Generating Agent-Based Models From Scratch With Genetic Programming

Rory Greig and Jordi Arranz

1Improbable, London, UK, E1 6EG
rory@improbable.io, jordiarranz@improbable.io

Abstract

Program synthesis (PS) and genetic programming (GP) allow non-trivial programs to be generated from example data. Agent-based models (ABMs) are a promising field of application as their complexity at a macro level arises from simple agent-level rules. Previous attempts at using evolutionary algorithms to learn the structure of ABMs have focused on modifying and recombining existing models targeted to the domain in question, which requires prior domain knowledge. We demonstrate a new domain-independent approach which is able to evolve interpretable agent logic of an ABM from scratch. We employ a flexible domain-specific language (DSL) which consists of basic mathematical building blocks. The flexibility of our method is demonstrated by learning symbolic models in two different domains: flocking and opinion dynamics, targeting data produced from reference models. We show that the evolved solutions are behaviourally identical to the reference models and generalise extremely well.

Introduction

Agent-based models (ABMs) are an effective tool for building bottom-up models of complex systems by representing many agents individually (Wooldridge 2009). ABMs are capable of recreating emergent phenomena at the macro-level by employing principled and intuitive logic at the micro-level, i.e. individual agents. However, defining these behavioural rules at the agent level is a challenging task, and typically requires a detailed understanding of the underlying micro-level phenomena. This understanding can be time consuming to build, and often intuitions about the dynamics of a system can be misleading. Additionally, there are situations where we do not have sufficient information about the world to build an accurate model.

One approach for developing models which accurately recreate observed data is to tune numerical parameters of the model, referred to as calibration. However, this requires having an existing set of behavioural rules, and it is possible that these rules will not be sufficiently flexible, meaning that no combination of parameter values will allow them to match observed data.

Deep learning (DL) is a popular approach to learning models from data without having to encode detailed prior domain knowledge into the model (Bengio et al. 2013). Still, DL has two important limitations when applied to synthesising ABM logic: it requires a differentiable loss function (LeCun et al. 2015; Schmidhuber 2014) and lacks transparency, reducing our ability to “look inside” and interpret these models, which has practical, legal, and theoretical implications, i.e. the black-box-problem (Burrell 2016; Ribeiro et al. 2016).

ABM agents often rely on hidden or latent states which are not present in the observed data. Under these conditions calculating a loss function between the learned model output and target data requires running all the timesteps of the model sequentially, so that the hidden state calculated at $s_t$ can be used as the input for $s_{t+1}$. Running the full model typically requires non-differentiable operations, such as determining which agents interact with each other. This non-differentiable loss function makes DL unsuited to this problem.

Program synthesis (PS) / genetic programming (GP) avoids both these limitations, while retaining the ability to learn behavioural rules from data without manually encoding detailed domain knowledge. On one hand PS/GP does not require a differentiable loss function, hence it is still applicable to a larger set of ABMs including those with hidden states. Consequently, this approach can benefit from the inductive bias that all ABMs share the same structure of looping over agents with individual agent update functions. On the other hand PS/GP provide us with highly interpretable solutions — i.e., agent logic — which in turn can be used to illuminate the phenomena being modelled.

In this paper we present a GP technique for learning symbolic behavioural rules for agent-based models from scratch, without relying on a priori domain knowledge. We have developed a generic and flexible domain-specific language (DSL) designed for ABMs. Our implementation of the DSL can execute generated code at runtime and is optimised for high performance. It is shown that this technique can be applied to ABMs in multiple domains.
Our approach is closely related to inverse generative social science (Vu et al. 2019), which is a discovery process for testing multiple hypotheses automatically from data, in order to explain and illuminate observed phenomena. This technique enables us to generate models which are relatively free from existing domain priors and human preconceptions, and may shed light on completely new dynamics which have been overlooked because they are unintuitive or non-obvious. The output is an interpretable symbolic model which can be understood and extended by a human modeller, so this could be used for automatically building quick prototype models before a modeller refines them.

