Resources for Agent-Based Modeling

By Robert Axelrod

Published as Appendix B of Robert Axelrod, *The Complexity of Cooperation: Agent-Based Models of Competition and Collaboration* (Princeton, NJ, Princeton University Press, 1997).

This appendix offers resources to learn more about complexity theory and agent-based modeling in the social sciences. The three sections provide:

* advice on computer simulation techniques,

* exercises for learning how to do agent-based modeling, and

* a syllabus for a short course in agent-based modeling in the social sciences.

Associated with this volume is an Internet site.[1] The site provides links to the source code and documentation for most of the models in this book, as well as the exercises in this Appendix. The site also provides links to information about literature, people, organizations, software, and educational opportunities related to complexity, agent-based modeling, and cooperation.

Computer Simulation for Agent-Based Modeling

Programming Your Own Agent-Based Model

There are several programming environments specifically designed for agent-based models.

a. StarLogo is a programmable modeling environment for exploring the behaviors of decentralized systems, such as bird flocks, traffic jams, and ant colonies. It is designed especially for use by students. Resnick (1994) provides an excellent introduction. While not designed for serious research, StarLogo does provide an easy way to get started in agent-based modeling from someone who has never done any programming. StarLogo is available for Macintosh computers free of charge on the Internet. Another version of the Logo family that can run many agents at once is available commercially as Microworlds Project Builder. It comes with an excellent user interface, and is available in both PC and Macintosh versions.

b. For advanced programmers, the Swarm programming environment provides a very rich set of tools for agent-based modeling. It allows nested hierarchies of agents, full control of the scheduling of events, and probes to report the current state of the agents and their environment. The Swarm system is available free from a team at the Santa Fe Institute. The system runs in UNIX, and requires programming in Objective C. The aspiration of the designers is that the community of Swarm users will to be able to share each other's ever expanding set of simulation tools. For those familiar with UNIX and any object-oriented version of C, Swarm is a good choice because its powerful tools make it easy to implement a new model, and to analyze its dynamics.

Selecting A Programming Language

Because StarLogo is not designed for serious research, and Swarm requires considerable programming sophistication, many modelers prefer to work with the general purpose tools that come with the compiler of a standard programming language. This leads to the question of which programming language to use. My answer is that although many types of programming languages exist, a beginner should select a procedural language.[2] The most common procedural languages are Basic, FORTRAN, Pascal and C. These languages have different histories and characteristics.

1. Basic is designed for beginners, and is perhaps the simplest to learn and use. It is suitable for small projects, but usually runs slower than the other languages for projects involving large amounts of computation. The early versions of Basic were quite

rudimentary, but recent versions are a delight to use. For example, Visual Basic has tools for constructing good user interfaces, as well as tools for debugging. Visual Basic is available within some spreadsheet programs such as Excel. The integration of a programming language within a spreadsheet is a very convenient way to combine the full control of a programming language with the intuitive look and feel of a spreadsheet.

2. FORTRAN is an old language that is not as convenient to use as the others. Because of its age and prior popularity, many programmers are familiar with it, and many old programs are available in FORTRAN. A beginner, however, should definitely pick one of the more modern languages.

3. Pascal was designed to be a first language for serious programmers. It is easy to learn, and is structured to encourage good programming habits. Most of simulations in this volume were programmed in Pascal.

4. C is most common procedural language among serious programmers. It is designed to allow relatively easy conversion between one type of computer and other. It includes many shortcuts which a beginner need not learn. Unfortunately, the availability of these shortcuts can make understanding someone else's C code difficult. Besides the popularity and compatibility of C, another advantage is that it is the basis of the most popular object-oriented language, C++. An object-oriented language makes really large projects easier to program. It also makes it much easier to use portions of an old program in a new context. For all these reasons, C++ was chosen as the foundation for the Java programming language designed to be used over the World Wide Web.

