Assisted Parameter and Behavior Calibration in Agent-based Models with Distributed Optimization

Matteo D'Auria[1], Eric O. Scott[2], Rajdeep Singh Lather[2], Javier Hilty[2], and Sean Luke[2]

[1] Università degli Studi di Salerno, Salerno, Italy
`matdauria@unisa.it`
[2] George Mason University, Washington D.C., USA
{`escott8,rlather,jhilty2`}`@gmu.edu`, `sean@cs.gmu.edu`

Abstract. Agent-based modeling (ABM) has many applications in the social sciences, biology, computer science, and robotics. One of the most important and challenging phases in agent-based model development is the calibration of model parameters and agent behaviors. Unfortunately, for many models this step is done by hand in an ad-hoc manner or is ignored entirely, due to the complexity inherent in ABM dynamics. In this paper we present a general-purpose, automated optimization system to assist the model developer in the calibration of ABM parameters and agent behaviors. This system combines two popular tools: the MASON agent-based modeling toolkit and the ECJ evolutionary optimization library. Our system distributes the model calibration task over very many processors and provides a wide range of stochastic optimization algorithms well suited to the calibration needs of agent-based models.

Keywords: Agent-based Models · Model Calibration · Evolutionary Computation

1 Introduction

In an agent-based model, many *agents* (computational entities) interact to give rise to emergent macrophenomena. Agent-based models (ABMs) are widely used in computational biology, social sciences, and multiagent systems. An important step in developing an agent-based model is *calibration*, whereby the model's parameters are tuned to produce expected results. Agent-based models can be challenging to calibrate for several reasons. First, agents often have numerous and intricate interactions, producing complex and difficult to predict dynamics. Second, the agents themselves may be imbued with *behaviors* that need to be tuned: and thus the parameters in ABMs may not just be simple numbers but computational structures. Finally, ABMs are often large and slow, which reduces the number of trials one can perform in a given amount of time.

Despite its importance, ABM calibration is often done by hand using guesswork and manual tweaking, or the model is left uncalibrated because the model's complexity makes it too difficult for the modeler to perform the calibration! For example, in [6] approximately half of the surveyed models performed no calibration at all.

In this paper we consider the task of *automated agent-based model calibration*. We marry two tools popular in their respective fields: the MASON agent-based simulation toolkit [10], and the ECJ evolutionary optimization library [20]. MASON is an efficient ABM simulation tool which can be serialized and encapsulated in a single thread, making it a good choice for massively distributed model optimization, and ECJ has facilities critical to ABM optimization: it can perform distributed evaluation on potentially millions of machines, and it has a wide range of stochastic optimization facilities useful for agent-based modeling.

We will begin with an introduction to the ABM model calibration problem and discuss previous work in model calibration and optimization. We will next provide some background on ECJ and MASON, then present our approach to massively distributed ABM calibration, including examples that provide insight into the breadth of the approach.

2 Agent-Based Modeling and MASON

Agent-based models are often used to simulate large groups of interacting entities, such as flocks of birds, swarms of robots, warring nations, people flowing through airport checkpoints, and so on. In particular, an ABM describes the *interactions* among the agents, and the complex macrophenomena that arise as a result.

MASON and GeoMASON MASON is a popular, high-performance, open-source ABM library. MASON maintains a real-valued discrete event schedule that stores agents waiting to respond to time-based events, and one or more *fields*, that is, representations of spatial relationships between arbitrary objects (possibly including agents themselves). Basic provided fields include continuous spaces, various kinds of grids, and networks. MASON comes with extensive optional GUI visualization facilities for agents, objects, and fields in 2D and 3D.

GeoMASON augments MASON with Geospatial Information Systems (GIS) facilities in the form of vector and raster geospatial data, including spatially organized fields, visualization, and data manipulation utilities. GeoMASON can model agents that use earthbound objects and features such as networks of roads or rivers, vegetation, and topology, and is often used to study both social behavior and its response to natural processes such as rainfall and erosion.