Program synthesis, also referred to as program induction (Manna and Waldinger 1971), is a computational method for generating computer programs from scratch by combining basic symbolic building blocks, see David and Kroening (2017) and Gulwani et al. (2017) for a review of both theory and applications. When genetic algorithms (GAs) are used for program synthesis this is referred to as Genetic programming (Koza 1992). GP is a challenging problem for evolutionary search, due to the large, sparse combinatorial search space, as well as the topology of the fitness landscape (Wright 1932), which tends to be rugged instead of smooth, with multiple local maxima separated by low fitness wells or valleys which increase the difficulty of finding and climbing a global maximum.

Program synthesis techniques have been around for several decades and have been successfully applied to many practical problems such as data extraction and string processing. For example Microsoft Excel FlashFill is able to induce a program that fills the blanks of a spreadsheet based on patterns from existing data (Gulwani 2011).

Recently, GP has been synthesized to solve social science problems from basic mathematical building blocks (Real et al. 2020). Authors show how the AutoML-Zero algorithm is capable of rediscovering fundamental deep learning algorithms such as two-layer neural networks with backpropagation by employing a GA (Whitley et al. 1989).

We propose that ABMs are a particularly promising application for program synthesis, since complex macro level behaviours emerge from simple logic at the individual agent level, which can be encoded with relatively few lines of code in the agent update function. Learning only the relatively concise agent update rules makes it feasible to evolve complex emergent behaviours without having to synthesize correspondingly long, complex programs, which remains a challenging problem for current GP techniques.

In this work we borrow from Real et al. (2020) among others, and apply these GP techniques to agent-based modelling (ABM). The technique was applied to learning the agent update function for models in two different domains: flocking and opinion dynamics. Since real data was unavailable, we generated the target by running hand-coded models taken from existing literature. Our approach allowed us to automatically generate ABM logic which closely approximates the target reference datasets.

**Related Research**

Automatically learning the model logic for ABMs is not a new idea, however the existing literature has focused on learning model structure by mutating existing models which are already adapted to the modelling problem in question, by recombining a set of “primitives” which is tailored to the domain. This is often referred to as “structural calibration”.

For example Vu et al. (2019) evolves the structure of a model of alcohol use in the US population to better fit real-world observed data, using GP to recombine primitives derived from an initial “tentative” model of the domain. This approach performs the calibration of parameter values and the structural calibration in separate optimisation steps. The evolved models are used to compare the effects of different variables and evaluate the plausibility of different hypotheses which could explain the data, which the authors present as an example of inverse generative social science.

Other works apply similar structural calibration techniques to different domains, such as opinion dynamics (Husseilman et al. 2015), archaeological simulations (Gunaratne and Garibay 2017), and human crowding models (Junges and Klügl 2011), (Zhong et al. 2017). In (Zhong et al. 2017) the authors use a “dual-layer” architecture with a lower level social force model for collision avoidance and a higher level navigation model, however only the navigation model is learned by evolutionary search. Conversely in the flocking experiment in this paper the whole model is learned, outputting a raw velocity for agents which captures their tendency to avoid collisions. Applying the techniques in this paper to a crowding problem would potentially allow more flexibility and remove the need for this dual-layer architecture and the social force model assumption.

There is also a significant body of work that employs evolutionary algorithms to evolve agent logic to solve cooperation and coordination problems. For instance, Koza (1993) evolved a solution to the painted desert problem. Arranz et al. (2011) evolved a classifier to solve a simple cooperative task that was previously solved with neuroevolution by Quinn (2001). Smith (2008) also evolves the rules for a classifier which determines the social behaviour of agents representing birds. In all these cases the results are highly interpretable, however the evolutionary process has to be steered by a fitness function which has domain knowledge built-in.

All of these examples start with an existing model or primitives tailored to the domain, however in this paper we start from scratch with an empty model and use a more flexible DSL. This takes structural calibration one step further and removes the need for a model with encoded domain knowledge as a starting point. This increased flexibility allows the evolution of models with fewer human priors.
Methods

In order to induce the agent update rule of two different models — flocking and opinion dynamics — we employ an evolutionary algorithm which evolves a population of individuals consisting of programs expressed in a DSL. In turn, these programs are combinations of the DSL operators and operands. The fitness of each individual in the population is evaluated by executing its program and comparing the output to the output of a reference model. The best programs are then copied and randomly changed — i.e. mutated. This process is repeated until the average fitness of the population reaches a desired threshold.