Where do all these options leave the beginner? If you are a beginner to programming, my advice is to avoid FORTRAN, and pick one of the others based upon what help is readily available. What you will need most is someone who can help answer questions when you get stuck. If a friend or co-worker is available to answer questions about Pascal, for example, pick Pascal. Alternatively, if you want to take a programming course and there is one available on C, then that would be a good choice. If the availability of help or instruction does not lead to a clear choice, then you can make a decision based on how serious you plan to be about programming. If you just want to try it out to get a feel for doing your own programming or because you have a simple idea you want to test, then Visual Basic is a good choice. If you are fairly sure that you will be doing programming for some time and want to start with a language that you can grow with, then C is the best choice.

Goals of Good Agent-Based Programming

The programming of an agent-based model should achieve three goals: validity, usability, and extendibility.

The goal of validity is for the program to correctly implement the model. (Whether or not the model itself is an accurate representation of the real world is another kind of validity that is not considered here.) Achieving validity is harder than it might seem. The problem is knowing whether an unexpected result is a reflection of a mistake in the programming, or a surprising consequence of the model itself. For example, in the model of social influence presented Chapter 7, there were fewer stable regions in very large territories than there were in middle-sized territories. Careful analysis was required to confirm that this result was a consequence of the model, and not due to a bug in the program.[3] The goal of usability is to allow you and those who follow to run the program, interpret its output, and understand how it works. You may be changing what you want to achieve while you doing the programming. This means that you will generate whole series of programs, each version differing from the others in a variety of ways. Versions can differ, for example, in which data is produced, which parameters are adjustable, and even the rules governing agent behavior. Keeping track of all this is not trivial, especially when one tries to compare new results with output of an earlier version of the program to determine exactly what might account for the differences.

The goal of extendibility is to allow a future user to adapt the program for new uses. For example, after writing a paper using the model, the researcher might want to respond to a question about what would happen if a new feature were added. In addition, another researcher might want someday want to modify the program to try out a new variant of the model. A program is much more likely to be extendible if it is written and documented with this goal in mind.

Project Management

In order to be able to achieve the goals of validation, usability and extendibility, considerable care must be taken in entire research enterprise. This includes not just the programming, but also the documentation and data analysis. I often find myself wanting to get on with the programming as quickly as possible to see the output of the simulation. Whether I do the programming and documentation myself, or whether I use a research assistant, I tend to be too eager to see results. I have learned the hard way that haste does indeed make waste. Quick results are often unreliable results. Good habits slow things down at first, but speed things up in the long run. Good habits help by avoiding some costly mistakes and confusion that can take a great deal of effort to unravel.

If you are just beginning to do computer simulation, it pays build good habits. Your own needs will depend upon your programming experience and the demands of the project. The aim is to develop a set of habits that are effective, with a minimum of administrative overhead. Here is some recommendations based upon my own experience.

1. Use long names for almost all of the variables rather than short names that will be incomprehensible a month later. It is fine to use i and j, or x and y for a few really common variables. Beyond that, however, the clarity of long names are worth the extra typing, or cutting and pasting.

2. List all the variables at the start of the program. Some languages, such as Pascal, require you to declare all the variables and their type (e.g., integer or floating

point) before you use them. Other languages, such as Visual Basic and C, make explicit declaration of variables optional. Force yourself to be explicit about declaring variables in advance of their use. This will save you many hours of debugging by warning you that a misspelled variable name deep in the code is illegal, rather than letting the compiler make the false assumption that it is meant to be a new variable.

3. Write helpful comments. For example, when you declare a variable, write a comment about what the variable is intended to represent, and how it is distinguished from related variables. Likewise, write comments in the body of the code to describe what each subroutine does, and how it fits into the main program. It is a good idea to have as much text in comments as in code. Days or months later, these comments can be very useful.

