Available online at www.sciencedirect.com
SCIENCE DIRECT®
Environmental Modelling & Software 21 (2006) 283–309
Environmental Modelling & Software
www.elsevier.com/locate/envsoft

What every agent-based modeller should know about floating point arithmetic

J. Gary Polhill*, Luis R. Izquierdo, Nicholas M. Gotts

Macaulay Institute, Craigiebuckler, Aberdeen AB15 8QH, UK

Received 14 June 2004; received in revised form 10 August 2004; accepted 27 October 2004
Available online 22 January 2005

Abstract

Floating point arithmetic is a subject all too often ignored, yet, for agent-based models in particular, it has the potential to create misleading results, and even to influence emergent outcomes of the model. Using a simple demonstration model, this paper illustrates the problems that accumulated floating point errors can cause, and compares a number of techniques that might be used to address them. We show that inexact representation of parameter values, imprecision in calculation results, and differing implementations of mathematical expressions can significantly influence the behaviour of the model, and create issues for replicating results, though they do not necessarily do so. None of the techniques offer a failsafe approach that can be applied in any situation, though interval arithmetic is the most promising.

Keywords: Agent-based modelling; Emergence; Floating point arithmetic; Interval arithmetic

Software availability

Name: charity-1.1
Developers: Gary Polhill and Luis Izquierdo, Macaulay Institute, Craigiebuckler, Aberdeen, UK
Telephone, fax, and email: As above, plus l.izquierdo@macaulay.ac.uk, n.gotts@macaulay.ac.uk
Year first available: 2004
Hardware required: Intel PC or Sun Sparc, other platforms may work with Linux
Software required: Swarm 2.1.1 (PC, Windows 2K), Swarm 2.2 pretest 10 (PC, Windows XP) or Swarm 2001-12-18 (Sun, Solaris 8), fully implemented IEEE library functions
Program language: Objective-C
Availability: On-line at http://www.macaulay.ac.uk/fearlus/floating-point/charity-world/
Licence: GNU GPL
Cost: Free

1. Introduction

Agent-based modelling is a technique with growing popularity that has been applied to a diverse range of environmental applications. One of the classics is Lansing and Kremer's (1993) work on Balinese water temples, establishing a pedigree for the modelling of various water-related scenarios that has continued to the present day, as exemplified by authors such as Feuillette et al. (2003) and Pahl-Wostl (2005). At a more abstract level, agent-based modelling has also been applied to resource sharing (Rouchier et al., 2001) and common-pool resource dilemmas (e.g. Izquierdo et al., 2004; for a review see Gotts et al., 2003c; and see also CIRAD's CORMAS platform: Bousquet et al., 1998). It has been

* Corresponding author. Tel.: +44 1224 498200; fax: +44 1224 311557.
E-mail address: g.polhill@macaulay.ac.uk (J.G. Polhill).

1364-8152/$ - see front matter © 2004 Elsevier Ltd. All rights reserved.
doi:10.1016/j.envsoft.2004.10.011

used to investigate trapping strategies for cowbirds (*Molothrus ater*) (Harper et al., 2002) and to study forestry processes in Indiana (Hoffmann et al., 2002). Hare and Deadman (2004) and Bousquet and Le Page (2004) review various agent-based models in environmental and ecosystem management. The issue of numerics in such models has, however, not been covered in any great depth.

Floating point arithmetic is the standard way to represent and work with non-integer numbers in a digital computer. It is designed to create the illusion of working with real numbers in a machine that can only strictly work with a finite set of numbers. Some programming languages (FORTRAN, for example) use the `real` keyword to declare floating point variables. Thinking of floating point numbers as real numbers, however, is incorrect, and we show here that the differences have the potential to create seriously misleading results in research using agent-based models.

Rather than thinking of floating point numbers as reals, it is better to regard them as the product of an integer and an integer power of an integer base – a subset of rational numbers. For example, the normalised IEEE 754 single precision floating point numbers (IEEE, 1985) can be expressed as the product of a 24 bit integer and a power of two in the range −149 to +104 inclusive. Single precision IEEE 754 floating point numbers give between 6 and 9 significant decimal digits of accuracy, and double precision 15–17 significant decimal digits of accuracy (Sun Microsystems, 2001, p. 27).

One might argue that few models require as much as 15 significant figures of accuracy – after all, particularly in the case of agent-based models of social systems, it is unlikely that any measured parameter used as input to the model will have that level of accuracy. However, the agents in such models typically exhibit highly nonlinear behaviour, in that their course of action can depend on comparing variable values with precise thresholds. The behaviour of the model could depend on these values being calculated and compared accurately to ensure that the correct action is always selected.

While such sensitive dependence on correct calculations may be particularly important in agent-based modelling – because real-world decision-makers frequently choose between small sets of discrete alternatives with very different consequences – it may certainly occur in models of other kinds. An obvious example is mathematical models of chaotic systems, such as Lorenz's work with weather forecasting (Gleick, 1988, chapter 1), in which radically different behaviour was observed when starting a run half way through using parameters with three significant figures from a printout instead of the six stored in the computer's memory. This is a good analogy for what is happening in every calculation using floating point arithmetic.

Many programmers are already aware of some of the issues with floating point numbers, though as Fernandez et al. (2003) point out, "floating point representation still remains shrouded in mystery by the average computational science student, and only well understood by experts". Not testing for equality of floating point numbers is a commonly-taught "best practice" heuristic (Knuth, 1998, p. 233), and some compilers feature warning options to check for this (e.g. -Wfloat-equal in GNU's gcc (Stallman, 2001)). Where there are concerns about floating point accuracy, particularly in the use of comparison operators, another heuristic is to use small constants added to one or other of the terms to allow for the possibility of some loss of accuracy in floating point calculations. Neither of these heuristics, however, is sufficient to guarantee that an agent-based model using floating point numbers will be free from unintended effects arising purely from floating point arithmetic.

