1

Open Source Agent Based Modelling Frameworks

Russell K. Standish

School of Mathematics and Statistic
University of New South Wales
R.Standish@unsw.edu.au

1.1 Introduction

Artificial Life as a field of study was inaugurated by Chris Langton, who described it as the study of man-made systems exhibiting behaviours characteristic of life. As such it is complementary to traditional biology, locating *life-as-we-know-it* within the larger picture of *life-as-it-could-be*[20].

The core of artificial life research involves putting together simple computational objects, or *agents* that interact to produce "lifelike" behaviour. The key term is *emergence*[1] of nontrivial, possibly even unexpected, behaviour from the interactions of the agents.

The term "agent" is often used to refer to pieces of software exhibiting autonomy, reactivity, goal orientation and persistence[11] running on a computer or migrating between hosts of a computer network. Agent based modelling is not about these sorts of agents, per se. In an agent based model, the thread of execution is passed back to the simulation environment after each agent's method is completed, which is analogous to the way execution control is passed between independent tasks in a multitasking operating system. Nor do they often have goal orientation. What distinguishes agent based modelling from other sorts of modelling is a focus on building models "bottom up", constructing model systems from a large number of small interacting software components, that in themselves model a component of the modelled system. Thus in a model of schooling fish, each fish is individually modelled, as opposed to aggregate concepts like "school of fish".

In artificial life the focus is on the systems themselves, without reference to external systems except perhaps by analogy. The agents are often called

[1] The concept of emergence is a difficult one for philosophers to pin down. I can recommend [15, 3, 12] for discussions of the topic, and [34] for my own take on it. However, for the purposes of this chapter, we can rely on our intuitive understanding of novel behaviour emerging from the interactions between parts of a system.

digital organisms[32] to stress this fact. In *agent based modeling* (ABM), the same methodology of constructing artificial analogues from the bottom up is applied to modeling real world systems – forest fires, stock markets, traffic to name but a few. The agents in the model will correspond in some way to physical objects in the system being modelled.

In biology, *individual based modelling*[17] (IBM) has become increasingly important, as inadequacies of traditional population based modeling have become apparent. Individual based models track individual animals or plants rather than population aggregated quantities. Individual based models may be constructed in an agent-based way, with computational objects representing each individual organism, or alternatively each individual is represented by a collection of numbers, and the population of individuals is therefore a collection of vectors. For the sake of clarity, let us call this type of model a *vector-based IBM*. Conceptually, a vector-based IBM is little different from a specialised agent based model, where each agent consisting of a collection of numbers, however in practice data is laid out differently in the computer's memory, which leads to substantially different performance characteristics.

In physics, the main form of individual based modelling is molecular dynamics (MD) simulations. Here, each molecule of the physical system of interest is represented by a collection of numerical properties: position, momentum, mass, charge etc., and the corresponding position and momentum vectors are updated according to the laws of classical dynamics. Whilst MD simulations could be implemented in an agent-based fashion, it is rarely done due to the performance degradation experienced in doing so.

Many artificial life systems have been created over the years, of prominent note are Tierra, Avida, Echo, Framsticks to name but a few of the most well known. Each of these systems is implemented from ground up in a general purpose programming language like C or C++. A typical simulation needs to implement not only the agents, but also an environment, an event generator, a means to specify input parameters, as well as visualisation and analysis tools. In an attempt to introduce some commonality and code reuse between these disparate artificial life models, Langton initiated the Swarm project, which produced an agent-based modeling platform into which scientists could insert their agents into an environment adapted from a library of containers (aka "Swarms"), and use event generators and visualisation probes to analyse the progress of the simulation.

Over the years, a number of similar agent-based modeling platforms have been created, each with a differing rationale for existence. Each platform specifies a particular implementation language for the agents and has a different balance between performance, scalability, generality and usability.

This chapter surveys open source (Sect. 1.3.1), agent based modeling platforms. Being open source is important, for ensuring replicability of results between different research groups, and also for auditing against implementation artifacts. This chapter does not examine commercial agent based modelling options.

1.2 Applications

Agent based models have been used in a wide variety of application areas, so this section will necessarily be illustrative. ABMs are commonly used in social science settings, to test theories for why particular customs have arisen. Any recent issue of the *Journal of Artificial Societies and Social Simulation* will provide any number of agent based models. Economics too use agent based models to model the behaviour of markets. Here, though, agent based models compete with more traditional Monte Carlo techniques that model each economic agent as a simple set of state variables.

Another important area of application is traffic modeling, with vehicles in a road network being represented by software agents with intended destinations, and differing levels of behaviour (eg conservative, experimental and so on). Here the concern is quite pragmatic – what happens if a new road is created here, or another road blocked? Similar considerations have motivated research into modelling crowd behaviour within restricted environments such as a sporting venue.

Grimm[14] presents a decade health check of individual based modelling in ecology. Huston et al. argued that individual based models had the potential to "unify ecological theory"[17], yet Grimm found that a decade on from Huston's paper, this potential had not been realised. Individual models have their place in answering particular questions inaccessible to more traditional simulation techniques, but linking the results back to theoretical concerns has not proved easy.

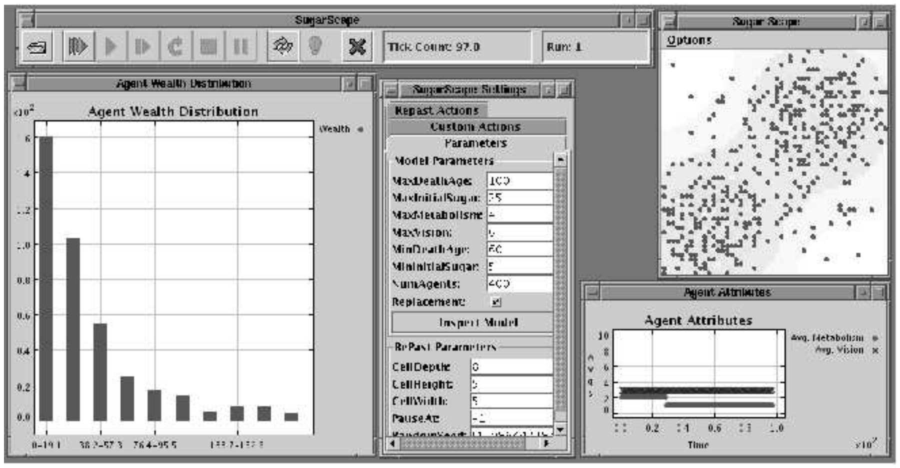
Finally, to artificial life, the field that inspired Swarm and later agent based modelling platforms. Agent based modelling is an important tool for the generation of complex life-like behaviours, others being cellular automata and boolean networks[38]. However, curiously, general purpose agent based modelling environments such as Swarm and Repast have rarely been used in the artificial life literature, with researchers developing, or making use of more special purpose simulation software such as Avida[1].

The following specific models are well known classic models that illustrate the sorts of modelling ABM's are applied to. They are exemplars only, agent based modelling as a field has grown far beyond the bounds of a single chapter like this one.

1.2.1 Sugarscape

Sugarscape[9] is a classic agent based model of a society of agents living on a 2D grid. Each agent has properties of metabolism and vision, inherited from their parents. Agents have separate requirements for "sugar" and "spice". They need both goods to survive and exhaustion of either will lead to death by starvation. The agents therefore need to search for these goods and accumulate them in order to survive. They do so by following a simple set of connected instructions referred to as a ruleset. For example, the ruleset for gathering is:

1. Evaluate personal stocks and determine which good is needed more urgently (preferred good).
2. Look around as far as vision permits and identify the site with the greatest value of the preferred good.
3. If the greatest value exists at multiple locations, select one randomly.
4. Move to that site and harvest all resources from that site.
5. If no value is found within the visible grid, then the citizen randomly relocates to one of the farthest cells within its vision range.