4. Develop the sequence of programs so that they are upwardly compatible. This means that later versions should include all the useful features of old versions. Suppose, for example, that you start with a model that employs a 10x10 array of cells, each with exactly four neighbors. You can do this by having the map "wrap around," thereby making cells on the north and south edges into neighbors, and doing the same for cells on the east and west edges. Suppose that you later wanted to have a flat map with boundaries at the edges. You should not delete the portion of the code that deals with the neighborhoods on the original map. Instead, you should isolate that portion as a subroutine, and write another subroutine dealing with the neighborhoods the new map. Then add a control parameter to serve as a switch specifying whether a given run uses the wrap-around map or the flat map. Doing it this way gives you the option to return to a wrap-around map again later.

5. Fully label the output. The output should also include the time and date of the run, a sequential run number, the version number of the program, the settings of all the parameters that specify the nature of the run, and the random number seed if there is one. The output should allow you to replicate a run precisely. If the versions are upwardly

compatible, as suggested earlier, then you will be able to replicate a previous run using any later version of the program.

6. Practice defensive programming. Taking a few minutes to be cautious can save hours of searching for a mysterious bug. As already mentioned, it pays to explicitly declare all variables so that some typos can be caught early, and it pays to use long variable names so that a month later you do not get confused about what a mysterious "XKLT" might mean. Another defensive move is to declare the intended limits of your variables so that you can use the automatic debugging features of your compiler. For example, if the Agent_X_Location should be between 1 and Max_X_Coordinate, then be sure that your compiler knows this, or that you confirm it in your code. Finally, using a programming technique called pointers is dangerous. While pointers can be the most efficient way to program certain relationships, they should be used with great care because a mistake with a pointer can often cause bizarre symptoms that can be hard to isolate and repair.

7. Document each version of the code as you go. The documentation should include the exact specification of the model, a description of the algorithms used in the calculations, details about the inputs needed, and information about how to read the output. Explaining the output is particularly important because a column of data typically has such a brief label that you may not be sure what it means a month later. In many cases, the documentation of a new version can be as simple as the following. "Version 2.3 is the same as version 2.2 except an option is added to have a wrap-around map. This is implemented by setting Wrap_Around to True. If Wrap_Around is False then the map is flat as in previous versions." The documentation need not be elegant or concise. It does need to be complete and accurate. It pays to put some documentation into the program itself, but usually full documentation is more easily managed as a memo written with your word processor.

8. Use a commercial program to do most of the data analysis. Within your program do only simple data analysis, such as the calculations of averages, to reduce the amount of output that is required. Beyond that, it pays to use reliable software developed by others. For example, a good spreadsheet will allow you to manipulate the variables, do simple statistical analysis, and graph the data in different ways. An alternative to a spreadsheet is a program like Mathematica that includes not only statistical and graphics capability, but also a well-organized notebook structure to keep track of your work.

9. In validating the program, check the micro-dynamics, not just the aggregate results. Since agent-based models often have surprising results, you typically can not confirm the code by checking its output against known results, as you could with a prime number generator, for example. Instead you need to confirm that your program is correctly handling the details. For example, you should check how a given agent behaves in the various possible circumstances. This will require interim reports that are much more detailed than you will usually need for main data analysis.

10. If the modeler and the programmer are two different people, make sure the two of you understand each other at each step. It is surprisingly easy for two people to be confident they understand each other, and then find out much later that there was a subtle difference in what each thought they had agreed upon. Since it is not easy to validate an agent-based model, it is especially important that this sort of programming be based on thorough and accurate communication between the modeler and the programmer.

There is more to project management than good programming. You also need to develop systematic methods for archiving the output, analyzing the data, and interpreting the results. For example, it is a good idea to write memos to yourself as you go about your interpretation of particular runs. Like the documentation, these memos need not be concise or elegant, but they do need to be accurate. For example, suppose you are doing sets of runs to see what difference it makes whether the map is flat or wrap-around.