DSL

Learning useful programs in a general-purpose programming language with complex control flow is still considered to be an open challenge for program synthesis (Shin et al., 2019). This is because these languages are too flexible and result in huge search spaces which are difficult to explore. As a solution most PS techniques use DSLs, which are constrained programming languages, often subsets of other languages, targeted at a specific use case or domain. Using a DSL reduces the search space of possible programs, which can make PS more tractable. It also allows simpler custom mutation rules to be defined which are guaranteed to produce valid, executable programs.

A new DSL is presented in this paper for defining agent logic for ABMs, which is a subset of the programming language Julia (Bezanson et al., 2017). It is comprised of a set of basic mathematical operators which operate on scalar or vector values. It has limited control flow; only IF conditions are allowed, not loops or IF-ELSE statements. This is primarily to reduce the size of the search space. Loops are also avoided because they could increase the execution time of the learned program or result in infinite loops, which would require extra checks to handle. Future work could expand the DSL to incorporate more types of control flow.

Each operator is either a binary or unary function. The set includes simple mathematical operators, however arbitrary Julia functions could be added to this set by the modeller, if desired. The DSL is designed for expressing the agent update function of an ABM, which typically performs some mathematical transformation of their input values, and possibly some conditional operations. Despite these constraints it is still flexible enough to express a wide variety of ABM logic, since more complex mathematical functions can be comprised of simple operations. However, it is not Turing complete because it does not allow loops.

Agent update functions encoded in this DSL are hereafter referred to as behaviours. A behaviour is comprised of a list of instructions, which are interpreted line-by-line as opposed to being compiled. Each instruction has an op-code which is the index of an operator from the DSL. It also has input indexes which select the function arguments (these can either be input data about the agent’s environment, numerical parameters or specific memory registers). Instructions store their results in one of a finite number of memory registers, an output index determines which memory register to write to. This is an example of linear genetic programming, and differs from more typical genetic programming approaches which represent the program as a syntax tree and mutate that directly (Koza, 1994), rather than a linear list of instructions.

There is a special type of instruction which represents an IF condition, which can branch the program. The op-code specifies a conditional operator such as “less than”, and the output refers to the number of instructions to skip — e.g., the size of the IF block.

Additional configuration is required for the DSL depending on the target modelling problem, including the maximum number of instructions in a behaviour, the number of inputs, numerical parameters, and memory registers.

The set of operators in the DSL can be easily configured. The DSL uses strongly typed instructions, and currently only a single type is supported for a given experiment, either scalar floating point values or 2D vectors. The two experiments in this paper use different types and operator sets, the choice of which is governed by the required output type (2D vector for flocking, scalar for opinion dynamics). This typing adds additional constraints to the DSL, however it is not as restrictive as may be expected, since in the flocking case scalar values can be broadcast to 2D vectors.

The 2D vector operators are as follows: \([-, +, *, ÷, square, normalise, norm, clamp, reciprocal, max, min, exp, abs, relu, sin, cos, tan, log, sqrt, <, >, ==, ≠\]. This set of 23 operators was chosen to cover most basic mathematical operations, such as arithmetic, trigonometry and simple linear algebra. A smaller set of 13 scalar operators was used for the opinion dynamics experiment: \([-, +, *, ÷, <, >, ==, ≠, square, clamp, reciprocal, abs, identity\].

Both these sets of operators are much larger, and therefore more flexible, than those used in existing work, for example Gunaratne and Garibay (2017) uses a set of 4 operators and Zhong et al. (2017) uses 6.

Approximate versions of == and ≠ were used, with a tolerance of 0.001. To avoid causing numerical errors “safe” versions of some functions were used: for log and sqrt any negative input values were replaced with a very small positive number close to zero before calling the operation. For trigonometric functions, e.g. sin, any infinite values were replaced with zero.