Edmonds and Hales (2003) have shown the potential for authors to draw the wrong conclusion from their models because of implementation-specific artefacts, and recommend that models be reimplemented to check results. They refer, however, to the complex and potentially ambiguous process of translating a model described in natural language into a computer program. Issues with floating point numbers occur at a different level. Even if a reimplemented model confirms the results of the original, both models could be reporting the wrong results because of accumulated errors arising from computations based on parameters that are not exactly representable using floating point numbers.

As will be shown below, many of the workarounds used in the field to try and cope with or avoid floating point errors are not safe, and thus may create a false sense of security. However, it will also be shown that the IEEE 754 standard stipulates facilities that, if properly used, do permit certainty that a model has not incurred floating point errors that could cause agents to act inappropriately. This places somewhat less of a burden on programmers than infinite accuracy – they need only familiarise themselves with the functions provided by any conforming platform to access these facilities. However, we downloaded 10 publicly available agent-based models[1] written in C or Objective-C, and though all of them included either single or double precision floating point variables, none included the header file

[1] Arborgames (ftp://ftp.swarm.org/pub/swarm/apps/objc/contrib/arborgames-2.1.141-Swarm2.1.141.tar.gz), Biofilm and Colony (http://www.theobio.uni-bonn.de/people/jan_kreft/bacsim.html), The Artificial Stock Market in Swarm (http://prdownloads.sourceforge.net/artstkmkt/), Echo (http://www.santafe.edu/projects/echo/), MAG (http://alife.tuke.sk/projekty/mag_html/mag_intro.html), SCL (http://www.eeng.dcu.ie/~mcmullin/), SLUCE (http://www.cscs.umich.edu/research/projects/sluce/), Tierra (http://www.isd.atr.co.jp/~ray/tierra/) and Village (http://www.santafe.edu/projects/swarm/users/Pages/Village/village.html).

ieeefp.h needed to access the functions. It seems, therefore, that programmers seldom make use of the error detection and rounding functions – reflected perhaps in the fact that both the SmallTalk platform on which SDML (Moss et al., 1998) is built and Java make no direct provision for these facilities (Gosling et al., 2000); the latter having been heavily criticised by floating point experts (Kahan and Darcy, 2001). For agent-based modellers this is bound to be contentious, since three popular agent-based modelling libraries, Ascape (Parker, 1998), Repast (Collier, 2000), and Mason (Luke et al., 2003) are Java-based, and Swarm (Langton et al., 1995), though it still has an Objective-C core, also provides a Java interface. That said, there are ways of detecting imprecision errors and implementing interval arithmetic indirectly in Java (Darcy, 2003, pers. comm.).

Whilst authors such as Fox (1971), as highlighted by Higham (2002, p. 31), estimate that "about 80 per cent of all the results printed from the computer are in error to a much greater extent than the user would believe", Winkler (2003) cites Lawson and Hanson (1995) as claiming that measurement errors are more significant than floating point rounding errors. Agent-based models, however, are sometimes sufficiently abstract in their design that their parameters and input data cannot meaningfully be supplied from real-world measured data. Examples include Epstein and Axtell's SugarScape (1996), the Artificial Stock Market (LeBaron et al., 1999; Johnson, 2002), and FEARLUS, an agent-based model of land use change (Polhill et al., 2001; Gotts et al., 2003b). We have shown separately how floating point errors affect the behaviour of the last two (Polhill et al., 2005). McCullough and Vinod (1999) in a review of numerical issues in econometric software, argue that since econometric data are known to only a few decimal places, it is certainly not worth reporting many decimal places in the *output* from a model, but intermediate calculations should still be done with as many digits as possible (p. 638).

The ensuing text shows by illustration that developers of agent-based modelling systems need to be aware of the issues arising from errors in floating point arithmetic, and compares some ideas for dealing with them. After a brief introduction to floating point arithmetic in Section 1.1, Section 2.1 introduces the simple model that is used to illustrate the potential for floating point arithmetic to have a significant impact on model behaviour. In Section 2.2, demonstrations will be given of how floating point arithmetic can cause problems through imprecise representation of parameters, imprecise calculations, and in replication or reimplementation of models from mathematical descriptions. Section 2.3 gives details of some of the techniques applied in the programming community or recommended by academics for addressing issues arising from floating point errors, and Section 2.4 shows, through particular parameterisations of the demonstrator model, how those techniques can fail. Section 2.5 describes the software used to conduct the comparison of the techniques with the demonstrator model. Section 3 gives the results of the comparison, and Section 4 comments on the results and the implications for programming agent-based models. Section 5 presents the conclusions.

1.1. What every agent-based modeller should not (ideally) have to know about floating point arithmetic

This section will briefly outline how floating point numbers are represented in a computer, and how even apparently simple calculations can introduce errors. The discussion focuses on floating point arithmetic conforming to the IEEE 754 standard (IEEE, 1985; Goldberg, 1991), to which most modern computers adhere, but will use 8-bit numbers for simplicity rather than the 32 (for single) and 64 (for double) stipulated by the standard.

In what follows, the following notation is used to distinguish operations using floating point arithmetic and mathematical operations as they are conventionally understood:

> $[\text{expr}]_f$ — The closest representable number to the result of expression expr in the current floating point environment.

In a computer, a floating point number consists of three parts: one sign bit s, x exponent bits e, and $p-1$ bits f used to represent the significand[2], with $x < p$ and where e and f are bit-strings. In IEEE 754 single precision, $x = 8$ and $p = 24$, and in double precision, $x = 11$ and $p = 53$. Here, we will use, say, $x = 3$ and $p = 5$. These bit-strings are (for the most part) used to construct a number of the form:

$$n = (-1)^s 2^E F$$

where n is the number, E is the exponent derived from e, and F the base 2 p-bit significand derived from f, with the binary point immediately after the first digit.