When you get the results you can write a brief memo comparing a set of runs done one way with a set of runs done the other way. The memo should include the identifying information from the output, such as the version number of the program, the parameter settings, and the run numbers. This can easily be done by opening the output file from your word processor, selecting the required information, and pasting it into the memo you are writing. Then you can write your interpretation of the data, including a graph or two copied from your analysis of the comparative data. A series of memos such as these can serve as a "lab notebook" recording your progress in understanding the implications of your modeling decisions. Eventually, the memos can provide the basis for the data analysis section of your final report.

Exercises

The best way to learn about agent-based modeling is to build and run some model. Here are a set of exercises that can help you get started. Each one can be done in a few days once you have mastered a programming language. These exercises are suitable for classroom use or independent study. The first three exercises can be done by relatively simple modifications of source code available on the Internet.[4]

Exercise 1. Schelling's tipping model. Thomas Schelling, who is best known for his work on deterrence theory, was also one of the pioneers in the field of agent-based modeling. He emphasized the value of starting with rules of behavior for individuals and using simulation to discover the implications for large-scale outcomes. He called this "micromotives and macrobehavior" (Schelling, 1978). One of his models demonstrates how even fairly tolerant people can behave in ways that can lead to quite segregated neighborhoods. This is his famous tipping model (Schelling, 1978, pp. 137-55). It is quite simple. The space is a checkerboard with 64 squares representing places where people can live. There are two types of actors, and they are represented by pennies and nickels. One can imagine the actors as White and Blacks. The coins are placed at random among the squares, with no more than one per square. The basic idea is that an actor will be content if more than one-third of its immediate neighbors are of the same type as itself. The immediate neighbors are the occupants of the adjacent squares. For example, if all the eight adjacent squares were occupied, then the actor is content if at least three of them are the same type itself as itself. If an actor is content, it stays put. If it is not content it moves. In Schelling's original model, it would move to one of the nearest squares where it would be content. For the purposes of this exercise, it is easier to use a variant of the model in which a discontented actor moves to one of the empty squares selected at random.

The exercise is to implement this model, and explore its behavior. In particular, test one of Schelling's speculations about his tipping model. His speculation was, "Perhaps ... if surfers mind the presence of swimmers less than swimmers mind the presence of surfers ... the surfers will enjoy a greater expanse of water (Schelling, 1978, p. 153)."

To make things concrete, assume there are 20 surfers and 20 swimmers.[5] Let the surfers be content if at least a third of their neighbors are surfers, and let the swimmers be content if at least half of <u>their</u> neighbors are swimmers. The actors can be numbered 1 to 40, and activated in the same order each cycle. Run the model for 50 cycles. Then for each of the 24 empty cells, write an A in the cell if more of its neighbors are surfers than swimmers, and write a B in the cell if more of its neighbors are swimmers than surfers. (It will probably be easier to do this step by hand than to automate it.) Then see if the surfers enjoy a greater expanse of water by seeing if there are more A's than B's. Do ten runs to get some idea of the distribution of results.

If you are a beginner, you might want to take advantage of source code and documentation I have written to implement this variant of Schelling's model. The source code is available both in Visual Basic and Pascal. Doing the exercise would then require making some simple modifications in the code to allow the two types of actors to have different requirements to be content.

Exercise 2. Schelling's Tipping Model, continued. The purpose of this exercise is to study some of the effects of having unequal numbers of the two types of actors. You should use the variant of Schelling's model described in the first exercise. The exercise is to answer these two questions and explain why things worked out as they did:

1. Do minorities get packed in tighter than majorities?
2. Does the process settle down faster when the numbers are unequal?

To make things concrete, use 30 Whites and 10 Blacks, both with the original (and equal) requirements for contentment. For density studies, use the A and B measures described for the first exercise.

Your should arrange to stop a run when no further change is possible. An easy way to do this is use periods of at least 40 events, and check whether there has been no movement in the current period since that would imply that everyone is content and no will ever move again. As in the first exercise, a beginner may choose to use the source code provided, and make the necessary changes.