3 Model Calibration and Evolutionary Optimization

Testing a model for correctness involves several steps. First, the model is *verified*, that is, it is debugged. Second, the model is *calibrated*, where its parameters are iteratively adjusted to minimize error between its output and some standard provided by the modeler. This standard can be many things, such as: the opinions of a domain expert observing the model, published benchmark values, or a sample of real-world data. Finally, the model is *validated* by comparing its results to a much more significant body of real-world data.

We focus here on model calibration. This is essentially an optimization task: the modeler repeatedly tries new settings of parameters until he finds ones that minimize error. Model parameters are of several kinds, only some of which are used in the calibration process. Consider the following four parameter types:

1. Parameters fixed to constants because their values are known beforehand.
2. Parameters fixed to constants because they are part of the canonical theory that the model developer is trying to demonstrate.
3. Parameters whose values are unknown, or can only be guessed at.
4. Parameters for which we wish the model to be *insensitive*. These are important but less common.

The calibration task is largely concerned with the third and fourth kinds, and particularly the third one: tuning parameters with unknown or unknowable values. Unfortunately, these parameters may exhibit significant nonlinearity, complex linkage with other parameters, and stochasticity. All this may demand many repeated tests of the model, but ABM models can take a long time to run. For this reason, automated calibration of agent-based models is desirable.

Historically model developers have resorted to linear and nonlinear gradient-based optimization approaches, even as simple as gradient descent. These techniques can fail with agent-based models for two reasons. First, such models may have large numbers of local optima. Second, these models may not yield a gradient, either because it is unknown or because the space is not metric: for example, a parameter might be a tree or an edge in a graph. Nominal categorical values (such as race or religion) may cause related problems.

The classical approach to optimization of data of this kind is to use a *stochastic* optimization procedure such as hill-climbing, simulated annealing, or an evolutionary computation method such as a genetic algorithm. The evolutionary computation family is particularly attractive because it is efficiently and massively distributed. This allows us to optimize a model by running many trials in parallel.

ECJ ECJ is an open-source stochastic optimization toolkit which emphasizes evolutionary computation techniques [20] and is one of the most popular tools in the evolutionary computation community. ECJ can run in a single process and can be distributed across a very large number of machines. ECJ has a many optimization features, such as customizable representations of candidate solutions (*individuals*), facilities for massively distributed model evaluation, and a wide range of optimization algorithms, some of which we will highlight later.

4 Related Work

Because of their complexity, many ABMs are calibrated by simply fixing the parameters in advance using known real-world parameters. But whenever the quality of a model's overall behavior can be assessed, such as using the opinion of a domain expert (as in [7]), or via comparison to real-world output, we might instead optimize one or more model parameters to fit it. When a model has (≤ 3) free parameters, researchers often optimize them with ad-hoc manual tuning, or by reviewing the outcome of an exhaustive parameter sweep (grid search) [1].

Automated model calibration will require an optimization algorithm. Evolutionary algorithms and related stochastic optimization algorithms are well-established approaches to calibrating free parameters for many kinds of models, and have been used to tune models of neuron behavior [19, 23], agriculture [12], and textile folding [13], among many others. In the ABM community evolutionary algorithms have been used to calibrate a number of models [4, ch. 10]. Many are based on an ad-hoc variation of the genetic algorithm [2, 16, 15]. ABMs often also have multiple conflicting objectives that need to be optimized simultaneously. Some authors have applied multi-objective evolutionary algorithms [18, 14], but have rarely used state-of-the-art methods (such as NSGA-III or MOEA/D). These techniques make it increasingly possible to tackle complex and high-dimensional problems, but take effort to implement properly.