Model Parameters
MaxDeathAge: 100
MaxInitialSugar: 25
MaxMetabolism: 4
MaxVision: 6
MinDeathAge: 60
MinInitialSugar: 5
NumAgents: 400
Replacement: ✓

RePast Parameters
CellDepth: 0
CellHeight: 5
CellWidth: 5
PauseAt: -1

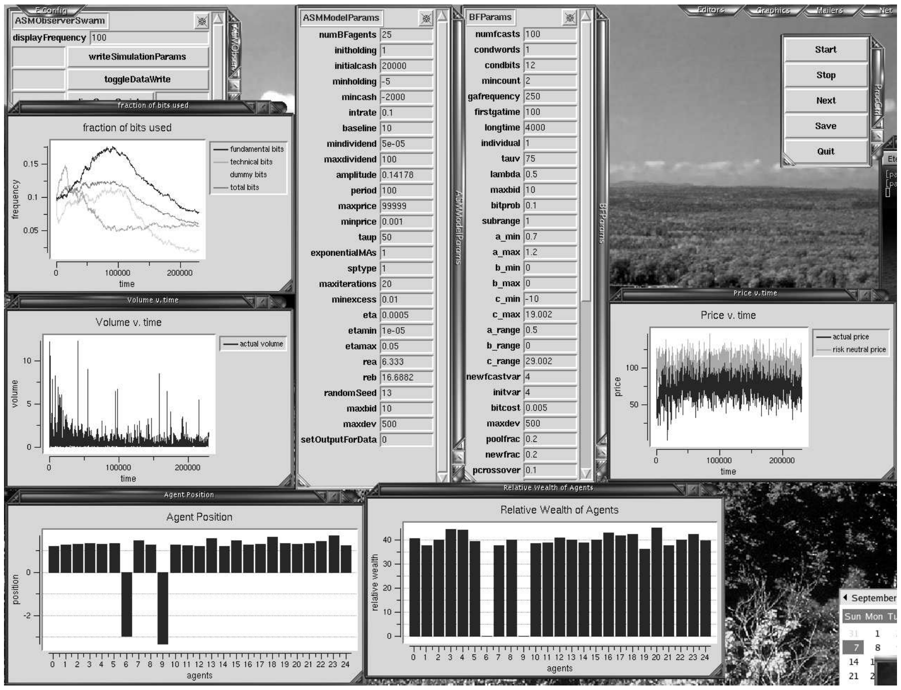
Fig. 1.1. Sugarscape model implemented in Repast

Agent replacement is either handled randomly upon the death of any agent, or agents will mate according to the mating ruleset, giving rise to offspring agents. The offspring agents inherit part of their parents' stores of sugar and spice.

Agents need both sugar and spice to survive. They have independent metabolism rates for each resource. There may arise situations where agents starve to death due to a paucity of one resource, despite having a plentiful supply of the other. Sugarscape allows agents to trade resources. The personality of the agent is an important attribute affecting their approach to trade. Personality is randomly assigned at birth and determines the trading strategy pursued by the citizen. A bear (cautious) personality seeks to minimize surplus and will only trade the surplus commodity. A bull (aggressive) personality seeks to maximize trades even if it involves trading the scarce commodity. The bull only trades the minimum quantity required to receive one unit of the other commodity. By maximizing trades, the bull seeks to hedge its exposure to unfair trades. For instance the bear seeks trading partners that possess a surplus of its scarce commodity. It then attempts to trade a certain proportion of its surplus such that the quantity received in exchange can be combined with the balance of its surplus to mitigate the risk of depletion of any one commodity. The bear strategy is at risk of wild fluctuations in the exchange price depending on the variance in the marginal rate of substitution values (perceived value of spice relative to that of sugar) of the trading partners. Since they attempt to sell all available surplus immediately, bears could end up trading all their surplus in a single unfavorable trade. The bull strategy, by trading unit amounts with as many traders as possible, seeks to average out price fluctuations and arrive closer to the equilibrium price.

Sugarscape has been implemented in Swarm, Repast, Mason, Cormas and NetLogo, with the Mason version being perhaps the most complete.[2]

1.2.2 Santa Fe Artificial Stock Market



ASMObserverSwarm

displayFrequency	100

ASMModelParams

numBFagents	25
initholding	1
initialcash	20000
minholding	-5
mincash	-2000
intrate	0.1
baseline	10
mindividend	5e-05
maxdividend	100
amplitude	0.14178
period	100
maxprice	99999
minprice	0.001
taup	50
exponentialMAs	1
sptype	1
maxiterations	20
minexcess	0.01
eta	0.0005
etamin	1e-05
etamax	0.05
rea	6.333
reb	16.6882
randomSeed	13
maxbid	10
maxdev	500
setOutputForData	0

BFParams

numfcasts	100
condwords	1
condbits	12
mincount	2
gafrequency	250
firstgatime	100
longtime	4000
individual	1
tauv	75
lambda	0.5
maxbid	10
bitprob	0.1
subrange	1
a_min	0.7
a_max	1.2
b_min	0
b_max	0
c_min	-10
c_max	19.002
a_range	0.5
b_range	0
c_range	29.002
newfcastvar	4
initvar	4
bitcost	0.005
maxdev	500
poolfrac	0.2
newfrac	0.2
pcrossover	0.1

Fig. 1.2. A screenshot of the Artificial Stock Market

The "Santa Fe" artificial stock market was developed by Brian Arthur, John Holland, Blake LeBaron, Richard Palmer, and Paul Taylor [28, 2]. The market consists of a population of heterogeneous agents that buy, sell, and

[2] Sean Luke, private correspondence.

hold stocks and bonds. An agent's buy, sell, and hold decisions are made on the basis of that agent's beliefs about whether the stock's dividend is likely to go up or down, and those beliefs are determined by a set of market forecasting rules that are continually being assessed as to accuracy. Over time an agent's set of market forecasting rules evolve under the action of a genetic algorithm.

The market contains a fixed number N of agents that are each initially endowed with a certain sum of money. At each timestep each agent must decide whether to invest her money in a risky stock or in a risk-free asset analogous to a real world Treasury Bill. The risk-free asset is in infinite supply and pays a constant interest rate r. The risky stock, issued in N shares, pays a stochastic dividend that varies over time. The stock's dividend stream is an exogenous stochastic process whose present value is unknown to the agents.

Agents apply their market forecasting rules to their knowledge of the stock's price and dividend history to perform a risk aversion calculation and decide how to invest their money at each time period. The price of the stock rises if the demand for it exceeds the supply, and falls if the supply exceeds the demand. Each agent in the market can submit either a bid to buy shares, or an offer to sell shares – both at the current price p_t – or neither. Bids and offers need not be integers; the stock is perfectly divisible. The aggregate demand for the stock cannot exceed the number of shares in the market. The agents submit their decisions and offers to the market specialist – an extra agent in the market who controls the price so that his inventory stays within certain bounds. The specialist announces an initial trial price, collects bids and offers from agents at that price, from these data announces a new trial price, and repeats this process until demand and supply are equated. The market clearing price serves as the next period's market price.