Genetic Algorithm

We reimplemented the GA described in Real et al. (2020) and Real et al. (2019) which relies on steady-state selection and mutation but lacks recombination/crossover. Contrary to generational algorithms, steady-state algorithms do not replace their population at each generation — see Goldberg and Deb (1991) for a mathematical analysis of the different
selection methods. The authors also implemented the GA as an island model in order to mitigate typical early convergence towards local minima and the resulting critical loss in diversity. Island models are a type of “niching” technique which evolves multiple sub-populations at the same time, allowing periodical migration between them. The general underlying principle is that by evolving multiple genetic pools in parallel each one of them can explore different evolutionary trajectories, hence increasing the overall diversity of the meta-population and slowing down convergence (Starkweather et al., 1990). Furthermore, island models are also easy to parallelise by running each pool in a different thread/process which is an extremely desirable feature given the computational costs associated with GP.

Figure 1 shows a diagram of the training loop for this algorithm. It maintains a population of \( N \) candidate behaviours, all initialised as empty behaviours with zero instructions. At each step of the algorithm a subset, \( S \), of the population is chosen, the candidate with the best fitness in this subset is selected to reproduce, which is done by mutating this candidate with probability \( p \). If the candidate is mutated then the new fitness score is evaluated by running the loss function using the target data. This candidate is then added back to the population (whether it has been mutated or not) by replacing the oldest candidate, the fitness score is also stored. If the oldest candidate happens to have the highest fitness in the population then it is retained in order to guarantee that the best fitness score cannot decrease, and the second oldest candidate is replaced instead. In genetic algorithms retaining the best candidate is known as elitism. For all experiments the parameter values used were \( N = 128, S = 10, p = 0.9 \). The mutation probability, \( p \) and subset size \( S \) are the same values used in AutoML-Zero, the population size \( N \) was chosen to be at the lower end of the range of values used in AutoML-Zero.

Migration between pools occurs every 64000 ticks for the flocking model and every 80000 ticks for the opinion dynamics model. A longer period between migrations should help to maintain diversity and prevent early convergence, the period was larger for the opinion dynamics model because this is considered to be a more challenging search problem. Candidates are migrated by randomly sampling \( N/2 \) candidates from each pool and adding them to a global collection of candidates, then each pool replaces half of its members by randomly sampling from this global collection. Both experiments used 64 pools, which is slightly lower than the number used in AutoML-Zero, since this problem has a smaller search space. No hyperparameter optimisation was done for any parameter values.

The behaviour are mutated according to a set of mutation rules. These rules need to be “aware” of the DSL, so that it is guaranteed that any mutations result in syntactically valid behaviours. The following mutation rules were used: (i) add a random new instruction, or remove an instruction at a location randomly selected and replaced with another instruction; (ii) change the input arguments of an instruction, (iii) change the value of a numerical parameter, (iv) completely randomize a stretch of instructions by replacing them with random new instructions.

For each mutation event one of these rules was applied, with the choice of rule selected uniformly at random. The number of instructions to randomize is selected by sampling a Poisson distribution, in order to favour randomizing shorter stretches of instructions. Removing an instruction was twice as likely as adding a new instruction, in order to bias towards shorter behaviour lengths.

Numerical parameter values were randomly initialised by sampling from an exponential distribution. The parameter mutation rule selects a parameter and changes its value by flipping the sign with 50% probability and multiplying by a number sampled uniformly between 0.5 and 2.0. We are optimising the numerical parameter values alongside the structure of the model, which means we are effectively performing traditional ABM calibration as part of this optimisation process. Existing approaches have tended to perform these steps separately, for example Gunaratne and Garibay (2017) which recognised the benefits of integrating parameter calibration with learning the structure of the model, but considered this impractical due to computational constraints.

When adding or removing instructions the size of the IF blocks must be taken into account, i.e. the instructions to be skipped. When an IF instruction is added the block size is sampled from a uniform distribution. Nested IF blocks are allowed, the size of an inner IF block is capped at the size of the containing block. When an instruction is added or removed inside an IF block, the size of the block is adjusted.

**Loss Function**

In order to steer the evolutionary process to match the reference data we need an adequate fitness function. Since we
are employing data as the source of truth, our fitness function is a loss function between the output of the candidate behaviour and the results from the “true” reference model.