Only a handful of software tools are available that allow researchers to apply optimization techniques to ABMs without needing to implement their own calibration framework from scratch. The *BehaviorSearch* module in NetLogo supports simple multithreaded optimization of model parameters via a few classic metaheuristics — hill climbing, simulated annealing, and a genetic algorithm [22], and a few basic solution representations. This tool can only be applied to the small, computationally inexpensive models typical of NetLogo, and cannot be distributed across machines, nor applied to noisy objective functions. The OpenMOLE framework (https://openmole.org) can distribute parallel simulations of a model across several kinds of clusters, either following a master-worker parallelization model or with island models. OpenMOLE supports models implemented in arbitrary languages and offers a Scala-based scripting language so that the calibration can be controlled from within a unified GUI. Its optimization algorithms are limited to classic GAs and NSGA-II, but it offers a few advanced features, such as a strategy for handling noise in the objective function.

The field of evolutionary computation has grown to encompass many techniques that perform considerably better than traditional GA-style algorithms or are applicable to a wider variety of tasks. A notable example is CMA-ES, which performs efficient vector-space optimization [5]. The wider EA family also includes many techniques for evolving complex computational structures, such as programs and neural networks [9, 21].

Some general-purpose evolutionary algorithm frameworks exist that offer massively distributed algorithms and configuration options useful for optimizing computationally expensive ABMs. The Evolving Objects (or EO) framework has long filled this role for C++ programmers [8]. The ECJ framework we use here has traditionally filled a similar role for Java programmers.

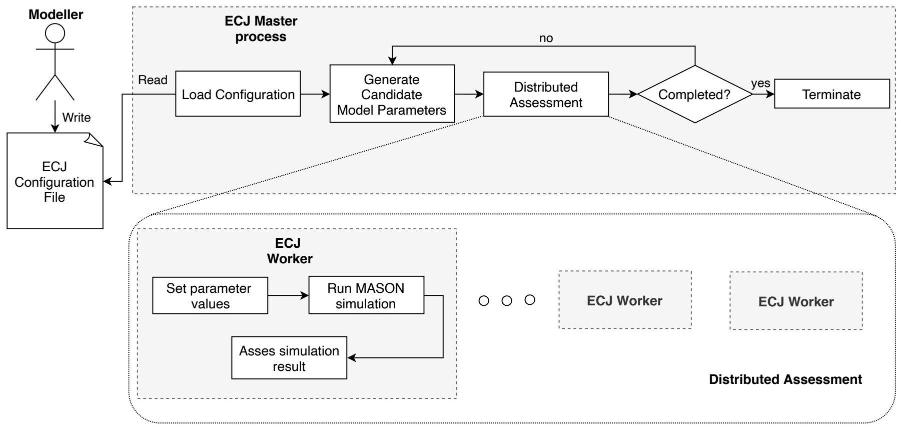


Fig. 1: Automated model calibration workflow using distributed ECJ and MASON.

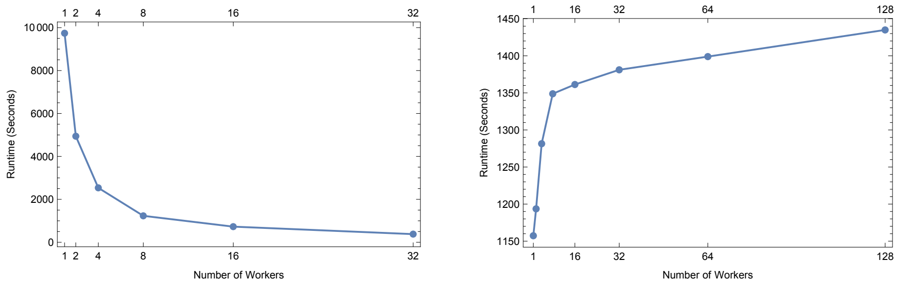
5 Approach

As agent-based models become more common and more detailed, an automated approach to calibration will be increasingly needed. We envision the automation process to work as follows. The modeler first builds the simulation, then assigns values to those parameters he knows or wishes to be fixed. A distributed system then optimizes the remaining parameters as best it can against criteria specified by the modeler. The modeler then examines the results: if they are poor, this could be due to bugs in the model, or insufficient model complexity to demonstrate the modeler's hypothesis, or a hypothesis that is wrong. Accordingly, the modeler revises the simulation and resubmits it to the system to be recalibrated.