For the majority of numbers, the exponent bits e are translated into the actual exponent E using an offset b, where $E = e - b$. In single precision, $b = 127$, double $b = 1023$, and for the examples here, $b = 3$. The translation of the significand is based on the concept of *normalised* floating point numbers: those for which the first digit of F (which appears immediately before the binary point) is non-zero. Normalised numbers have the advantage that each number is represented by a unique

[2] The word 'significand' supersedes 'mantissa' to refer to this part of a floating point number (Goldberg, 1991).

value of E and F, and hence of their corresponding bit-strings e and f. In binary, if a digit is non-zero, then it must be 1, and hence there is no need to include it in the format. The leading 1 in F becomes a 'hidden bit' that is not stored in the bit-string f in the computer's memory. For example, in the 8-bit format, 0.75 would be stored as 0 010 1000: s is then 0, e is 010 (2 in decimal) and f is 1000, meaning $E = 2-3$, F is 1.1000 or 1.5 in decimal, and n is $+1 \times 2^{-1} \times 1.5$; whilst 1 110 1111 would represent $-1 \times 2^{(6-3)} \times (1 + 15/16) = -15.5$.

Using normalised numbers to represent F precludes the representation of zero. It would also be an advantage to allow denormalised numbers for very small fractions, giving a finer resolution around zero than possible with normalised numbers given the bit-width restrictions on the exponent. The IEEE 754 standard allows this: if all the e bits are 0, n is taken to be $(-1)^s\ 2^{(1-b)}\ 0.f$. To complicate things further, the IEEE 754 uses the case where all the e bits are 1 to allow the representation of non-numbers, specifically: positive and negative infinity, and a series of quantities called 'Not-a-Number' (shortened to 'NaN'). It also allows the distinction between positive and negative zero. These measures allow a program to perform calculations such as $-1/0$ and $\sqrt{(-1)}$ without crashing. The non-numerical aspects of floating point numbers will not be pursued further here.

The set of positive non-zero numbers that can be represented using the 8-bit IEEE 754-like floating point format is illustrated in Fig. 1. Although using only 8 bits exaggerates the sparseness with which floating point numbers cover the set of real numbers, the point is nevertheless made that even double precision numbers cannot and should not be treated as though they are real numbers. What is shared between the 8-bit implementation and the single and double precision IEEE 754 representations is that there is no exact representation of simple-looking base-10 numbers such as 0.1 (Wallis, 1990a). Indeed of the nine base-10 numbers 0.1, 0.2,…, 0.9, only 0.5 is exactly representable.

Clearly, since the coverage of the set of real numbers is limited, many arithmetic operations with exactly representable operands will result in numbers that do not belong to the set of numbers that can be represented using the floating point format. For example, in the 8-bit format, if the exponent is $+3$, the last digit of f represents

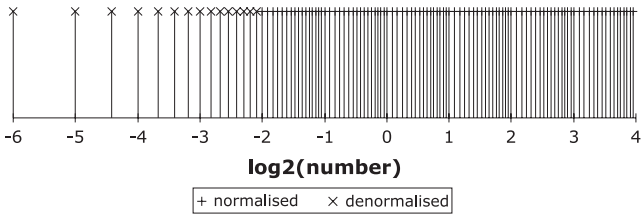

Fig. 1. The set of positive numbers representable using the illustrative 8-bit IEEE 754-like format shown on a logarithmic scale.

0.5, and hence, although 8.0 and 0.25 are both members of the set N_8 of numbers representable using this format, their sum is not. The IEEE standard stipulates that all arithmetic operations should be computed as though the precision is infinite and the result rounded to the nearest representable number. This is termed *exact rounding* (Goldberg, 1991). Where two such numbers are equally near, the option with 0 as the least significant bit is selected. In the 8-bit format, therefore, $[8 + 0.25]_f = 8$, since 8.5 is represented using 0 110 0001, and 8.0 using 0 110 0000. Fig. 2 shows the proportion of results for each arithmetic operator involving two members of N_8 with an exact result appearing in N_8. Unsurprisingly, numbers with more zeros in the least significant bits of the significand are, on the whole, more likely to be operands in exactly representable computations, and plus and minus are more likely to result in a member of N_8 than multiply or divide.

There are thus two potential sources of error in a computation of $x \otimes y = z$, where x, y and z are real numbers and $\otimes$ stands for any of the arithmetic operators: plus, minus, multiply, and divide. The first potential source of error lies in the conversion of x and y to their floating point representations $[x]_f$ and $[y]_f$. The second potential source of error lies in the conversion of the result of $[x]_f \otimes [y]_f = z'$ to its floating point representation $[z']_f = [[x]_f \otimes [y]_f]_f$. This is represented schematically in Fig. 3. The IEEE 754 standard stipulates that (in the default rounding mode) the result of the calculation should be the nearest representable floating point number to z', which is different from stipulating the nearest representable number to z. However, if $x = [x]_f$ and $y = [y]_f$, then $z = z'$, and $[z']_f$ will be the closest floating point number to z.

Thus far, we have been dealing only with a single floating point calculation, which could be just one part of an expression to compute a single value for an agent. The result of that part, $[z']_f$ then becomes an operand in another part of the expression, generating a new, possibly unrepresentable, result. Thus, even when 15 significant figures of accuracy are guaranteed for a single operation, that accuracy may not be carried forward to the result of evaluating an expression. In discrete-event models, if a parameter describing part of an agent's state at time $T-1$ is used as an argument to computing the state at time T, errors can accumulate over time. For example, the upper bound of the relative error of the sum of a series of numbers is proportional to the length of the series (Higham, 2002, p. 82). Hence, although 15 significant figures of accuracy may seem more than sufficient, it is sometimes not enough.