To create the target dataset the output from the reference model is converted to data pairs of inputs and outputs of the agent update function for every agent at every timestep of the model. The input is the agent’s current state and the state of the surrounding environment, and the output is the next value calculated by the true agent update function.

Since the data pairs can be treated as independent data points, the problem can be approached as a typical supervised learning problem where we want to be able to predict the dependent variable Y. In the present case Y represents the output of the agent update function. The loss function is calculated as the root-mean-squared error between the predicted and true outputs for every data pair. Given that the agents of the two models presented in this paper — opinion dynamics and flocking — do not have hidden states, for each data pair we only need to calculate the update function and nothing else.

**Flocking Experiment**

Flocking models recreate the collective movement of large swarms of animals, such as fish or birds. An influential example is Boids (Reynolds, 1987), an agent-based model where individual birds update their velocity based on their nearest neighbours. They use three simple steering behaviours: separation (avoiding collision), alignment (match velocity of neighbours), and cohesion (move towards the average position of neighbours).

This experiment investigates whether the steer function which calculates the velocity of each agent could be synthesised using evolutionary search, starting from scratch with an empty behaviour.

A hand-written implementation of the Boids flocking model was used as a “reference” model to produce target data for synthesising a new behaviour. The pseudocode for the step function which is run for every timestep is shown in algorithm. The aggregate sums of positions and velocities are stored in a spatial grid data structure so agents can efficiently access the position and velocity of neighbours within the same grid cell. Two grids are used at different resolutions; a coarse grid with a cell size of 280 units and a fine grid with a cell size of 12 units. The edge length of the whole grid space is 2000 units.

The reference steer behaviour is shown in listing and contains all of the logic for calculating the new velocity for each agent, this is the function to be replaced by a synthesised version. It takes as input the agent’s own velocity, the relative centre of mass of the neighbours in the same grid cell for both the fine and coarse aggregate grids, the aggregate neighbour velocity relative to the agent’s velocity, and the number of neighbouring agents for both aggregate grids.

The move() function is a simple physics calculation which adds the newly calculated velocity to the current position. It also handles the logic of wrapping around positions so that the agents move on a toroidal space. This function is not replaced by a synthesised version.

It is clear that the steer behaviour contains the bulk of the logic relevant to the specific flocking domain, i.e. it defines how birds update their velocities in relation to their neighbours. The logic outside this function simply updates the aggregate grids, and updates positions using known velocities. For this reason we say that if the steer function could be synthesised from basic mathematical building blocks, then the flocking model has been learned with minimal a priori domain knowledge.

**Listing 1: Reference steer behaviour**

```python
momentum = velocity * 0.28
separation = -1 * nearby.fine.position * 0.03
alignment = nearby.velocity * 0.32
cohesion = nearby.coarse.position * 0.0015
target.vel = momentum + alignment + separation + cohesion
new.velocity = normalise(target.vel) * 5.0
```

To produce the target data for the loss function a single trajectory of the reference model was run with 1400 agents for 25 timesteps, which gives 35000 data pairs. Note that the term trajectory here refers to the positions of all agents over time for a single run of the model, not the path of a single agent.

A genetic algorithm was run to minimise the loss between synthesised behaviours and the target data, using the set of 2D vector operators listed in the DSL section. The loss calculation only runs the steer() function, using the aggregate positions, velocities and neighbour counts from the target data.

**Flocking results**

Figure 2 shows how the loss decreases over time during training. There is a significant decrease in loss overall, with some large jumps at specific points. The end result is a low loss of 3.558, indicating that the best learned behaviour is able to reproduce the target data very accurately. This experiment took around 8 hours (wall clock time on a n2d-highcpu-32 Google Cloud Instance).

To understand this low loss value we can assess whether it corresponds to model results which are subjectively similar to the reference behaviour, and whether it captures important flocking model characteristics. Figure 3 compares the output of the reference model and the best learned behaviour. The plot shows the agent positions after running each model with these two behaviours.
Figure 2: Best loss over time of a single run of the evolution of the flocking model. The x-axis shows evolutionary time in terms of the number of individual replacements. The y-axis shows the best loss of the population. Simulation parameters: num_pools=64, pool_size=128, migration_period=64,000, mutation_p=0.9

for 80 timesteps, with coloured traces added to show the preceding movement of the agents, the yellow colour indicates more recent positions. Subjectively, these plots show clear similarity in the flocking patterns. Many of the agents are in approximately the same position in both plots, and recognisable movement patterns and clusters of agents can be identified. The results are not identical, but since this is a chaotic system it is expected that even minor differences in agent logic will be amplified over time and result in differing trajectories as the model progresses.