To do this procedure, we merged ECJ and MASON to take full advantage of their technical characteristics. To merge them, it was necessary to make changes to both. Without going into implementation details: first, ECJ was modified so that MASON simulations could be used in the evaluation procedure of a candidate solution. Second, MASON was modified to be able to receive the values of model parameters from outside (that is, from an ECJ process) and to provide the modeler with a way to develop a score function for the simulation (which would then be used by the optimization algorithm).

Figure 1 shows the general workflow of the system. We first define one ECJ process to be the *master*. This process performs the top-level optimization algorithm. When this process has one or more candidate solutions (individuals) ready to be assessed, they are handed to a remote *worker* process. Each candidate solution is simply a set of those agent parameters and behaviors that we wish to test: the worker does this by creating a MASON simulation using those parameters and behaviors, running it some number of times, and assessing its performance. The worker then returns the assessments as the *fitness* (quality) of its tested solutions, and the master uses these results in its optimization procedure.

The modeler can completely customize this procedure if need be, and we demonstrate one such scenario in Section 6.3. But if the modeler's optimization



(a) Strong Scalability

(b) Weak Scalability

Fig. 2: Scalability Analysis, Refugee model

needs only involve global model parameters — as is typical for many ABM calibration scenarios — then we provide a simple alternative. The modeler specifies which parameters of interest to optimize, then selects from a few optimization options, and MASON does the rest: it defines the candidate solution representation as a fixed-length list of the parameters in question, builds the fitness mechanism, creates the evolutionary process, prepares the workers to run the proper simulation and default settings, then sends the candidates to remote workers.

6 Experiments

We begin with a large and nontrivial model drawn from GeoMASON's contributed model library, which we use to demonstrate speedup results for two different approaches to distribution. Then we highlight different capabilities of evolutionary optimization applied to model calibration using proofs of concept with a much simpler (and faster!) model drawn from MASON's demo suite. Finally, we demonstrate the distributed optimization of agent behaviors using genetic programming style parse trees.

6.1 Speedup Demonstration

We first show the efficiency of our distributed model calibration facility on a nontrivial ABM scaled horizontally across a cluster of machines. For this demonstration, we use the *Refugee* model drawn from the contributed models in the GeoMASON distribution. This model can take several minutes to run. *Refugee* explores the pattern of migration of refugees in the Syrian refugee crisis. The model demonstrates how population behavior emerges as the result of individual refugee decisions. The agents (the refugees) select goal destinations in accordance with the Law of Intervening Opportunities and these goals are prone to change with fluctuating personal needs.

We calibrated the model using a simple genetic algorithm from ECJ and assessed candidate solutions by comparing the number of arrivals in each city against real-world data gathered from UNHCR and EU agency databases.

Setup We calibrated over four real-valued parameters in the model. The model ran for 10,000 steps. We used a genetic algorithm with a tournament selection of size 2, one-point crossover, and Gaussian mutation with 100% probability and a standard deviation of 0.01. We ran the models on a cluster of 24 machines, each with Dual Intel Xeon E5-2670@2.60GHz, 24 GB, Intel 82575EB Gigabit Ethernet, Red Hat Enterprise Linux Server 7.7 (Maipo), OpenJDK 1.8.0. Running on these machines were some $N \leq 276$ MASON worker processes.

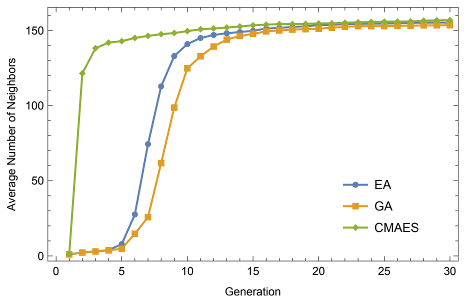
Results We performed *strong* and *weak scalability* analysis. Strong scalability asks how much time is needed for a *fixed size* problem given a *variable number of workers.* Weak scalability asks if an *increasingly difficult problem* can be handled in the same amount of time with a corresponding *increasing number of workers.* All the scalability results are statistically significantly different from one another ($p < 0.01$) as verified by a one-way ANOVA with a Bonferroni post-hoc test.