The agents make their investment decisions by using a set of hypotheses or rules about how to forecast the market's behavior. At each time period, each agent considers a fixed number of forecasting rules. The rules determine the values of the variables a and b which are used to make a linear forecast of next period's price:

$$E(p_{t+1} + d_{t+1}) = a(p_t + d_t) + b \tag{1.1}$$

where p_t is the trial price, d_t the dividend and a and b are the forecasting parameters. The forecasting rules have the following form:

$$\textbf{if}\ \ (\textit{the market meets condition } D_i)\ \ \textbf{then}\ \ (a = k_j,\ b = k_l) \tag{1.2}$$

where D_i is a description of the state of the market and k_j and k_l are constants.

Market descriptors (D_i) match certain states of the market by an analysis of the price and dividend history. The descriptors have the form of a boolean function of some number of market conditions. The set of market conditions in each rule is represented as an array of bits in which `1` signals the presence of a certain condition, `0` indicates its absence, and `#` indicates "don't care". The breadth and generality of the market states that a rule will recognize is proportional to the number of `#` symbols in its market descriptor; rules with descriptors with more `0`s and `1`s recognize more narrow and specific market states. As these strings are modified by the genetic algorithm, the number of `0`s and `1`s might go up, allowing them to respond to more specific market conditions. An appropriate reflection of the complexity of the population of forecasting rules possessed by all the agents is the number of specific market states that the rules can distinguish, and this is measured by the number of bits that are set in the rules' market descriptors.

An example may help clarify the structure of market forecasting rules. Suppose that there is a twelve bit market descriptor, the first bit of which corresponds to the market condition in which the price has gone up over the last fifty periods, and the second bit of which corresponds to the market condition in which the price was 75% higher than its fundamental value. Then the descriptor `10###########` matches any market state in which the stock price has gone up for the past fifty periods and the stock price is not 75% higher than its fundamental value. The full decision rule

$$\textbf{if}\ \ \texttt{10\#\#\#\#\#\#\#\#\#\#}\ \ \textbf{then}\ \ (a = 0.96,\ b = 0) \tag{1.3}$$

can be interpreted as "If the stock's price has risen for the past fifty periods and is now not 75% higher than its fundamental value, then the (price + dividend) forecast for the next period is 96% of the current period's price."

If the market state in a given period matches the descriptor of a forecasting rule, the rule is said to be *activated*. A number of an agent's forecasting rules may be activated at a given time, thus giving the agent many possible forecasts to choose from. The agent decides which of the active forecasts to use by measuring each rule's accuracy and then choosing at random from among the active forecasts with a probability proportional to accuracy. Once the agent has chosen a specific rule to use, the rule's a and b values determine the agent's investment decision.

The Artificial Stock Market website is implemented in Swarm (both Java and Objective C versions), and available from http://artstkmkt.sourceforge.net.

1.2.3 Heatbugs

Heatbugs was originally written a "demonstrator" model for Swarm, illustrating the main techniques for setting up agents in a 2D grid, and having the agents interact with the environment. Because it is a fairly simple, yet non-trivial model, and well documented, it has been ported to a number of other agent-based modelling environments, including Repast and Mason.

An agent in Heatbugs (the "bug") emits a certain quantity of heat per timestep into the environment, which then diffuses by the standard heat equation. Each heatbug has a preferred temperature, so by tuning the model parameter one can see the formation of clusters of bugs that manage to heat their environment to around the desired temperature.

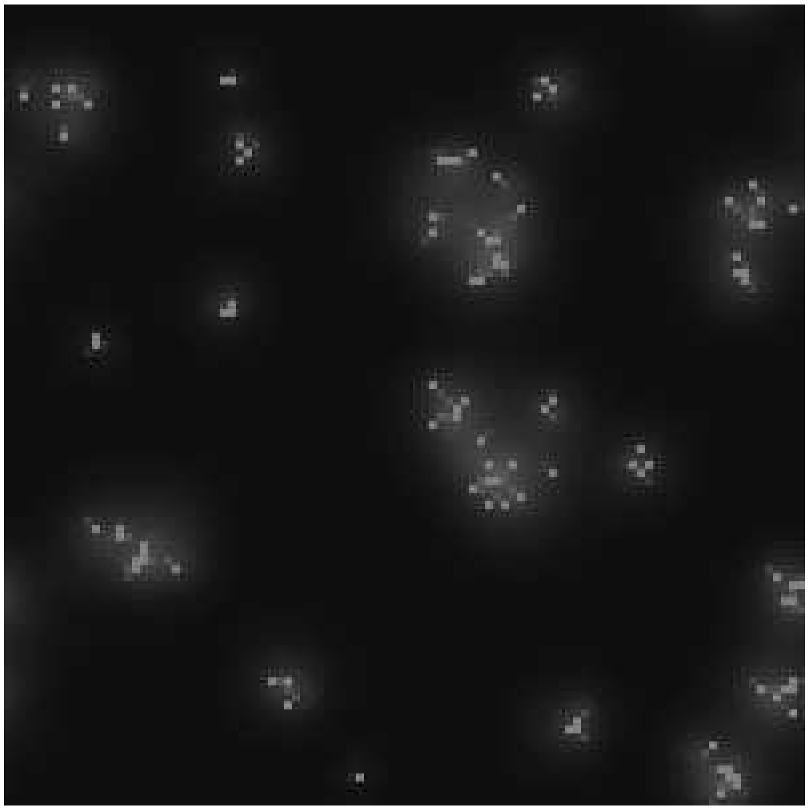


Fig. 1.3. Snapshot of the world display of the Heatbugs model, running under Repast

1.2.4 Mousetrap

Mousetrap was again a demonstrator model for Swarm, to illustrate the use of dynamic scheduling. It consists of a plane of loaded mousetraps. The initial event consists of a ball dropped on the central mousetrap, which releases the mousetrap sending two balls at random to other mousetraps, releasing them in turn in a chain reaction. In fact, the mousetrap model was originally introduced as popular means of conveying the idea of a nuclear chain reaction[31] There is no concept of a timestep, actions happen when caused by previous actions.

Mousetrap has been implemented in Swarm, Repast and MASON.

1.3 Software Modelling Tools

1.3.1 Open Source versus Freeware

The English language has an unfortunate ambiguity with the word “free”, which can mean free of restrictions or alternatively available for zero cost. This ambiguity is not always present in other languages, eg French *libre/gratuit* or German *frei/kostenlos*. The Free Software Foundation[3] defines free software as having freedom from restrictions on how to use or distribute the software. Think of “free” as in “free speech,” not as in “free beer”.[10] Free software need not be free as in beer, as one may still have to pay distribution costs (media, handling charges etc), but in practice the modern internet means these charges are negligible.

One of the most important characteristics of free software is that of being *open source*, namely that the source code for the software is publicly available for study and improvement.

In scientific modelling, open source is crucial to allow independent validation of scientific experiments. Often, when computational experiments are replicated by a second research group, differences in behaviour are observed between the original reported results, and the reproduced experiment. It becomes important to understand whether the problem is due to implementation bugs in either the original, or replicated code, or whether the published model specification is inadequate[23].

If the source code of the computational experiment is openly published along with scientific article describing the experiment, then it becomes possible for later researchers to tease apart any anomaly that might appear. Unfortunately, it is not yet commonplace for researchers to publish the source code of their experiments, however the artificial life community encourages the practice through asking reviewers to check and comment on the availability of experimental source code.

What about the software components used to build the computational experiment? To help fix the magnitude of the problem, it is worth imagining being a researcher thirty years from now attempting to replicate a contemporary experiment (as McMullin found himself attempting to replicate Varela’s work[23]). Assume that the source code used for the original experiment was available. To be able to rerun the original experiment, you would need a compiler for the language the experiment was coded in, and also a copy of any libraries used. Since you probably do not have access to the original hardware (how many thirty year old functional computers are you aware of), and unless you have a functional emulator, you will need the source code for any libraries as well.