A further potential source of error lies in the hardware and software platform on which the calculation is taking place. Axelrod (1997) has reported problems with floating point arithmetic when reimplementing models on different platforms. The bug in the Intel Pentium

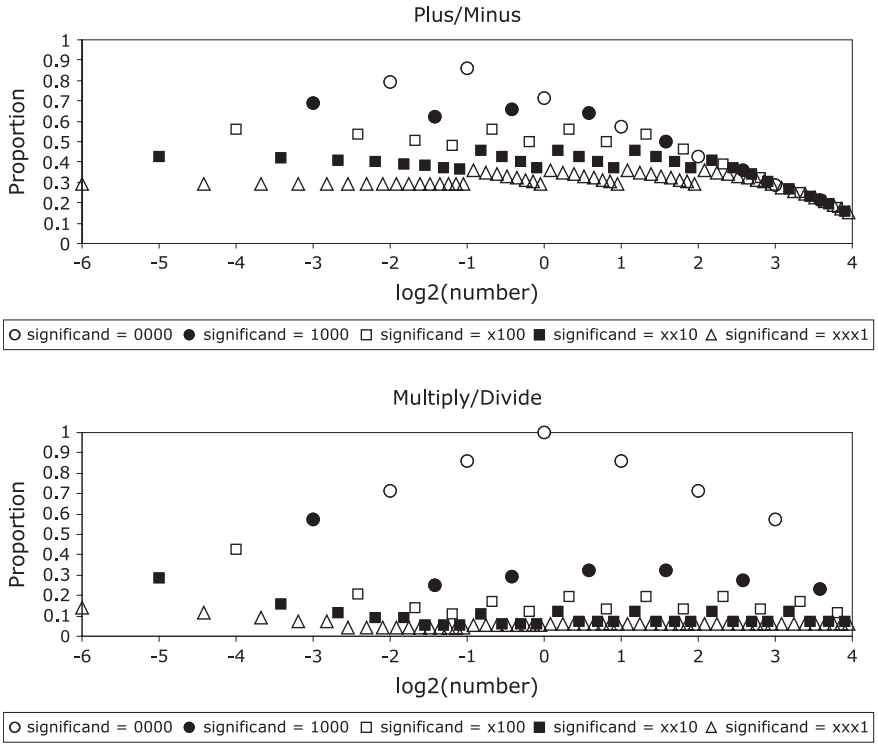

Fig. 2. Graphs showing, for the simple 8-bit floating point representation discussed in the text, the proportion of exactly computed results of $y \otimes number$, where *number* is a positive member of the set of numbers represented using the 8-bit implementation, y iterates over all members of this set, and $\otimes$ stands for any of the basic arithmetic operators. Note that the graphs for $+$ and $-$ are the same, as are those for $\times$ and /.

floating point processor is a well-known (Coe et al., 1995), but extreme, example of this. Another example is porting from IEEE 754 to non-IEEE 754 compliant platforms, which creates potential for the same program to deliver different results (Wallis, 1990b). However, it is also possible for two IEEE 754 compliant platforms to generate different output from the same program (Anon., 2001, p. 238). This is partly because the standard does not completely specify the accuracy of conversion between binary and decimal formats, and partly because of potential ambiguity in the precision of non-programmer-specified destinations for results of floating point operations, such as intermediate results in sub-expressions. Further differences can also be caused on the same platform through using different compiler optimisation settings.

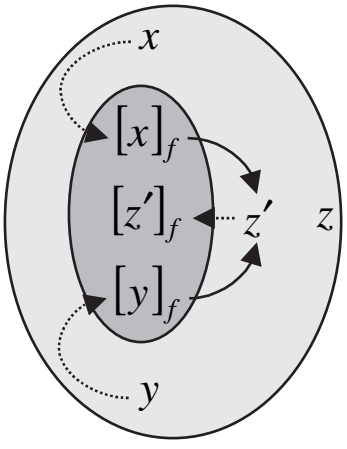

Fig. 3. Schematic representation of stages in a floating point calculation of two real operands x and y to the floating point result $[z']_f$. The area with a light grey background indicates real numbers not representable using the floating point implementation (possibly including the real result z), whilst the set with the dark grey background shows the subset of real numbers that are exactly representable. Dotted lines show conversion from a non-representable to a representable number, and hence a potential source of error, whilst the solid lines indicate the operation.

2. Method

The rest of this paper will focus on a simple agent-based model of wealth redistribution called CharityWorld, designed specifically for the purpose of illustrating problems with floating point numbers and comparing potential solutions. After introducing CharityWorld, illustrations of three issues with floating point arithmetic are provided, using particular parameterisations of the model:

(i) The effect of converting parameters from decimal to binary.

(ii) The effect of imprecision in calculations even when parameters are exactly representable.
(iii) The effect of rewriting expressions in ways that are equivalent in real arithmetic, but not (necessarily) in floating point.

These illustrations are used as a basis for comparing a number of techniques for avoiding problems with floating point arithmetic that exist either as folklore among programmers, or are recommended by academics studying floating point arithmetic. From an understanding of the techniques, it is possible to devise specific counter-example parameterisations of CharityWorld, designed to highlight any weaknesses. These counter-examples are tried with other techniques, to check whether there are corresponding issues. The comparison thus consists of the results of running CharityWorld using each technique on each illustration and counter-example parameterisation. The Software section (2.5) describes the software used to make this comparison possible.

2.1. Introduction to CharityWorld

CharityWorld is a model of wealth redistribution, in which a number of spatially embedded agents begin with a highly unequal distribution of wealth. This unequal distribution is then redistributed by the agents using rules determining when, to whom, and how much money is given. In this model, wealthy agents ('donors') donate money to their less wealthy neighbours without prompting. Two rules are used to control the exchange of wealth: *awareness*, which determines whether an agent will donate any money; and *approach*, which an agent uses to find an appropriate neighbour to donate to. The classes of agent used henceforth differ only in the awareness and approach conditions and the calculation of the amount exchanged. Fig. 4 shows the behaviour of the agents.