To do strong scalability analysis we fixed the problem to ten generations, each with 32 individuals, for a total of 320 evaluations. The number of workers was varied from 1 to 32. For 1 to 8 workers we gathered the mean result of ten experiments; for 16 and 32 workers (which ran faster), we gathered the mean result of 20 experiments. Figure 2a displays the speedup results. The *strong scalability efficiency* (as a percentage of the optimum) came to 71.88% using 32 workers to solve the problem.

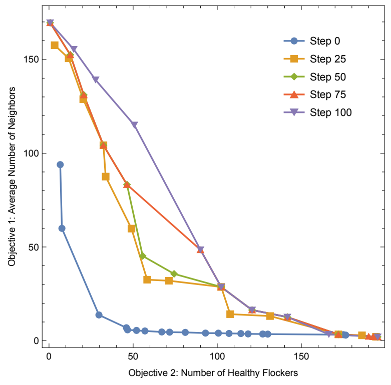
To do weak scalability analysis, we varied the problem difficulty by adjusting the population size such that, regardless of the number of workers, each worker was responsible for four individuals (and thus four simulation runs) per generation. In all cases, the results reflect a mean of ten experiments. For each optimization process the number of generations was fixed to 10 and the population size varied in $\{4, 8, 16, 32, 64, 128, 256, 512\}$, and thus the number of workers varied as $p \in \{1, 2, 4, 8, 16, 32, 64, 128\}$. Figure 2b shows the weak scalability results. The *weak scalability efficiency* (as a percentage of the optimum) was 83.18.

The previous experiment involved a *generational* evolutionary optimization algorithm: the entire population of individuals had to be evaluated on the remote workers before the next generation of individuals was constructed. We next considered an *asynchronous* evolutionary algorithm to improve efficiency when the model runtime varied greatly. An asynchronous evolutionary algorithm only updates the population a little bit at a time, rather than wholesale, and doesn't need to wait for slowly-running models.

The approach works as follows: there are some N workers and a master with a population of size P. The master first creates random individuals, then assigns each to an available worker. When a worker has completed its assessment, the individual is returned and added to the population, and the worker becomes available for another task. When the population has been fully populated, the master switches to a *steady-state* mode: when a worker is available, the master applies the evolutionary algorithm to produce an individual which is then given to the worker to asses. When an individual is returned by a worker, the master selects an existing individual in the population to be replaced by the new individual.



(a) Mean best-so-far performance of a Genetic Algorithm, Evolution Strategy, and CMA-ES on the Flockers domain, averaged over 30 runs.



(b) Pareto Nondominated Fronts (higher values preferred) for a typical run of the two-objective Flockers domain at generations 0, 25, 50, 75, and 100.

Fig. 3: Evolutionary Optimization Examples

Setup We compared the generational genetic algorithm from Section 6.1 against an asynchronous evolutionary algorithm using a steady-state genetic algorithm. When replacing an individual in the population, the steady-state algorithm selected the least fit individual. In our experiment, there were 128 workers, and the population size was 128: the generational approach was again run for ten generations, while the asynchronous approach was run until it had evaluated 1280 individuals. We performed twenty experiments per treatment. To simulate varying runtimes in the Refugee model, when a model was to be tested we changed the number of simulation steps at random. 1/4 of the time we halved them, 1/4 of the time we left them as normal, and 1/2 of the time we doubled them.

Results Asynchronous Evolution had a mean runtime of 293.77 seconds; while Generational Evolution had a mean runtime of 437.77 seconds. These results were statistically significantly different ($p < 0.01$) as verified by a one-way ANOVA with a Bonferroni post-hoc test.