The language used in coding will probably no longer be used (Fortran and C are exceptional in being languages maintaining backward compatibility over this sort of timeframe), so you may also need an open source copy of the

[3] http://www.fsf.org

language compiler. At very least, with well documented language standards (eg ANSI/ISO standard C, C++) and associated standard libraries, it may be possible to manually translate an existing open source program into a modern language with some sort of fidelity.

Agent based modelling systems fall into the "any libraries" category. Since it is unlikely for any ABM system to be perfectly documented, nor for the scientific report to list precisely which version of the tool's specification is used, nor for the actual ABM system to be bug-free, it is vitally important that the ABM system's source code is available for perusal.

As mentioned above, it is perhaps not so important for the implementation language and standard library to be open-source, provided it is one of the well documented standard languages. However, it is important that the language is freely available (as in beer), to remove any barriers to independent verification of computational models. The use of Java is a case in point. Java is developed by Sun, with a well documented language and standard library, and a free (as in beer) reference implementation available for most modern computing platforms. With the GNU gcj project[4], Java will become an open source option in the future, providing the language does not evolve too fast for the gcj development effort to keep up.

1.3.2 Programming Languages

Traditional scientific modelling has been implemented using a general purpose high level language such as Fortran, C and more recently C++ and Java. Standard libraries of numerical methods are employed where relevant, but these tend to be oriented towards models expressible in terms of linear algebraic operations: vectors, matrices and so on. Much of a scientific code deals with reading model parameters, and reporting results. If the calculation takes more than a day or so to complete, additional code needs to be added to allow the calculation to be paused, resumed and migrated as computational resource availability varies. Further code will need to be added to distribute the calculation across multiple processors to enable computations to finish in a reasonable time. The upshot is that a sizable fraction, perhaps as much as 50% of the lines of code of a scientific application, is not directly implementing the scientific calculation, but performs these incidental tasks.

The amount of extra effort needed to obtain a functioning scientific application has lead in many areas to "application frameworks", where for a limited range of scientific models, users can plug in problem specific methods. One example of this is in the area of *computational fluid dynamics*, where the leading packages *Fluent* and *CFX* allow users to supply subroutines coded in C or Fortran to implement specific physical models not supplied in the core functionality.

[4] http://gcc.gnu.org/java

Agent based models have not only the usual demands for scientific models, but also need interactive modes of exploration. Many phenomena of interest may only occur in specific scenarios, so it is useful to be able to restart the model from a known point, and to drill down to individual agents and their interactions, to establish what agent behavior is essential for the phenomena to occur. The process of designing an agent based model typically involves an interactive "playing with the model" stage, in which the modeler develops a feel for how the agents behave, and what might be the most significant parameters of interest, followed by a second stage of "parametric" exploration to establish what range of model parameters the phenomenon of interest occurs.

The first question to ask of any agent based modelling platform is what language is used to implement the agents. Since agents have *behaviour*, they cannot be represented just by numbers, as might be the case in other scientific computations. Some simulation systems allow a limited range of agent types to be implemented without programming knowledge (eg StarLogo or RepastPy), but for greatest generality, agents should be implemented in a general purpose programming language, preferably object-oriented as this matches the agent based paradigm.

The choice of ABM systems surveyed in this chapter is organised by this agent implementation language. I have chosen systems that use a widely implemented object oriented programming language, as this allows programmers familiar with that language to become productive in a short period of time. Also the design of a new language is a nontrivial task – using existing languages ensures that bugs have known workarounds, and that efficient implementations are available.

The most popular ABM systems are based on the Java language, an object oriented language influenced by the C/C++ language family, but from the outset designed to be a simpler language than C++, in both usability and functionality. Java compilers are freely available for most platforms, and the language and its system library publicly documented. Whilst *open source* Java compilers are not as mature as the closed source options[5], as mentioned previously this is not such an issue for scientific computing. Nevertheless there are issues with Java's floating point model one should be aware of.[19]

Because Java is widespread, and is a simpler language to learn, it is often the language of choice for learners of object oriented programming. Its main competitors are C#, which is still somewhat tied to the Windows .Net environment (although Mono[6] is now available as an open source implementation of the .Net runtime and C# compiler), and C++, whose feature rich capabilities typically take a couple of years to master.

Java, and its .Net cousins compile to a *virtual machine*. By providing a uniform machine model for the compiler to target, it is easier to write portable

[5] As this chapter was being prepared, Sun has open-sourced the core parts of its Java development environment, so even this is no longer an issue

[6] http://www.mono-project.com

code. However, the virtual machine needs to be emulated by the physical computer, and this introduces a performance penalty. The use of *just in time* compilation technology substantially reduces this performance penalty, though not completely. For full performance, which is just as important in ABMs as in general scientific computing, a compiler that targets the native machine is needed. For Java, there is an experimental Java front end for the GCC family of compilers called gcj. It can not only compile Java source code to Java byte code (replicating Sun's Java compiler), but also compile the byte code directly to native machine code. That said, in a direct comparison of the same simple agent based model implemented in Java using Repast, and C++ using EcoLab, both the Java and C++ versions executed at the same speed (Sect. 1.4).

For compiling to the native machine, the main choices are Objective C (as used in Swarm) and C++ (as used in EcoLab). Objective C was chosen by the Swarm project as being a very minimal object-orient extension to C that would not impose too much of a learning curve on prospective users. Unfortunately, the very simplicity of this language means that object management is largely the responsibility of the programmer, and places a large burden on the programmer to get the code functioning correctly. One other downside to Objective C is that GCC is the only commonly available compiler, and GCC tends not to produce as well optimised code as do commercially available compilers (although on Intel x86 architectures, the reverse can often be true).

The other mainstream language is C++, which has consistently held the 3rd spot (behind Java and C) in the *TIOBE Programming Language Community Index*[7] over the last 5 years. Vendor optimised compiler optimisers are available for obtaining performance, and an open source reference implementation exists (GCC). There are two main advantages of C++ over Java: more opportunities for optimising performance and operator overloading. *Operator overloading* allows the creation of mathematical types like vectors, or complex numbers, and express mathematical operations on them using conventional algebraic notation (`+`, `*` etc.) instead of functional notation (`add(,)`, `mul(,)` etc.). This is an important feature in scientific computing.

Ultimately, the choice of ABM platform should probably depend on the programming language you are most familiar with. If you are familiar with Java, then Repast or Mason would be a good choice. If C++ is your familiar language, then EcoLab would make a good choice. If C was your language, then you might consider Swarm. If programming is not your forte, then perhaps NetLogo might make a good choice for dipping into the world of agent-based modelling.

1.3.3 Reflection

As previously mentioned, a large part of a typical scientific code is involved in reading in the model's parameters, and in providing checkpoint-restart

[7] http://www.tiobe.com/tiobe_index/

functionality for long running codes. For a rapidly evolving model, as many agent-based models are, each time the model accumulates another instance variable, or deletes one, this ancillary code needs to be updated to keep track of the changing model.

The notion of *reflection* is the ability to determine an object's structure at runtime, the names and types of all its instance variables, and lists of methods. By using reflection, this ancillary code can automatically track model changes without further burden on the programmer. Furthermore, the concept of a *probe*, or a dynamic visualiser of agents making up the model system, needs to make use of reflection to understand and represent the instance variables and methods of the object.

Unfortunately, traditional compiled languages such as C and C++ throw away this information at compile time, whereas other popular languages such as Java and Objective C have inbuilt reflection capability. Reflection was the other main reason for the choice of Objective C over C++ in the Swarm system.