Exercise 3. Extending the Social Influence Model. To gain practice in extending an existing model, here is an exercise based on the model of social influence described in Chapter 7. The model is simple enough that even adding an extension keeps it quite manageable. The purpose of this exercise is to implement a particular extension of the model to see what insights can be gleaned.

A dozen suggestions are briefly outlined in the section of Chapter 7 on "Extensions of the Model." Here are some more details about two of these possibilities.

a. Early Geographic Differences. The initial values of cultural features are assigned at random in the basic model. A wide variety of interesting experiments could be conducted by having some or all of the features given particular values. For example, suppose one wanted to study the effects of the seemingly universal phenomena that "things are different in the south." An easy way to do this would be to give one or two of the cultural features one value in the northern sites, and a different value in the southern sites. Would regions tend to form based on these small initial differences? If so would the eventual boundary between the regions closely correspond to the initial line between the north and the south? Another example of an interesting experiment would be to see how easy or hard it is for an initial advantage in numbers to take over the entire space. A simple way to study this is by giving a slight bias in the original assignment of values to features throughout the space, and seeing how much bias it takes to overcome the tendency to get swamped by random variations, and eventually to dominate the entire space.

b. Cultural Attractiveness. Some cultural features might be favored in the adoption process over others. For example, Arabic numbers are more likely to be adopted by people using Roman numerals than the other way around. This differential attractiveness of cultural features might be due to superior technology (as in this case of number systems), or it might be due to seemingly arbitrary preferences. One way to model this process would be to favor higher values of a given cultural feature over lower values. Presumably, features that are culturally attractive would tend to drive out less attractive features. Just how attractive does a feature have to be to dominate? Does the process of differential attractiveness lead to larger (and thus fewer) regions?

Incidentally, I do not have good answers to the questions raised by these two suggested extensions or any of the others mentioned in Chapter 7. These are open research topics.

Source code and documentation for the social influence model is available on the Internet in both Visual Basic and Pascal.[6]

<u>Exercise 4. Standing Ovation</u>. Consider the situation in which people are seated in an auditorium listening to a brilliant performance. At the end, the applause begins, and perhaps a standing ovation ensues. The exercise is to model the process of a standing

ovation. You should consider potentially interesting future directions for your model. You should also consider whether there are some economic or social scenarios that could be usefully modeled using such a process.[7]

Exercise 5. Human Waves. People are seated in a football stadium. Someone gets up and shouts, "Wave." A human wave can be formed if everyone follows rule A: stand up for one second if either (or both) immediate neighbors are standing.

To keep things simple, suppose there are only 20 people, and they are seated in a circle around the field. These people can be represented as being on a line, with the understanding that person 20 is adjacent to person 1 and as well as being adjacent to person 19.

Now suppose that people also follow rule B which says that regardless of rule A, once a person sits down, he or she stays seated for at least two seconds.

The first part of the exercise is to compare what will happen under rule A alone, and under the combination of rules A and B.

The second part of the exercise is to modify the A-B model to take into account some or all of the following observations:

Observation 1. A human wave actually moves in one rather than two directions.

Observation 2. A human wave can continue even if there is one person who never stands.

Observation 3. A human wave eventually dies out.

Observation 4. Some people need to stay seated for more than 2 seconds.

A Short Course in Agent-Based Modeling in the Social Sciences

If you want a structured way to learn more about applications of complexity theory in the social sciences, here is the outline of a short course that you can do on your own. It is designed for advanced undergraduate, or for graduate students. The only prerequisite is knowledge of some programming language if you want to do the exercises.

The fact that complexity theory is a new and rapidly evolving field has two important implications for how the course is designed. First, there is not yet a comprehensive textbook for complexity theory. Therefore, the best way to learn the field is to read mainly original research. Second, since the principal methodology of complexity theory in the social sciences is agent-based modeling, the best way to learn the field is to study a variety of specific models. Each of the reports on specific models contains not only ideas for how to do modeling, but also useful techniques for the analysis of simulation data, and examples of how to draw inferences from the performance of the models for the understanding of real social problems. Even if you are interested primarily in a single discipline you will find that readings on many of the topics contain ideas that can be used in your own field of interest.