6.2 Evolutionary Optimization Examples

So far we have shown speedup results to demonstrate performance: next, we turn to simple examples of some of many evolutionary algorithms approaches afforded by our facility, to illustrate capabilities of the system and justify their value in an ABM. Accordingly, the remaining demonstrations will be mere proofs of concept, and so will not be accompanied by statistical analysis.

Demonstration 1: Different Evolutionary Algorithms We begin with a demonstration of some different evolutionary algorithms to show breadth. We turn to the *Flockers* model, a standard demo model in the MASON library. This model is a simulation of the well-known *Boids* algorithm [17], where agents develop collective realistic flocking or swarming behaviors.

Flockers has five classic parameters (avoidance, cohesion, consistency, momentum, and randomness) that together define the behaviors of its agents. We optimized over these parameters and assessed the model performance as the mean number of flockers within an agent's neighborhood, averaged over three trials. This is not a hard problem to optimize: the calibration facility need only maximize cohesion. To show the optimizers at work, we fixed every individual in the initial population to represent the opposite situation (minimal cohesion, maximal values for other behaviors).

We ran for 30 generations using a population size of 276, spread over 276 separate workers. We compared three different evolutionary algorithms: the genetic algorithm as described before; a so-called "$(46, 276)$" *evolution strategy*; and a *CMA-ES* estimation of distribution algorithm with standard parameters. Figure 3a shows the performance of these three algorithms on this simple agent-based model: as expected, CMA-ES performs extraordinarily well.

Demonstration 2: Multi-Objective Optimization Next, we demonstrate our system's ability to optimize problems with multiple conflicting objectives. The classic approach finds a set of solutions that have advantages or disadvantages relative to one another with respect to these objectives. A solution A is said to *Pareto-dominate* another solution B if A is at least as good as B in all objectives and better than B in at least one objective. The optimal *Pareto Nondominated Front* is the set of solutions not Pareto-dominated by any other solution.

We extended the Flockers model by introducing an "infection" into the population. Healthy flockers have the same behavior as shown in the previous example, but infected flockers will, with some probability, infect their neighbors or be cured. Our new second objective was to maximize the number of healthy flockers. To do this, flockers must stay as far away from each other as possible, putting our new objective in direct conflict with the first one.

We used the NSGA-II [3] multi-objective evolutionary algorithm with four workers and 100 generations, having 24 individuals per generation. Figure 3b shows the improvement in the Pareto front over time for a typical run.

An Aside: Coevolution Though we do not provide demonstrations of them, it is worth mentioning two other capabilities of our system, which may be of value to an agent-based modeler.

In Section 3 we mentioned that one might wish to calibrate a model to be *insensitive to* one or more global parameters (parameter type 4 in that Section). For example, we might wish agents to perform migration the same way regardless of rain or shine. One attractive evolutionary optimization approach is *competitive coevolution.* Here we optimize the population A against a second *foil* population B of parameter settings simultaneously being optimized to trip up the first population. Thus while A is trying to be insensitive to B, B is searching for corner cases to challenge A.

A related technique, called *cooperative coevolution*, is a popular way to tackle high-dimensional problems. When the number of parameters to optimize is high, the joint parameter space is exponentially too large to efficiently search. Cooperative coevolution breaks the space into N subspaces by dividing the parameters into N groups, each with its own independently optimized population. Individuals are tested by combining them with ones from the other populations to form a complete solution. The fitness of an individual is based on the performance of various assessed combinations in which it has participated. This reduces the search space from $O(a^N)$ to $O(aN)$, but assumes that the parameters in each group are largely statistically unlinked with other groups.

6.3 Optimizing Agent Behaviors

Agent-based models are unusual in that not only do they have (typically global) *parameters* which must be calibrated but agents with *behaviors* that may benefit from calibration as well. Agent behaviors are essentially programs which dictate how the agents operate in the environment and interact with one another. Unfortunately, it is often the case that the modeler does not know what the proper behavior should be for a given agent, or only understands part of the behavior and needs to fill in the blanks with the remainder.