For C++, EcoLab uses a C++ language processor called Classdesc[22], that emits overloaded C++ function calls that walk the structure of the object. This allows serialisation of objects to a binary representation for checkpointing and other functionality, and exposure of object internals for probing.

1.3.4 User Interface and Scripting

All the agent-based modelling frameworks mentioned here have a GUI interactive mode with the ability to attach probes to objects, and to plot basic statistics and display histograms of the system behaviour. Once interactive development of the model is over, it is then usually desirable to turn off all graphical elements and run the model from a batch script. Only EcoLab has this capability without recompilation, as all graphical elements are implemented as distinct script commands from those that implement the model. Other models require distinct BatchModel and ObserverModel implementations.

EcoLab's script interface (which uses the TCL programming language) has the advantage that model parameters can be simply set from the script without the programmer having to write a single line of I/O code. Ousterhout[27] eloquently argues for the advantages of scripting interfaces to improve the plasticity of software, particularly for the development phase. With scientific codes, the development phase is often never finished. None of the other ABM packages offer a script interface, however Repast does offer a simple parameter file syntax, that allows just the setting of constant value parameters. Furthermore, RepastPy integrates the Python scripting language into Repast to produce a simple to use rapid development environment.

The TCL scripting language used by EcoLab also includes a complete platform independent GUI programming environment. This technology is also used by Swarm, but encapsulated to hide the TCL interface from the user.

Java systems have their own GUI programming environment, in fact several are possible: AWT, Swing and SWT.

1.3.5 Discrete Event Scheduling

Being the first package to provide explicit support for agent based modelling, Swarm's characteristics provides a benchmark for subsequent frameworks. The most important feature of Swarm is its discrete event scheduler – this allows for agents to register their method calls to occur at specific times during the simulation. Frequently, but not always, these actions are registered to take place periodically, defining the model's *timestep*. An asynchronous simulation would simply consist of scheduling one event, which in turn causes further actions to happen.

1.3.6 Random number library

Most agent based simulations rely upon streams of random numbers. Unfortunately real sequences have notoriously low generation rates, and in any case are not reproducible, which is a problem if you want to study an effect that only occasionally makes its appearance. Usually, algorithmically generated, or *pseudorandom* number generators are used. However, any algorithmically generated sequence of numbers is correlated by definition, and this may or may not be a problem for the system being studied. Many evolving artificial life systems are know to exploit bugs unintentionally left by the programmer (see [37]), so it would not be surprising if evolving systems could exploit a weak random generator. The choice of random number generator can also have a significant effect over the result in Monte Carlo simulations[29]. It is therefore desirable for an ABM framework to provide a well stocked library of different random number implementations, and allow for different generators to be swapped in easily.

1.3.7 Swarm

Swarm[24] is very much the grand-daddy of agent based modelling frameworks. It was initiated by Chris Langton as a reaction to the many and various implementations of artificial life models, complete with "life-support systems" to handle I/O, initialisation and visualisation. The idea was to provide a software framework into which a scientist could plug just the computational representation of the model, rather than requiring the scientist to create all the necessary extra parts needed to support the computation. Just as we no longer expect scientists to grind lenses, or wire up their own custom built particle detectors, we shouldn't expect them to have to build the tools needed to analyse their models.

When Swarm was originally designed, C and Fortran were the predominant scientific programming languages. Neither of these languages provide explicit object oriented programming support, and Fortran in particular was only widely available as Fortran 77 (as f2c and later g77 compilers) which lacked many modern programming features (now rectified with the Fortran 90 programming language). The GNU C compiler (gcc) supported two important object oriented extensions to C: Objective C and C++. Objective C had the advantage of being relatively simple for programmers to learn the object oriented syntax, and moreover had inbuilt support for reflection (see Sect. 1.3.3) which C++ does not (*EcoLab* uses an additional C++ language processor to implement the necessary reflection functionality). So the choice of Objective C as an implementation language for Swarm is obvious.

At the time Swarm was developed, there was only one cross-platform GUI technology in the form of TCL/Tk (see Sect. 1.3.4). So this was adopted for the visualisation components of Swarm. BLT, an addon package for TCL/Tk containing implementations of plotting widgets was a particularly useful component. For similar reasons, TCL/Tk was adopted by *EcoLab*, whose development also started around the same time. However, there was one key design decision made by Swarm developers that differs from *EcoLab*. In Swarm, the TCL components are wrapped by Objective C classes so users of Swarm do not see the TCL interface. Swarm does not provide a scripting interface for the user – users need to provide their own. By contrast, *EcoLab* makes a feature of the TCL interface – users are expected to write, or adapt existing, TCL scripts to reflect the requirements of their experiment.

To implement a Swarm model, one needs to implement three separate Objective C components called "swarms". The ModelSwarm, which implements the computational model under study, the ObserverSwarm, in which the experimenter must specify all the tools and visualisation widgets to be used for interactive model exploration, and BatchSwarm, for doing extensive model surveys such as data collection or parametric surveys.

A Java interface to Swarm was developed, which allowed the Swarm library to be accessed from Java, and also Java implemented agents to be executed by a callback mechanism. Performance tends to be lacking compared with native Objective C model implementations, and more recent pure Java-based packages such as Repast or Mason have made Java-Swarm somewhat obsolete.

Also, an experimental DCOM interface to Swarm was tried by Daniels[7], which allowed Swarm to be used by any language supporting the DCOM interface[8], but this version of Swarm was never integrated into the production version.

Swarm's most distinctive feature is its discrete event scheduler (Sect. 1.3.5), a feature that has been copied by Repast and Mason. It also blazed the way with dynamic object probes and plotting and histogramming widgets

[8] DCOM is a Microsoft specific remote procedure call mechanism. Open source equivalents to it exist, such as Mozilla's XPCOM, but these have largely fallen out of favour in recent years in favour of *web services*.

derived from the BLT library. The other main feature is the *Space* library, which implements a 2D grid in which agents can act.

A random number library is provided that provides the usual range of uniform generators such as linear congruential and Mersenne twister, and a number of nonuniform distributions such as the normal distribution, gamma distribution and arbitrary user specified distributions.

Swarm also provides a containers library – lists, maps, sets and so on. This is not needed in packages based on Java or C++, as containers are part of the latter languages' standard library.

Swarm has extensive documentation, as well as numerous well developed pedagogical exercises.

1.3.8 Repast

Repast[25] is a more recent Agent Based Modelling framework, heavily inspired by Swarm. It comes in three different flavours: RepastJ, which is a pure Java based platform; Repast.Net, which is implemented in C# using Microsoft's .Net environment, and RepastPy, a rapid application development environment based on Python and Java. Both RepastJ and RepastPy run in a Java Virtual Machine (JVM), whereas Repast.Net runs in a .Net virtual machine . It is unclear whether Repast.Net can be used in the Mono environment – North et al. note the existence of Mono, but also say they expect the vast majority of .Net code to only be run on the MSWindows operating system.

RepastPy is meant as a reduced learning curve environment situated somewhere between NetLogo and Repast in functionality. Python is an object oriented scripting language that has received a lot interest in the last few years for coding scientific applications. RepastPy uses the JPython interpreter, which implements a Python interpreter on JVM with access to the underlying Java class libraries loaded into the JVM. RepastPy, in fact, makes considerable reuse of the RepastJ class library.

RepastJ is perhaps the most popular agent based modelling environment in use today. This is in no small part because of the popularity of the Java programming language, but also because it is a pure Java platform (so less complex to use than Java Swarm), and also because it has a few years head start on Mason, therefore has more comprehensive documentation, and also a larger community of users.

