^1, u\rangle = u'(x)$ for all points x and functions u (in the space). The cone of monotone functions is then $C = \{u(\cdot) \in H : \forall x,\ \langle d_x^2, u\rangle \geq 0\}$.

The projection onto this cone exists because it is closed and convex. Computing it requires solving a quadratic program with infinitely many constraints, i.e. verifying that the derivative is positive at every point, and is generally impossible. However, the cone can be discretized by taking finitely many points at which to enforce the contraint, yielding a tractable optimization problem.

The second order Sobolev space H^2 satisfies the continuity requirement. The Delecroix et al. [2] paper gives closed forms for the necessary Riesz representers, and the remaining details of the algorithm.

I used this method previously with a local linear regression and found it worked as described. However, when I applied it to generate monotone perturbations here, there were two issues. First, it assumes one has an analytic derivative for the function one wishes to constrain. This is obviously not available for a random perturbation, but perhaps not an insurmountable barrier; I attempted to use finite differences to get around it. But then I found that the results were just not monotone at all.

I suspect that when the original function is fairly smooth, with the derivative only changing signs a few times at most, this method works effectively with a small discretization of the cone. But when adding random noise to each point, the derivative changes signs wildly, and then a small discretization may be very insufficient. I did not test with a massive amount of discretization points, and there is also a tuning parameter that plays a role (Delecroix et al. [2] don't use the standard Sobolev norm; they weight the derivative part, which becomes a tuning parameter regulating how much one wants the projection to place importance on the original function itself vs. its derivative). But if the projection gets too expensive or unstable it will not be a very effective way to enforce the constraints anyways here.

The method I explain in the main section of my report gives guaranteed monotonic perturbations, runs quickly (no quadratic program needed), and seems to give nice results. I think modulo some tweaking (and as mentioned, fixing the error I made) that perturbation generation problem is solved, at least in the current context. But this and other projection methods may be worth revisiting for concavity and other constraints, or if other difficulties come up.

In the figure below I show some of the non-monotonic perturbations the Sobolev method generates (shown from the fifth point on, removing the leftmost section so it's clearer). It may be hard

to see in the figures (to produce this graph for further inspection, see the next appendix), as the non-monotonicity is not huge, but it is present somewhere in virtually every perturbation.

As an aside, this shows attempting to enforce a level constraint with a standard norm (the Sobolev norm in question), and it does not perform as well as the one used in the main report.

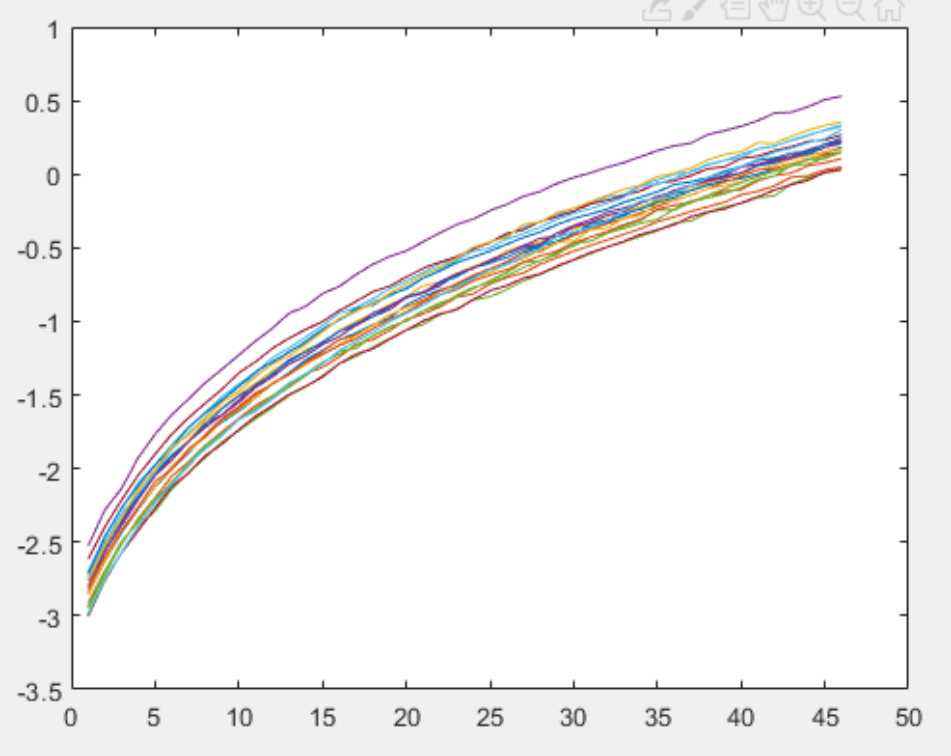

Figure 10: Sobolev projection perturbation generation.

6 Appendix B: notes on replication

The attached folder includes all the Matlab files needed to produce these results (as well as some which produce some results not covered here, and old saves of the code). Matlab was used to match the RBC code, as well as some code for Sobolev space projection I wrote before (which ended up unused here).

The RBC model is contained in the files "model.m" (most of the model), "u.m" (the utility function), and "markovchain.m" (discretization of the shocks). The files "model2.m" and "u2.m" accept and interpolate a utility function grid for the second set of experiments.

These are used by the files with "main" in the name to produce the experimental results. Those with "cluster" in the name are meant for use with Grace; submission scripts "submit_β.sh" and "submit_U.sh" are included. It is set for 20 CPUs; to run it on more or less the script must be modified but also the line "parpool(20)" at the top of the code (I don't think it will crash if one doesn't have 20 cores available, but it may multi-thread it in a weird way). "old_main_cluster_beta.m" produces results like those in the first set of experiments (perturbing β). This takes 15 minutes or so to run, 30 minutes job time is more than enough. "main_cluster.m" produces results like those in the second set of experiments (perturbing U). This one runs for 30 times more generations, so much more time is needed. In my experience it runs in a bit more than 5 hours but under 6. Both produce and save the maximum fitness per generation and the population itself (every generation) in a file "pop_and_fitness.mat," and the latter also includes checkpoints every 50 generations.

Some results are saved in the included copies of those files. The "[...] - uni + stepsize 015.m" is for betas perturbed with a uniform mutation of size 0.15. It gets stuck in the local minimum as described.

I don’t have any more output for the betas on hand (apologies, I should have saved one that doesn’t get stuck) but "old_main_cluster_beta.m" should produce it (there it’s uniform mutation of size 0.3). The other "pop_and_fitness" files - firstU, secondU, and thirdU are results for U perturbations with gridsizes 50, 100, and 500 respectively. The 50 and 500 ones are the results that produce the graphs shown in this report.

"u_perturbation_plots.m" produces pictures like the ones in Figure 5 (note: the base utility function won’t be included in the picture by default, only the perturbed ones; in Figure 5 I added it in manually). "old_main_sobolev.m" produces similar pictures for Sobolev projection constraints, which illustrate why they do not work here (not fully monotonic). This uses "projdiff.m." Neither of these files needs the cluster; they should run in seconds.

I’ve run these files on Matlab R2019a on my own computer and Matlab R2018b on the cluster with no versioning issues.