Because we are calibrating agent behaviors and not global model parameters, the modeler must do more than just specify a set of model parameters to calibrate and an optimization algorithm to use. He must also specify the *nature* of the representation of these agent behaviors (in our case below, an array of four parse-trees), and must also write glue code which, when given an individual, *evaluates* its parse trees in the simulation proper.

The evolutionary algorithm community has developed optimization techniques for a variety of agent behavior representations. Out of the box, we can support *policies* (stateless sets of *if→then* rules that determine actions to take in response to the current world situation), *finite-state automata* (as graph structures), *neural networks* (via NEAT), and untyped or strongly-typed "Koza-style" *genetic programming* (or GP) [9]; and provide hooks for a variety of other options.

In this example, we will focus on GP. Here, individuals take the forms of forests of parse trees populated by functions drawn from a modeler-specified *function set*. Functions may have arguments, types, and arbitrary execution order (like Lisp macros). Parse trees typically impact on behavior through side effects among their functions, or by returning some final result via the root of the tree.

Our example is drawn from the *Serengeti* model [11], in which four "lion" agents must capture a "gazelle" in a real-valued toroidal environment. The gazelle uses a simple hard-coded obstacle-avoidance behavior to elude the lions, and can move three times as fast as any single lion. The lions can sense the gazelle and each other. Each lion uses a GP parse tree that, when evaluated, returns a vector indicating the direction and speed the lion should travel at that timestep. Thus the behaviors to be calibrated consist of four different parse trees, one per lion.

We used a GP facility closely following the approach in [11], including its function set (we restricted ourselves to the "name-based sensing" and "restricted

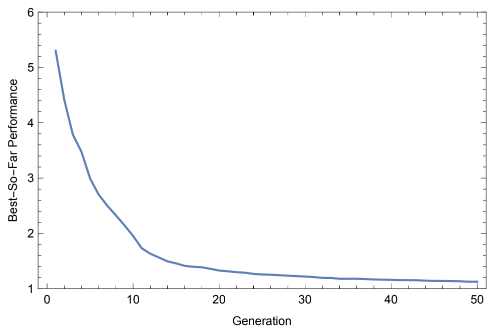


Fig. 4: Mean best-so-far performance, over 30 runs, of genetic programming on the Serengeti model (lower is preferred).

breeding" variants as described in the paper). We ran the GP algorithm as described, but with a population size of 5760 spread over 276 workers: each worker thus had 20 individuals per generation. Assessment of an individual's parse trees was performed over 10 random trials. Figure 4 shows the mean best-so-far performance of calibrated agent behaviors over 30 runs.

7 Conclusions

We have argued for the importance of automated model calibration for agent-based models. This will become only more pressing as these models increase in complexity and runtime, which will require the use of massively distributed evolutionary optimization tools. We have developed a tool of this kind which combines the popular MASON and ECJ libraries and have shown how their combination can produce a powerful, fully-featured model calibration facility with special capabilities of interest to the agent-based modeler. This preliminary work will be more tested and expanded with other functionalities, like a GUI, also including different types of simulation frameworks besides MASON.

References

1. Batty, M., Desyllas, J., Duxbury, E.: Safety in numbers? modelling crowds and designing control for the notting hill carnival. Urban Studies **40**(8), 1573–1590 (2003)
2. Canessa, E., Chaigneau, S.: Calibrating agent-based models using an improved genetic algorithm. In: International Conference of the Chilean Computer Science Society. pp. 25–29 (2014)
3. Deb, K., Agrawal, S., Pratap, A., Meyarivan, T.: A fast elitist non-dominated sorting genetic algorithm for multi-objective optimization: Nsga-ii. In: International conference on parallel problem solving from nature. pp. 849–858. Springer (2000)
4. Gilbert, N., Troitzsch, K.: Simulation for the social scientist (2005)
5. Hansen, N., Müller, S.D., Koumoutsakos, P.: Reducing the time complexity of the derandomized evolution strategy with covariance matrix adaptation (cma-es). Evolutionary computation **11**(1), 1–18 (2003)