Listing 2: Simplified flocking generated code

```
params = [-7.0192e7, -1.6739e9, 111829.82, 8.6766e9, -1.6476e6, 214.2948, 5.4268e8, 4.9987]
temp_a = cos(params[3])
momentum = (temp_a + 1 - 2./params[6]) .* velocity
separation = (temp_a + 4./params[6]) .* nearby_fine_position
alignment = nearby_velocity
cohesion = (1./params[6]) .* nearby_coarse_position
new_velocity = momentum .+ separation .+ alignment .+ cohesion
new_velocity = normalise(new_velocity) .* params[8]
```

Figure 3: Comparison of a single realisation of the reference flocking model in a Cartesian 2D space (top) with the evolved flocking model (bottom). Every trace follows the trajectory of a single agent over time.

which are recognisable as the separation, alignment and cohesion concepts found in the original boids model. Being able to interpret this learned symbolic model is an advantage of the PS approach.

**Opinion Dynamics Experiment**

In order to demonstrate the generality of the PS technique introduced in this paper, it is necessary to show that it can learn models in multiple different domains. To show this, an opinion dynamics model was chosen as an alternative target. The model is a simple threshold model, as described in Deffuant et al. (2000). This model simulates how opinions on a topic spread amongst a population of agents. The basic model with complete mixing was used, meaning that agents can interact with any other agent, and these interactions are chosen randomly with a uniform distribution. Each agent has an opinion represented by a real number between 0 and
1. For an interaction 2 agents are selected, if the difference between their opinion values is below a threshold, \(d\), then they each move their opinions towards the other, according to a convergence parameter \(\mu\). The logic for this interaction is shown in algorithm 2. The threshold logic is motivated by the idea of homophily, meaning that two agents only move their opinions if they are already close to each other.

The interaction function in algorithm 2 was chosen as the target to be learned. As with the flocking model, this function contains the core domain logic of the model, the surrounding code simply selects which agents interact with each other. One difference from the flocking model is that this requires control flow in the form of conditional logic, so that the opinions only get updated if they are already within the threshold. Learning conditional logic presents an additional challenge for the PS search process. Even though the code in algorithm 2 is only 4 lines long, this actually corresponds to 11 instructions when encoded in the DSL, because each DSL instruction can only execute 1 operation at a time and write the result to a memory register, so all intermediate calculations must be performed with separate instructions. This is highlighted to show that this still presents a potentially challenging task for PS. This experiment used the set of scalar operators listed in the DSL section.

The target data for the loss calculation is the before and after state of each agent’s opinions for every interaction. To produce this, the reference model was run with 1000 agents for 25 timesteps, with 1000 interactions per timestep, which gives 25000 data pairs. The threshold, \(d\), was set to 0.2, and the convergence parameter, \(\mu\), was set to 0.5. As with the flocking experiment, only one trajectory — i.e., a single run — from the reference model was used for training.

**Opinion dynamics results**

Figure 4 shows the loss decreasing over time for the opinion dynamics experiment. As with the flocking experiment, this also achieves a very low loss, however it takes around an order of magnitude more replacements, implying that this is a more challenging program synthesis task. This experiment took around 24 hours (wall clock time on an Apple MacBook Pro with a 2.4 GHz 8-Core Intel Core i9).

The outputs from running the reference and learned behaviours are compared in figure 5. These plots show how the opinions of each agent change over the course of a model run. The learned behaviour is arguably able to produce output that is almost identical to the reference model, as would be expected given the low loss achieved in training. In both sets of results there are two main clusters of opinions, which is the main characteristic of the model output discussed in Deffuant et al. (2000).