Repast comes with the following functionality: discrete event scheduler; a GUI controller which handles probing and interactively setting model variables, stepping and running the model; a parameter package for specifying model parameters in batch mode and/or managing parametric studies; an analysis package with plotting and histogramming, as well as some basic statistical functionality; and domain specific packages to handle 2D spatial grids, genetic algorithms, neural networks, support for Geographical Information Systems databases and some support for network modelling (classes for representing nodes and edges of a network).

Repast is distributed with the Colt numerical library, which includes an impressive array of random number generators .

The documentation consists a series of "how-to"s, relatively informal documents describing how to do one or two specific things. There are a number of example models which are good as starting templates for a new user. It is relatively simple to get the example models running in a GUI interactive mode, but not so easy to find out how to run models in a batch setting. There is a "-b" option that can be passed on the command line to disable the overhead of the GUI simulation controller, however any visualisation built into the model will continue to display unless the model has been coded with an explicit parameter that disables graphical output. There is no explicit support for model checkpointing, but since Java natively supports serialisation, a competent Java programmer should be able to add this functionality.

1.3.9 Mason

Like Repast, Mason is a 100% Java simulation platform that provides the usual array discrete event engines, probes and plotting widgets[21]. Its claimed strength lies in support from the outset for large scale modelling, with more optimised data structures, support for checkpointing and running of multiple batch runs. In my timing experiments (Sect. 1.4), Mason outperformed Repast, which at least backs up that claim. It also has extensive support for 3D calculations, something that is a little weak in Repast.

However, documentation is a weakness with Mason. It is not immediately obvious how one performs batch experiments, for example. Presumably one has to make specific allowances for this when coding the model, just as in Swarm and Repast. Mason is also a newer platform than Repast, hence it hasn't attracted as large a community of users as Repast.

1.3.10 EcoLab

EcoLab originally started life as a special purpose framework for hosting a single model written in C++, the Ecolab model[33]. Over the years it accreted several other similar types of models until by version 4 it had the ability to host an arbitrary C++ coded model[35]. The key feature needed for this was the Classdesc preprocessor, which effectively adds *reflection* to C++[22]. This allows the *EcoLab* framework to supply probing, scripting, checkpointing and even remote visualisation of running simulations.

Whilst *EcoLab* has been around for while, is reasonably mature software, and reasonably well documented, it has only been used by a small handful of groups. The example models provided with the source code are actually research models, so are not necessarily ideal for learning the system. One of these models is a continuous space agent based model, which illustrates a number of important ABM techniques.

The lack of pedagogical models is currently being addressed by implementing the Stupid Model [9] of Railsback et al. [30] in *EcoLab*. The Stupid Model has already been implemented in Swarm, Repast, Mason and NetLogo, so this is an ideal way of comparing the different environments. The implementation seemed to be about as easy as using Repast, and perhaps a little easier than Swarm. However, it lacks a special purpose spatial library – what it does have is something far more powerful (which also means more complex to use) called *Graphcode*[36]. Graphcode represents a network of agents (which could be cells of a spatial grid for instance), where the agents can be distributed across a cluster of computers enabling parallel processing. This allows for scaling agent based models to very large sizes. The jellyfish model provided in the examples has been run with several million jellyfish agents on 4–8 processor clusters.

One important aspect of agent based modelling is the use of references. When attaching a probe to an agent, the probe object needs to maintain a reference to its agent. When setting up schedules, lists of references to agents need to be maintained. C++ provides the notion of a static reference (reference initialised at construction), and pointers, but the former is too inflexible, and the latter too easily invalidated. *EcoLab* provides an experimental dynamic reference counted reference class that ensures the target object is destroyed once all references to it are. This problem is a nonissue in garbage collected languages like Java. Whilst garbage collection receives its share of opprobrium, for scientific modelling its performance impact is restricted to the interactive uses, when model performance is typically less important.

Unlike Swarm, Repast and Mason, *EcoLab* provides scripting interface. Your C++ model object is linked to a TCL interpreter[26], with the instance variables of your model available as TCL commands. Setting model parameters are simple TCL commands. Complicated initialisations can be computed – eg setting the random number seed to a function of the processor ID for instance to ensure independent random streams. The difference between batch processing and interactive processing is the presence of the 'GUI' command, and the presence graphical visualiser commands such as plot or histogram. The net effect is a sort of halfway house between a rapid application development environment, and a fully compiled application, allowing a great deal of flexibility during the experimentation phase.

EcoLab does assume competency with C++, but even though the user needs to program in TCL, not much knowledge of TCL is needed to do most experimental tasks. Sample scripts can be readily adapted by novice TCL programmers. Advanced TCL knowledge is really only needed for novel visualisation tools using the Tk canvas widget for instance.

Whilst *EcoLab* comes with a very elementary random number library, it is interfaced to use the far more comprehensive UNURAN[16] or the GNU Scientific Library[13] random number libraries. With UNURAN in particular,

[9] http://www.swarm.org/wiki/Software_templates

the random generators can be configured by a scripting interface, and this scripting interface is exported to *EcoLab*'s TCL interface.

1.3.11 The Logos, StarLogo and NetLogo

StarLogo[10][6] and NetLogo[11] use the *Logo* language, which was designed as an elementary teaching language for primary school students. The frameworks are simple and easy to use, so are recommended for users with little or no programming experience. However, the environments are often considered too simple for realistic research models. That said, Railsback et al. noted that NetLogo was sufficiently rich for them to be able to implement their pedagogical *Stupid Model* and that these environments should not be discounted completely for scientific research applications[30]. Of the three logo environments, NetLogo is the richest.

Both StarLogo and NetLogo are available for the Java virtual machine, and StarLogo has recently been released as a Java open source code version called OpenStarLogo. NetLogo is not open source. In reviewing the logos for this article, I was unable to build OpenStarLogo (on Linux), but both of compiled logos (Star- & Net-) had functional shell scripts for starting the simulator from the unix command line. Nevertheless, Logo is an interpreted code, with the interpreter running inside Java's virtual machine, so the modelling environments will be constrained in terms of performance.

1.3.12 Cormas

Cormas[5] is an agent based modelling platform written in Smalltalk that is mature, and has been used to implement a reasonable number of different models. The Smalltalk code comprising Cormas is available through an open source license, requiring registration with the Cormas development team. A variety of open source and freeware Smalltalk compilers are available, which typically compile to a bytecode interpreted representation. The Cormas website recommends the use of the Visual Works Smalltalk compiler from Cincom, which is available for MSWindows, MacOSX and Linux. It is unclear whether Cormas is ANSI Smalltalk standards compliant, or requires specific features of the Visual Works compiler.

To get started with Cormas, requires downloading a hefty amount of software – the Visualworks environment ISO image is around 600MB. However, once downloaded, the installation of Visualworks, and then Cormas on top of that on my Linux workstation was straightforward.

[10] http://education.mit.edu/starlogo/
[11] http://ccl.northwestern.edu/netlogo

1.4 Performance Comparisons

Rarely have different agent based modelling platforms been compared for performance, or ease of use, since reimplementing an existing model is a lot of effort, and people rarely have the cross-platform skills needed to do the task.

However, Railsback et al.[30] recently performed a cross platform study of Swarm, Java Swarm, Repast, Mason and NetLogo using a simple pedagogic model ("Stupid Model") that is in some way representative of typical agent based models. They structured their model in the form of a sequence of incremental steps that starts with implementing a number of agents moving around a featureless landscape at random up to a model with predator-prey interactions, and a renewable resource ("grass") that was replenished at different rates at different locations.