Agents in CharityWorld are distributed on a regular bounded rectangular grid of $x \times y$ square cells, with one agent per cell. A cell's neighbours are defined using the Moore neighbourhood which, for a cell not at the edge of the grid, consists of the eight cells sharing an edge or a corner with it. Agents are initially assigned a wealth w_0 (usually 0). To create an uneven distribution of wealth, the agents participate in a lottery with jackpot J, for which a single winner F is selected. Since each agent pays J/xy in wealth, F has a wealth of $w_0 + J(xy-1)/xy$ after the lottery and the other $xy-1$ agents have wealth w_0-J/xy.

Scheduling in the model is dynamic in that agents put themselves on a schedule-list of potential donors depending on whether their supply-awareness-condition applies. Immediately after the lottery has taken place, each agent is asked to determine whether it is to be put on the schedule-list. This completes the initialisation process.

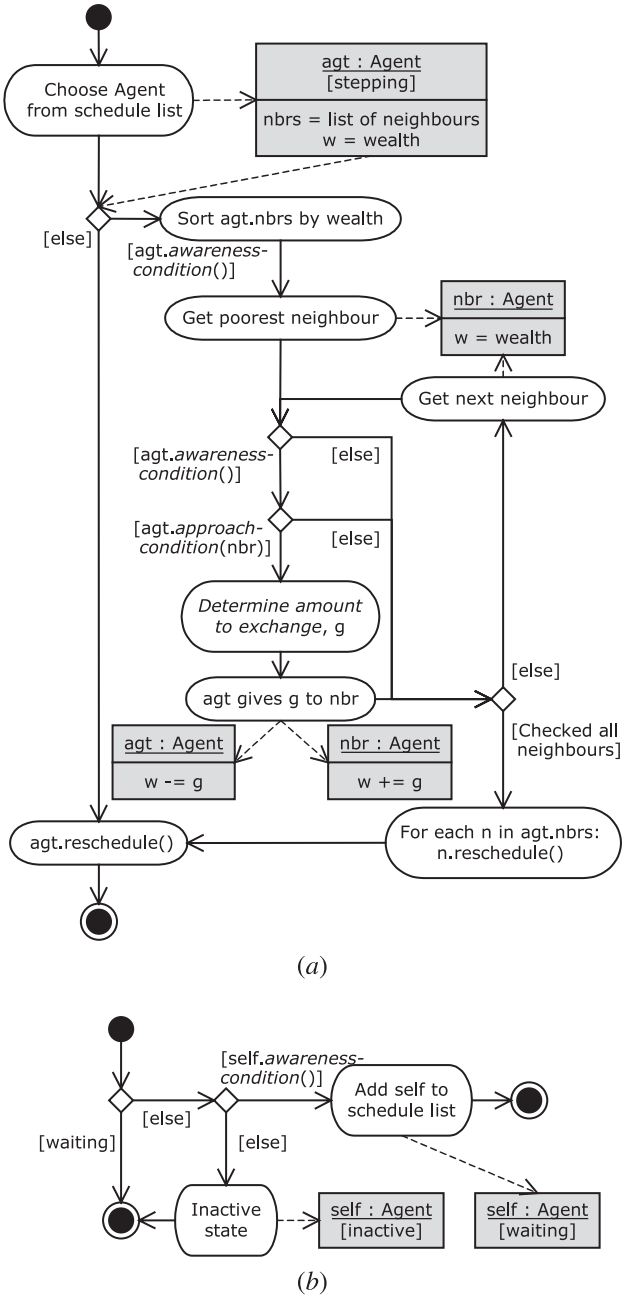

Fig. 4. UML Activity Diagrams (Booch et al., 1999) for agent behaviour in CharityWorld. Italicised activities are those that differ among Agent classes. (a) Donation process. (b) Process for rescheduling an agent.

After initialisation, an agent is selected from the schedule-list at random and asked to perform a cycle of redistribution (Fig. 4a). The selected agent considers all its neighbours in ascending order of wealth, and gives each a certain amount of wealth g if both the awareness-condition and the approach-condition are satisfied. After the wealth exchange is completed, each of the agents

involved in the cycle of redistribution is put onto the schedule-list if its supply-awareness-condition applies. The process of determining whether an agent should put itself on the schedule-list is termed *rescheduling* (Fig. 4b). An agent has three possible states: 'stepping' when it is in the process of choosing neighbours to give wealth to, 'waiting' when it is on the schedule-list, and 'inactive' otherwise.

The model then repeats the following step until the schedule-list is empty (i.e. all agents are inactive): if the schedule-list is non-empty, then select an agent from the schedule-list at random and ask it to perform one cycle of wealth redistribution (Fig. 4). When the schedule-list is empty, the model is said to have *terminated*. If, after termination, all agents have equal wealth, it has *converged*.

A CharityWorld model is therefore determined by the following parameters:

- The size of the environment, x and y.
- The size of the jackpot, J.
- The initial wealth, w_0.
- The class of agent to use (all agents in a model belong to the same class).
- The supply parameter, G. This is used by the agents to determine the amount g of wealth donated when the supply-approach-condition is satisfied.