6. Heppenstall, A., Malleson, N., Crooks, A.: "space, the final frontier": How good are agent-based models at simulating individuals and space in cities? Systems **4**(1) (2016)
7. Johnson, R.T., Lampe, T.A., Seichter, S.: Calibration of an agent-based simulation model depicting a refugee camp scenario. In: Winter Simulation Conference. pp. 1778–1786 (2009)
8. Keijzer, M., Merelo, J.J., Romero, G., Schoenauer, M.: Evolving Objects: a general purpose evolutionary computation library. In: Evolution Artificielle (EA). pp. 231–242 (2002)
9. Koza, J.R.: Genetic Programming: On the Programming of Computers by Means of Natural Selection. MIT Press, Cambridge, MA, USA (1992)
10. Luke, S., Simon, R., Crooks, A., Wang, H., Wei, E., Freelan, D., Spagnuolo, C., Scarano, V., Cordasco, G., Cioffi-Revilla, C.: The MASON simulation toolkit: Past, present, and future. In: International Workshop on Multi-Agent-Based Simulation (MABS) (2018)
11. Luke, S., Spector, L.: Evolving teamwork and coordination with genetic programming. In: Genetic Programming 1996: Proceedings of the First Annual Conference. pp. 141–149 (1996)
12. Mayer, D., Kinghorn, B., Archer, A.: Differential evolution–an easy and efficient evolutionary algorithm for model optimisation. Agricultural Systems **83**(3), 315–328 (2005)
13. Mongus, D., Repnik, B., Mernik, M., Žalik, B.: A hybrid evolutionary algorithm for tuning a cloth-simulation model. Applied Soft Computing **12**(1), 266–273 (2012)
14. Moya, I., Chica, M., Cordón, Ó.: A multicriteria integral framework for agent-based model calibration using evolutionary multiobjective optimization and network-based visualization. Decision Support Systems **124** (2019)
15. Nguyen, H.K., Chiong, R., Chica, M., Middleton, R.H., Dhakal, S.: Agent-based modeling of migration dynamics in the mekong delta, vietnam: Automated calibration using a genetic algorithm. In: IEEE Congress on Evolutionary Computation (CEC). pp. 3372–3379. IEEE (2019)
16. Olsen, M.M., Laspesa, J., Taylor-D'Ambrosio, T.: On genetic algorithm effectiveness for finding behaviors in agent-based predator prey models. In: SummerSim. pp. 15:1–15:12. San Diego, CA, USA (2018)
17. Reynolds, C.: Flocks, herds and schools: a distributed behavioral model. In: SIGGRAPH. pp. 25–34 (1987)
18. Rogers, A., von Tessin, P.: Multi-objective calibration for agent-based models (2004)
19. Rounds, E.L., Scott, E.O., Alexander, A.S., De Jong, K.A., Nitz, D.A., Krichmar, J.L.: An evolutionary framework for replicating neurophysiological data with spiking neural networks. In: International Conference on Parallel Problem Solving from Nature. pp. 537–547. Springer (2016)
20. Scott, E., Luke, S.: Ecj at 20: Toward a general metaheuristics toolkit. In: GECCO '19 Companion (2019)
21. Stanley, K.O., Clune, J., Lehman, J., Miikkulainen, R.: Designing neural networks through neuroevolution. Nature Machine Intelligence **1**(1), 24–35 (2019)
22. Stonedahl, F.J.: Genetic algorithms for the exploration of parameter spaces in agent-based models. Ph.D. thesis, Northwestern University (2011)
23. Venkadesh, S., Komendantov, A.O., Listopad, S., Scott, E.O., De Jong, K., Krichmar, J.L., Ascoli, G.A.: Evolving simple models of diverse intrinsic dynamics in hippocampal neuron types. Frontiers in neuroinformatics **12**, 8 (2018)