The present author has added to this study by implementing Stupid Model in *EcoLab*. The aim of this exercise was to show how *EcoLab* could be used for implementing the sorts of models one would use Repast and other similar ABM platforms, to gauge how difficult the task was from a programmer perspective and to compare simulation performance.

In order to be as comparable with Railsback et al.'s exercise as possible, the current public *EcoLab* release (4.D21) was used for implementing the models from the Stupid Model specification file. For ease of use, my experience was similar to that reported by Railsback et al., in getting the first model working within about 4 hours, and each model after that being a much smaller increment. The first model took as long as it did as the best way to represent a rectilinear space grid within Graphcode had not been determined. Aside from a specialised space library, no other needed feature was obviously missing. Versions 10 and 11 were performed in batch mode (no graphical output, no GUI control, Mason excepted), version 16 in GUI mode with a plot and histogram. *EcoLab*'s field version uses raster rather than canvas for display, and omits the expensive histogram widget.

The stopping criteria as specified by Railsback et al. is when the maximum bug size reaches 100. Since bug growth depends on the availability of food, which itself is a function of a random number generator call, and also of the grazing history, this stopping criterion is indeterministic, since the different frameworks will perform object updates in different orderings, and hence draw different sequences of random numbers. For the purposes of inter-framework performance comparisons, the stopping criterion was changed to be a fixed number of bug updates (500).

In version 10 of Stupid Model, bugs will randomly select a cell within their neighbourhood, and moving to it if the cell is empty, otherwise repeating the selection process. In version 11, all cells in the neighbourhood are iterated over, and the bug moves to the empty cell with the most food.

From version 12, bugs can reproduce and die according to random dynamics, so the amount of work per update step will depend on the number of living bugs. Even though these higher version models are more computationally intensive, run times cannot be compared between different platforms due to differences in the order that random numbers are generated. Hence the Stupid 16 measurements reported in table 1.1 should be taken with a certain amount of salt. Nevertheless, all models executed for 1000 steps without terminating early, and that the number of Stupid Bugs was roughly the same for each platform (approximately 8-900 after the initial population explosion).

Railsback et al. made no attempt to optimise model speed, so for comparison nor was the *EcoLab* model optimised. *EcoLab* is the only environment that explicitly supports a batch processing mode, and Railsback et al did not provide batch versions of their Stupid Model. Railsback et al's model code was modified to disable graphical updates, and CPU times used for comparison which eliminates any delay effect from having to launch the run manually with a mouse click. In comparing *EcoLab* with Repast, Mason and Objective-C Swarm, *EcoLab* was the fastest, with Mason and Repast not too much slower, but Swarm was substantially slower. Furthermore, in GUI mode, *EcoLab* was very slow, particularly compared with the Java platforms.

All performance benchmarks were run on a 2GHz Intel Pentium M processor with 1GB memory running Slackware Linux 10.0. The Java version used for Repast and Mason was SDK 1.4.2 standard edition. The compiler used for Swarm and *EcoLab* was GCC 3.4.3. I also did a comparison *EcoLab* run using the Intel C++ compiler 9.0, but this was more than 50% slower than the GCC compiled code. This somewhat surprising result indicates that icc's strength lies in vectorising loops that access data contiguously to exploit the inbuilt SSE instructions, but that for more general purpose ABM code, GCC performs better (at least on Linux!).

The sourcecode for *EcoLab* Stupid Model is available from the *EcoLab* website.[8]

Table 1.1. Execution CPU times (in seconds) for several Stupid Model versions for different platforms

Version	Repast	Mason	Obj-C Swarm	*EcoLab*
10	3.5	3.4	71	3.9
11	32.7	21.3	165	14.9
16	44	40.5	402	1014
field				67

My observations that the Java platforms performed almost as well as *EcoLab*'s C++-based one is broadly in line with other observations that Java implementations tend to be within a factor of 2 of natively compiled applications[4, 18]. The fact that Java code is obtaining comparable performance with native compiled object code indicates that just in time compilation technology has reached a comparable level of maturity compared with native code compilers. The stereotype of virtual machines having poor performance compared with native object code can be laid to rest, at least for the typically integer bound computations often seen in agent based models. Models requiring linear algebra operations will no doubt continue to perform better with C++ implemented code. Conversely, the poor performance of the GUI mode EcoLab is due to the graphical operations being implemented with the TCL library, which uses byte code interpretation. There will probably be some substantial wins in integrating EcoLab's C++ technology with the Repast or Mason execution engines.

1.5 Conclusion

Agent based modelling frameworks have matured a lot since Swarm was first released in 1995. Frameworks supply much of the necessary model independent functionality needed to get a scientific code running, and assist in exploring and debugging the model. Using a framework frees the scientific programmer to spend more time implementing the actual model.

The most important factor discriminating the frameworks reviewed here is the agent implementation language. Usually programmers have more experience in one language more than another, narrowing the choice to environments supporting the language with which they're familiar. If your language is C++, then EcoLab is a good choice, if Java then Repast (although Mason has some interesting additional features), if C then Objective C swarm, and if you're a novice programmer, one of the Logos.

The Java frameworks (Repast, Mason and Java Swarm) are by far the most popular, owing to the popularity of the Java language. Whilst earlier versions of the Java virtual machine exhibited performance problems, the most recent versions implementing *just in time compilation* can get close to the performance of a well optimised C++ application.

There are very few comparative studies comparing different ABM platforms, and it can be difficult to validly compare different platforms. Programmer familiarity with one programming environment will bias ease of use comparisons, and indeterminancy (due to different pseudo random number generators employed) will prohibit valid performance metrics.

The Stupid Model exercise, however, at least indicates the suitability of all the surveyed ABM environments for typical ABM requirements.

Most of the platforms have a range of standard and pedagogical agent based models implemented, some of which are described in earlier sections. EcoLab is the odd one out, in only supplying certain research models. There is a need for EcoLab versions of the standard models to assist newcomers in building their own models, and to assist in cross platform comparisons.