For example, consider a class of agents that will be referred to as Simple-Supply-Agents (SSA). The supply-awareness-condition, for an SSA A with a neighbourhood consisting of the set $N(A)$ of SSAs is:

$$\text{SSA-supply-awareness-condition}\,(A) \Leftrightarrow (\#N(A)+1)w_A > \sum_{a\in N(A)\cup\{A\}} w_a \quad (1)$$

using $\#S$ to represent the cardinality of set S, and w_i to represent the wealth of SSA i. Similarly, the supply-approach-condition for A and one of its neighbours $n(A)$ is given in Eq. (2):

$$\text{SSA-supply-approach-condition}\,(A,n(A)) \Leftrightarrow (\#N(A)+1)w_{n(A)} < \sum_{a\in N(A)\cup\{A\}} w_a \quad (2)$$

Essentially, an SSA will donate if its wealth is greater than its local average wealth. Recipient SSAs have wealth less than the local average of the donating SSA. The amount exchanged, $g = G$. Simulations involving SSAs will eventually converge with all SSAs having wealth w_0 if $J/G = jxy$, where j is a positive integer. (A proof of this is in Appendix 1.) Thus for any G, there exists a jackpot for which convergence is guaranteed; in particular, if $J \equiv jxyG$ then convergence occurs for any $j \in \mathbf{N}$ and $G \in \mathbf{R}^+$.

2.2. Illustrations

2.2.1. Floating point representation of parameters

This section demonstrates how the binary floating point representation of decimal parameters of a model can affect the outcome in CharityWorld. Imprecision in parameters refers specifically to using parameters that do not have an exact representation in the floating point implementation used. As already mentioned, of the numbers $0.1i$ from $i = 1, 2, \ldots, 9$, only 0.5 is exactly representable. More generally, the only exactly representable base 10 floating point numbers are those representing irreducible fractions with a power of two denominator. Considering the numbers between 0 and 1 exclusive, n decimal places admit a power of two denominator equal to at most 2^n, since the highest power of two factor of 10^n is 2^n. There will then be $2^n - 1$ exactly representable numbers between 0 and 1 exclusive out of $10^n - 1$ base 10 floating point numbers with n significant figures. This fraction decreases with increasing n.

Single precision floating point numbers guarantee 6 significant decimal digits of precision, and thus can exactly represent $2^6 - 1 = 63$ of the 999999 base 10 floating point numbers $0.000001i$ ($i = 1, \ldots, 999999$) between 0 and 1 exclusive. It would be a mistake to think that double precision can exactly represent a greater proportion of these numbers, but it does achieve a reduction in the error of representation, expressed as the absolute difference between the base 10 number and the base 2 number used to represent it.

Imprecision in parameters has the greatest potential to introduce errors in the model, since the model is making calculations using inexact values from the start of the run. It is not necessarily the case, however, that imprecise parameters will lead to an incorrect result.

2.2.1.1. Illustration 1. Consider a model containing 9 SSAs on a 3×3 grid with initial wealth $w_0 = 0$. The agent in the centre location (2, 2) receives the jackpot $J = 9jG$, where j is a positive integer, meaning it has wealth $8jG$ (after paying the price for participating in the lottery) whilst the other eight agents have wealth $-jG$. Throughout the run, the agent at (2, 2) will be the only donor according to the SSA-supply-awareness-condition, and there should be j cycles to convergence in each of which it donates G of wealth to each of its neighbours. At convergence, each agent should have wealth 0. This result should be independent of the supply parameter G.

Implementing this model in a computer, however, convergence occurs when G is 0.5, for which the model behaves as predicted, but if G is 0.4, the model does not

converge for any j. For example, if $j = 1$ and G is 0.4, then the agent at (2, 2) receives

$$[[[[[[[[0.4]_f + [0.4]_f]_f + [0.4]_f]_f + [0.4]_f]_f + [0.4]_f]_f + [0.4]_f]_f + [0.4]_f]_f + [0.4]_f]_f$$

initially. This sum (using double precision) is $3.19\ldots_{14}\ldots 9733546474\ldots$ instead of 3.2, where $x\ldots_n\ldots x$ represents digit x repeated n times. The agent at (2, 2) then gives $[G]_f$ to each of its eight neighbours, and $[0.4]_f$ is subtracted eight times from the initial wealth sum, leaving it with a wealth of $0.0\ldots_{15}\ldots 01110223024625\ldots$, and the neighbours each with 0.0. The agent at (2, 2) now has greater than average wealth, and it thus donates $[0.4]_f$ to one of its neighbours, leaving it with $-0.39\ldots_{15}\ldots 911182158\ldots$ in wealth and the neighbour with $[0.4]_f$. As the agent at (2, 2) now has negative wealth, the neighbour will donate $[0.4]_f$ back, returning the state of play to what it was two donations previously (Fig. 5).

Although $0.5 = [0.5]_f$ and $0.4 \neq [0.4]_f$, the problem is not due simply to the fact that 0.4 is not exactly representable, but to the specific nature of its inexact representation and how this interacts with the requirements of the calculations in the model. When $j = 1$, the wealth of the winning agent in the 3×3 environment is the result of summing the ticket price ($[[J]_f/9]_f$) eight times. If the ticket price could be added eight times to the winning agent's initial wealth and then G subtracted eight times to yield the initial wealth again, then the model would converge. Whether or not this happens depends on whether $[G]_f = [[J]_f/9]_f$, and on what effect any floating point errors have on the series of additions and subtractions.

Using the 100 two-decimal-place base 10 numbers between 0.00 and 0.99 inclusive for G (with $J = 9G$), we found that 17 of them converge including the only four for which G and/or J is exactly representable, and for all 17, $[G]_f = [[J]_f/9]_f$ (the ticket price is equal to $[G]_f$). Of the 83 that do not converge, 29 cases have $[G]_f \neq [[J]_f/9]_f$, and all 83 involve rounding errors during the addition and subtraction. The 13 inexact values for G that converge are: 0.07, 0.13, 0.14, 0.26, 0.27. 0.28, 0.51, 0.52, 0.54, 0.55, 0.56, 0.71 and 0.79, of which 0.51 and 0.55 involve rounding errors during addition and subtraction. Only four of the 13 inexact values that converge with double precision also converge with single precision: 0.13, 0.26, 0.51 and 0.52. Ten other values converge with single precision that do not converge in double precision, meaning that there is one more convergence out of the 100 values for G in single precision than in double. From the above, we conclude that:

- Inexact representation of parameters does not necessarily preclude correct behaviour, but (in this case at least) makes it less likely.
- Rounding errors in calculations do not necessarily preclude correct behaviour, but (in this case at least) make it less likely.
- Using extra precision does not necessarily make correct behaviour more likely. Indeed, in this case double precision converged less often than single!
- Parameter values that work in one floating point precision will not necessarily work in another.