The course considers a wide variety of applications of agent-based models to the social sciences, including elections, markets, residential segregation, social influence, war, alliances, nation formation, and organizational change. Among the issues examined across models are path dependence, sensitivity to initial conditions, emergence of self-organized structure, adaptation to a changing environment, and criteria for judging the value of an agent-based model.

The exercises described above can be done in tandem with the readings. The first three exercises can be done within the first four weeks. Either or both of the other two exercises can be done by the seventh week.

If you are an instructor interested in developing a semester-long course, I would suggest adding a major project to the readings and exercises. The project could be the development and analysis of an original agent-based model (perhaps one based in part on someone else's model), or the project could be a review and criticism of an application of complexity theory to a specific domain. The projects could done by individuals or small groups. The last few weeks of the class could be devoted mainly to student reports on their projects.

Additional suggested readings are available on the Internet.[8]

1. Introduction to Complexity Theory.

Chapter 1 of this volume.

Schelling, Thomas. 1978. Micromotives and Macrobehavior. New York: Norton, pp. 137-155.

2. Agent-Based Models.

Resnick, Mitchel, 1994. Turtles, Termites, and Traffic Jams: Explorations in Massively Parallel Microworlds. Cambridge, MA: MIT Press, p 3-19, on decentralization.

Epstein, Joshua M. and Robert Axtell, 1996. Growing Artificial Societies: Social Science from the Bottom Up. Cambridge, MA: MIT Press. Introduction.

Poundstone, William, 1985. The Recursive Universe: Cosmic Complexity and the Limits of Scientific Knowledge. Chicago: Contemporary Books, pp. 24-37, on the Game of Life. Programs to run the Game of Life are available on the Internet.

3. Social Influence.

Axelrod, Robert, 1997. "An Agent-Based Model of Social Influence: Local Convergence and Global Polarization." Included as Chapter 7 of this volume.

Axtell, Robert, Robert Axelrod, Joshua Epstein, and Michael D. Cohen. 1996. "Aligning Simulation Models: A Case Study and Results," Computational and Mathematical Organization Theory, 1, pp. 123-141. Included in this volume as Appendix 1.

4. Organization Theory.

March, James G. 1991. "Exploration and Exploitation in Organizational Learning," Organization Science, 2, pp. 71-87.

Simon, Herbert, 1982. The Sciences of the Artificial, second edition. Cambridge, MA: MIT Press, pp. 193-230.

5. Politics.

Kollman, Ken, John H. Miller, and Scott E. Page, 1992. "Adaptive Parties in Spatial Elections," American Political Science Review, 86, 929-37.

Cederman, Lars-Erik, forthcoming. Emergent Actors in World Politics: How States and Nations Develop and Dissolve.
Princeton, NJ: Princeton University Press. Chapters 8 and 9.

6. Genetic Algorithm and the Prisoner's Dilemma.

Holland, John H., 1992, "Genetic Algorithms," Scientific American, 267, July, 66-72.

Riolo, Rick, 1992. "Survival of the Fittest," Scientific American, 267, July, 114-6, on how to make your own genetic algorithm.

(If you are not familiar with the Prisoner's Dilemma, read Robert Axelrod, 1984. The Evolution of Cooperation. New York, Basic Books, pp. 3-69 and 158-68.)

Axelrod, Robert, 1987. "The Evolution of Strategies in the Iterated Prisoner's Dilemma," in Lawrence Davis (ed.), Genetic Algorithms and Simulated Annealing. London: Pitman, and Los Altos, CA: Morgan Kaufman, 32-41. Included in revised form as Chapter 1 of this volume.

7. Genetic Algorithm and the Prisoner's Dilemma, cont.

Lindgren, Kristian, 1991. "Evolutionary Phenomena in Simple Dynamics," in C. G. Langton et al. (eds.), Artificial Life II. Reading, MA: Addison-Wesley.