References

1. Chris Adami. *Introduction to Artificial Life.* Springer, 1998.
2. W. B. Arthur, J. H. Holland, B. LeBaron, R. Palmer, and P. Tayler. Asset pricing under endogenous expectations in an artificial stock market. In *The Economy as an Evolving, Complex System II.* Addison-Wesley, Menlo Park, 1997.
3. Mark A. Bedau. Downward causation and the autonomy of weak emergence. *Principia*, 6:5–50, 2002.
4. RF Boisvert, J. Moreira, M. Philippsen, and R. Pozo. Java and numerical computing. *Computing in Science & Engineering [see also IEEE Computational Science and Engineering]*, 3(2):18–24, 2001.
5. F. Bousquet, I. Bakam, H. Proton, and C. Le Page. Cormas: common-pool resources and multi-agent systems. In Angel Pasqual del Pobil, Jos Mira, and Moonis Ali, editors, *Tasks and Methods in Applied Artificial Intelligence*, volume 1416 of *Lecture Notes in Artificial Intelligence*, pages 826–838. Springer, 1998.
6. Vanessa Stevens Colella, Eric Klopfer, and Mitchel Resnick. *Adventures in Modeling.* Teachers College Press, 2001.
7. Marcus Daniels. Swarm and COM. In *Integration Workshop at the GIS/EM4 Conference*, September 2000.
8. EcoLab website. http://ecolab.sourceforge.net.
9. Joshua M. Epstein and Robert L. Axtell. *Growing Artificial Societies: Social Science From the Bottom Up.* MIT Press, 1996.
10. Free Software Foundation. The free software definition. http://www.fsf.org/licensing/essays/free-sw.html.
11. Stan Franklin and Art Graesser. Is it an agent, or just a program?: A taxonomy for autonomous agents. In Jörg P. Müller, Michael J. Wooldridge, and Nicholas R. Jennings, editors, *Intelligent Agents III Agent Theories, Architectures, and Languages*, volume 1193 of *LNCS*, pages 21–35. Springer, 1997.
12. Jochen Fromm. *The Emergence of Complexity.* Kassel UP, Kassel, 2004.
13. M. Galassi, J. Davies, J. Theiler, B. Gough, G. Jungman, M. Booth, and F. Rossi. *GNU Scientific Library Reference Manual - Revised Second Edition.* Network Theory Ltd, 2005.
14. Volker Grimm. Ten years of individual based modelling in ecology: What have we learned and what could we learn in the future? *Ecological Modelling*, pages 129–148, 1999.
15. John Holland. *Emergence: From Chaos to Order.* Addison Wesley, 1997.
16. W. Hörmann, J. Leydold, and G. Derflinger. *Automatic Nonuniform Random Variate Generation.* Series: Statistics and Computing. Springer, 2004.
17. M. Huston, E. DeAngelis, and W. Post. New computer models unify ecological theory. *Bioscience*, 38(1):682–691, 1988.
18. J.P.Lewis and Ulrich Neumann. Performance of Java versus C++. http://www.idiom.com/ zilla/Computer/javaCbenchmark.html, 2003.
19. Willaim Kahan and Joseph D. Darcy. How java's floating-point hurts everyone. http://www.cs.berkeley.edu/˜wkahan/JAVAhurt.pdf.
20. Christopher G. Langton. Artificial life. In C. Langton, editor, *Artificial Life*, page 1. Addison-Wesley, Reading, Mass., 1988.
21. Sean Luke, Claudio Cioffi-Revilla, Liviu Panait, Keith Sullivan, and Gabriel Balan. MASON: A multiagent simulation environment. *Simulation*, 81:517–527, 2005.

22. Duraid Madina and Russell K. Standish. A system for reflection in C++. In *Proceedings of AUUG2001: Always on and Everywhere*. Australian Unix Users Group, 2001.
23. Barry McMullin. Thirty years of computational autopoiesis: A review. *Artificial Life*, 10:277–295, 2004.
24. Nelson Minar, Roger Burkhart, Christopher G. Langton, and Manor Askenazi. The Swarm simulation system: A toolkit for building multi-agent simulations. Technical Report WP96-06-042, Santa Fe Institute, 1996. http://www.swarm.org.
25. M.J. North, N.T. Collier, and J.R. Vos. Experiences creating three implementations of the Repast agent modeling toolkit. *ACM Transactions on Modeling and Computer Simulation*, 16:1–25, 2006.
26. J. K. Ousterhout. *TCL and the Tk Toolkit*. Addison-Wesley, 1994.
27. John K. Ousterhout. Scripting: Higher-level programming for the 21st century. *IEEE Computer*, 31(3):23–30, 1998.
28. R. G. Palmer, W. B. Arthur, J.H. Holland, B. LeBaron, and P. Tayler. Artificial economic life: a simple model of a stock market. *Physica D*, 75:264–274, 1994.
29. Giorgio Parisi and Federico Rapuano. Effects of the random number generator on computer simulations. *Physics Letters B*, pages 301–302, 1985.
30. S. F. Railsback, S. L. Lytinen, and S. K. Jackson. Agent-based simulation platforms: Review and development recommendations. *Simulation*, 82:609–623, 2006.
31. H. D. Rathgeber. Mousetrap model of chain reactions. *American Journal of Physics*, 31:62, 1963.
32. Tom Ray. An approach to the synthesis of life. In C. G. Langton, C. Taylor, J. D. Farmer, and S. Rasmussen, editors, *Artificial Life II*, page 371. Addison-Wesley, Reading, Mass., 1991.
33. Russell K. Standish. Population models with random embryologies as a paradigm for evolution. *Complexity International*, 2, 1994.
34. Russell K. Standish. On complexity and emergence. *Complexity International*, 9, 2001. arXiv:nlin.AO/0101006.
35. Russell K. Standish and Richard Leow. EcoLab: Agent based modeling for C++ programmers. In *Proceedings SwarmFest 2003*, 2003. arXiv:cs.MA/0401026.
36. Russell K. Standish and Duraid Madina. ClassdescMP: Easy MPI programming in C++. In Sloot et al., editors, *Computational Science*, volume 2660 of *Lecture Notes in Computer Science*, page 896, Berlin, 2003. Springer.
37. Kurt Thearling and Tom Ray. Evolving multi-cellular artificial life. In R.A. Brooks and P. Maes, editors, *Artificial Life IV*, pages 283–288, Cambridge, Mass., 1994. MIT Press.
38. Andrew Wuensche. Discrete dynamical networks and their attractor basins. *Complexity International*, 6, 1999.

A ABM Platforms

Swarm

URL:	http://www.swarm.org
Model Language:	Objective C or Java
Visualisations:	Plotting, Histogram, Raster
Scripting:	None
Features:	2D Space, Event scheduler, Probes

Repast

URL:	http://repast.sourceforge.net
Model Language:	Java, Python and .Net (C#, etc.)
Visualisations:	Plotting, Histogram, Raster
Scripting:	Parameter files
Features:	2D Space, Event scheduler, Probes, GIS support

Cormas

URL:	http://cormas.cirad.fr
Model Language:	Smalltalk
Visualisations:	Raster, Vector graphics, Plot, Message
Scripting:	None
Features:	2D Space, Event scheduler, Probes, GIS support

Mason

URL:	http://cs.gmu.edu/~eclab/projects/mason
Model Language:	Java
Visualisations:	Plotting, Histogram, Raster
Scripting:	java.util.Properties (parameter files)
Features:	2D & 3D continuous, discrete or network Space, Event scheduler, Probes, Checkpointin

EcoLab

URL:	http://ecolab.sourceforge.net
Model Language:	C++
Visualisations:	Plotting, Histogram, Canvas
Scripting:	TCL
Features:	Network Space (Graphcode), Probes, Checkpointing, Parallel programming

NetLogo

URL:	http://ccl.northwestern.edu/netlogo
Model Language:	Logo
Visualisations:	Plotting, Histogram, Raster
Scripting:	None
Features:	2D & 3D Space, Probes

StarLogo

URL:	http://education.mit.edu/starlogo
Model Language:	Logo
Visualisations:	Plotting, Raster
Scripting:	None
Features:	Space, Probes

B Discussion Fora

The fora are not platform specific. For platform specific fora, such as asking for programming help or discussing bugs or software improvements go to the platform specific website.

swarm-modelling:
    http://www.swarm.org/wiki/Swarm:_Mailing_lists
Grey Thumb:
    http://www.greythumb.org/wiki/WikiHome
planet agents:
    http://planetagents.org
Agent-based Computational Economics:
    http://www.econ.iastate.edu/tesfatsi/ace.htm
SwarmFest:
    http://www.swarm.org/wiki/Swarm:_SwarmFest *Annual ABM conference*
complexity-science:
    http://necsi.net:8100/Lists/complex-science/List.html *General complex systems discussions, occasionally ABM related*
FRIAM
    http://www.friam.org/ *General complex systems discussions, occasionally ABM related*
http://www.swarm.org/wiki/Software_templates This page contains links to the StupidModel specifications, and implementations in various frameworks.

Index