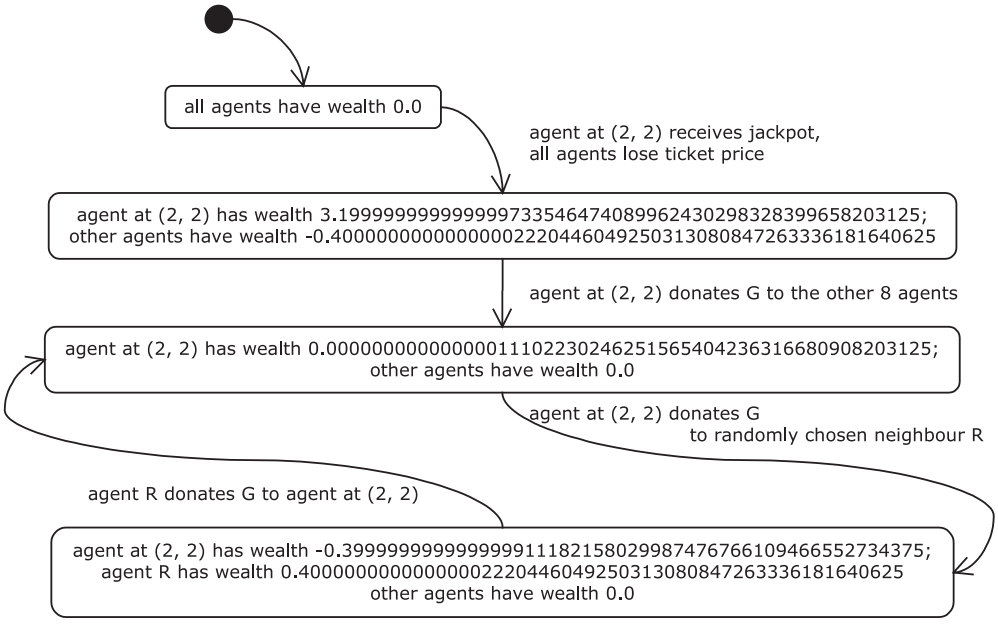

Fig. 5. UML Statechart Diagram (Booch et al., 1999) showing how a run of CharityWorld loops infinitely instead of terminating using SSAs on a 3×3 grid when $j = 1$ and $G = 0.4$ and the agent winning the jackpot is in location (2, 2) on the grid.

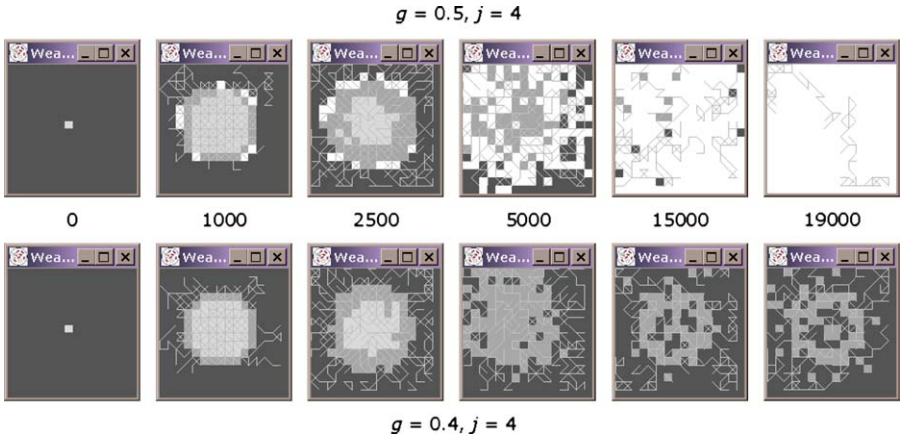

Fig. 6. Sequence of output from two runs of CharityWorld using the same seed that would show exactly the same output were it not for errors in floating point arithmetic.

The difference between runs using $G = 0.5$ and $G = 0.4$ for the same j, x and y is demonstrated in Fig. 6, using a larger environment than the 3×3 grid used above. Each snapshot in Fig. 6 colours each cell using four shades of grey according to the wealth of the agent in that cell. Dark grey cells indicate negative wealth. White cells indicate a wealth equal to zero. The lighter two shades of grey indicate positive wealth, the darker of the two shades indicating wealth closer to zero. A line is drawn between neighbouring cells if wealth was exchanged between the two neighbours during the preceding 500 cycles. Both runs use agents of class SSA with initial wealth 0, $x = 16$, $y = 16$, $j = 4$, and a jackpot J equal to $xyjG$. The top run uses $G = 0.5$, and by cycle 19000 all agents have zero wealth. If $G = 0.4$, however (bottom sequence), although the model behaves similarly for the first few thousand cycles (with some differences), it never reaches the termination state.

2.2.2. Imprecision in calculations

Even if parameters are exactly representable using floating point numbers, errors can creep in when the result of a calculation using the parameters is not itself exactly representable. In some cases, extra precision can help with this, but certain calculations using exact parameters (e.g. 2.0/3.0) are not exactly representable in binary floating point at any precision. Once an error has occurred in a calculation, it is possible for that error to accumulate in successive computations involving the result. According to Higham (2002, p. 14) poor results from imprecision in calculations most often arise from the "insidious growth of just a few rounding errors" rather than accumulation of many millions of rounding errors. He also finds that rounding errors can assist with the success of a computation rather than hinder it (pp. 22–24).