Listing 3: Simplified opinion dynamics generated code

```
Algorithm 2: Opinion dynamics agent interaction

if abs(opinion_a - opinion_b) < d then
    opinion_a += \mu * (opinion_b - opinion_a);
    opinion_b += \mu * (opinion_a - opinion_b);
end
```

Figure 4: Best loss over time of a single run of the evolution of the opinion dynamics model. The x-axis shows evolutionary time in terms of the number of individual replacements. The y-axis shows the best loss of the population. Simulation parameters: num_pools=200, pool_size=128, migration_period=80,000, mutation_p=0.9

Listing 3 shows the code for the best learned behaviour, which has been manually simplified in the same manner as listing 2. This appears to have learned a polynomial function of the difference between the two input opinions, delta_opinion. The conditional statement evaluates to true when the result of this polynomial \(\approx 0.0\), when graphed this polynomial goes below 0.001 (the tolerance for the approximate equals) when the delta_opinion is between exactly -0.2 and 0.2. This means that the opinions only update when the magnitude of the difference in opinions is less than 0.2, which is the same as the reference model logic, with an identical value for the threshold, \(d\).

The raw generated code is 20 instructions long, and includes some redundant lines which do not affect the result. This hits the limit of 20 instructions for this experiment.
Figure 5: Side by side comparison of 40 time-steps of a single realisation of the reference opinion dynamics model (left) with the evolved opinion dynamics model (right). The x-axis shows simulation time and the y-axis shows the opinion value for each agent.

Evolved Models’ Generalisability
So far we have shown that the solutions evolved by the genetic algorithm are capable of replicating the macro behaviours of both models with high accuracy. A recurring issue with traditional AI/ML techniques is over-fitting of the solutions to the training data, meaning that the models are not able to perform adequately in new situations — i.e., with new data. Evolutionary computation and genetic algorithms are no exception as they are also prone to over-fitting — e.g., Gonçalves and Silva (2011); Gonçalves et al. (2012); Langdon (2011). This is particularly problematic in cases where the training data-set is relatively small.

In order to find out how well our solutions would perform with out-of-sample data — i.e., trajectories of the model which have not previously been seen during training/evolution — we ran two new experiments, one for each model. In both cases we compare the output of the reference implementation and the output of the best evolved solution for 1000 different trajectories of the models. The results in figure 6 show that both the best flocking and opinion dynamics solutions generalise extremely well despite the fact that they were only trained with a single trajectory of the reference model.

Discussion and Conclusion
In this paper we have demonstrated how PS and GP can be used beyond model calibration to learn full symbolic representations of core model logic, by only providing reference data and without encoding previous domain knowledge. This improves over existing work in this area by learning models from scratch (starting from empty behaviours) and employing a generic and flexible DSL consisting of basic mathematical and conditional operations. We have successfully synthesised an opinion dynamics model and a flocking model. Although we employed only a single trajectory of the reference model, in both cases the resulting models were able to generate identical macro behaviours. More importantly, we have also shown that the evolved solutions in both cases (i.e., the agent update rules) generalise very well and are highly interpretable. This level of accountability offers a huge advantage over most DL techniques which suffer from opacity, opening the door to applications that go beyond modelling such as inverse generative social science in which the synthetic model is employed to explain and illuminate the phenomenon being modelled.

The next natural step we are aiming for is to apply our approach to real-world observed data. Perhaps this could provide new insights into the underlying dynamics of bird flocking behaviour, or any other modelling domain this was applied to, due to the increased flexibility and reduced reliance on human domain priors.

The flexibility and generality of this method has been clearly proven by synthesising agent-based models in two different domains: flocking and opinion dynamics. However, to give further confidence in the lack of restrictions it would be informative to target models and datasets in a wider range of domains. It would also be valuable to determine how this technique copes with incomplete or noisy real world data.

Acknowledgments
Stefano Romano, Bill Edwards, Freyr Arinbjarnar, Matt Lewis, Sebastian Schmon, Patrick Cannon, Jordan Langham-Lopez. We also want to thank the reviewers and particularly reviewer 3 for hers/his thorough feedback.
References


Wright, S. (1932). The roles of mutation, inbreeding, crossbreeding, and selection in evolution.