8. Economics.

Arthur, W. Brian. 1988. " Urban Systems and Historical Path Dependence," in Jesse H. Ausubel and Robert Herman (eds.), Cities and Their Vital Systems. Washington DC: National Academy Press, 85-97.

Arthur, W. Brian. 1993. "Why Do Things Become More Complex?", Scientific American, May, p. 144.

Epstein, Joshua M. and Robert Axtell, 1996. Growing Artificial Societies: Social Science from the Bottom Up. Cambridge, MA: MIT Press. Chapters II and IV.

Albin, Peter and Duncan K. Foley, 1992. "Decentralized, Dispersed Exchange Without and Auctioneer: A Simulation Study," Journal of Economic Behavior and Organization, 81, 27-51.

9. Concepts of Complexity.

Resnick, Mitchel, 1994. Turtles, Termites, and Traffic Jams: Explorations in Massively Parallel Microworlds. Cambridge, MA: MIT Press, pp. 129-144, on decentralized mind set.

Todd, Peter M., 1995. "Unsettling the Centralized Mindset," Adaptive Behavior, 3, 225-9. A useful review of Resnick's book.

Gell-Mann, Murray, 1995. "What is Complexity?" Complexity, 1, 16-19, on measuring the degree of complexity in a system.

Holland, John, 1995. Hidden Order. Reading, MA: Addison-Wesley, pp. 1-40, on elements of a complex adaptive system.

Bak, Per and Kan Chen, 1991. "Self-Organized Criticality," Scientific American, January, 46-53.

10. Adaptive Landscapes.

Axelrod, Robert and D. Scott Bennett, 1993. "A Landscape Theory of Aggregation," British Journal of Political Science, 23, 211-33. Included in this volume as Chapter 4.

Kauffman, Stuart, 1995. At Home in the Universe: The Search for Laws of Self-Organization and Complexity. New York and Oxford: Oxford University Press, pp. 252-271, on his patches model.

Coveney, Peter and Roger Highfield, 1995. Frontiers of Complexity: the Search for Order in a Chaotic World. New York: Fawcett Columbine, pp. 130-149, on neural nets.

References

Poundstone, William, 1985. The Recursive Universe: Cosmic Complexity and the Limits of Scientific Knowledge. Chicago: Contemporary Books.

Resnick, Mitchel, 1994. Turtles, Termites, and Traffic Jams: Explorations in Massively Parallel Microworlds. Cambridge, MA: MIT Press.

Schelling, Thomas. 1978. Micromotives and Macrobehavior. New York: Norton.

[1]http://pscs.physics.lsa.umich.edu/Software/ComplexCoop.html

[2]Among the alternatives to the common procedural languages are the following.

LISP is preferred by many researchers in artificial intelligence, and is especially good for handling data structures that change in the course of the execution of the program. Unfortunately, LISP is relatively difficult to learn, and typically runs much slower than compiled procedural languages.

Stella is designed for problems involving systems of difference or differential equations. Unfortunately, it is usually not easy to represent an agent-based model in this framework.

Gauss is designed for advanced statistical problems. It makes it easy to do matrix manipulation, for example. Unfortunately, most agent-based models can not be easily represented in ways that exploit Gauss's strengths.

[3]Indeed, one of the contributions of the alignment exercise of Appendix 1 was to confirm the validity of the original program.

[4]See the first footnote of this Appendix.

[5]Schelling made his speculation in the context of unequal numbers for the two types, but equal numbers are a good place to start.

[6]See the first footnote of this Appendix.

[7]This exercise is adapted from a homework problem given in the 1995 Santa Fe Institute Graduate Workshop in Computational Economics. It was placed on the Internet with some original answers at http://zia.hss.cmu.edu/econ/homework95.

[8]The Internet site associated with this volume includes links to a wide variety of additional source material on complexity theory. See the first footnote of this Appendix.