2.2.2.1. Illustration 2. The following example shows imprecision in calculations causing most of the agents to make the wrong decision. Consider a model containing 100 SSAs with initial wealth 0 on a 10×10 grid and jackpot, J, where J is an exactly representable number. Immediately after the lottery, the winning SSA's wealth should be $0.99\,J$ whereas the other SSAs' wealth should be $-0.01J$. There is no need to determine G, since we are only interested in the beginning of the simulation. Fig. 7 shows the distribution of donors at the beginning of the simulation for $J = 70$, when there are 87 agents who wrongly see themselves as donors and for $J = 700$, which shows the output in the absence of floating point errors. Donors (those who satisfy the supply-awareness-condition) are represented in grey whereas those who do not satisfy the supply-awareness-condition are represented in white.

The 87 agents who wrongly see themselves as donors are those surrounded by either 5 or 8 other agents with the same wealth as they have $[-0.7]_f$. They do so because, in double precision floating point arithmetic, the following two inequalities hold:

$$[6\times[-0.7]_f]_f > [[[[[[-0.7]_f+[-0.7]_f]_f+[-0.7]_f]_f + [-0.7]_f]_f+[-0.7]_f]_f+[-0.7]_f]_f \quad (3)$$

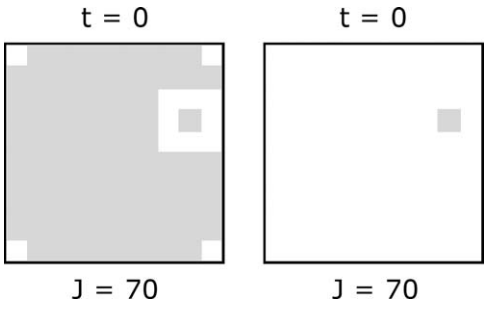

Fig. 7. Distribution of donors at the beginning of the simulation from two runs of CharityWorld ($J = 70$ and for $J = 700$) that would show exactly the same distribution were it not for errors in floating point arithmetic. Agents that see themselves as donors are coloured in grey, other agents in white. The distribution on the right ($J = 700$) shows the output in the absence of floating point errors.

for agents at the edge of the space, and:

$$[9\times[-0.7]_f]_f > [[[[[[[[[-0.7]_f+[-0.7]_f]_f+[-0.7]_f]_f+[-0.7]_f]_f+[-0.7]_f]_f+[-0.7]_f]_f+[-0.7]_f]_f+[-0.7]_f]_f \quad (4)$$

for non-edge-or-corner agents. The SSA-supply-awareness-condition is therefore satisfied for these agents when it should not be, mathematically speaking.

The agents on the corners do not see themselves as donors because for these agents, floating point arithmetic behaves like conventional arithmetic sufficiently that:

$$[4\times[-0.7]_f]_f = [[[[-0.7]_f+[-0.7]_f]_f+[-0.7]_f]_f+[-0.7]_f]_f \quad (5)$$

Thus, in a toroidal rather than bounded environment, since all agents would check their supply-awareness-condition as per Eq. (4), no agents would behave correctly as do the corner agents.

Repeating Illustration 2 for integer $J = 1,\ldots,100$, we found that in 34 cases, either edge agents or non-edge-or-corner agents or both satisfy their supply-awareness-condition. All of these involved a rounding error in the computations, and half were cases where only edge agents satisfied their supply-awareness-condition. The supply-awareness-condition of corner agents was not satisfied in any case. Of the 66 cases where none of the agents satisfied their supply-awareness-condition, all but four involved a rounding error in one or more of the computations involved. Just as in Illustration 1, we find that rounding errors do not necessarily mean that the model will behave incorrectly. For spatially-explicit agent-based models, however, if the environment is bounded there is the potential for errors to affect edge, corner, and other agents differently.

2.2.2.2. Illustration 3. A special case of imprecision occurs when a very small number is involved in a calculation whose result is an even smaller number that is outside the set of numbers representable using the floating point implementation. An example using the 8-bit representation discussed in Section 1.1 would be dividing the smallest denormalised number ($0\,000\,0001 = 1/64$) by 3. In this case, the result 'underflows' to zero. This problem demonstrates underflow.

This problem introduces the Portion Supply Agent (PSA), which has the same awareness condition as the Simple Supply Agent. Once the PSA is aware of the need to supply, it will then donate a proportion of the difference between its wealth and the average wealth of its neighbours to each of its neighbours unconditionally. The PSA-supply-approach-condition is therefore always true. The amount donated g is calculated as follows:

$$g = \frac{G}{\#N(A)+1}\left(w_A - \frac{\sum_{a\in N(A)} w_a}{\#N(A)}\right) \quad (6)$$

The model parameter G is used as a measure of the generosity of the donating agent. If $G = 1$, and all the neighbouring agents have equal wealth, then the donating agent A will redistribute its wealth so that it, and all its neighbours, have equal wealth.

In a simple simulation involving just two agents, if $G = 1$, the simulation will terminate in a single step, but if $G < 1$, the simulation should never terminate. However, in a computer, using $G = 0.75$ (or any $1.0 > G > 0.5$, but 0.75 is the simplest exactly representable such number), the simulation does terminate after a number of iterations, as the responding agent's wealth underflows from the smallest representable number to zero. Here, in contrast with Illustration 1, there is convergence when there should not be.

McCullough and Vinod (1999) argue that underflow is a pernicious error, since a model in which it occurs can produce "sensible-looking answers that are completely inaccurate" (p. 643). Its counterpart, overflow, which occurs when the result of a computation is larger in magnitude than the largest representable positive or negative number (as appropriate), is more easily detected on IEEE 754 compliant platforms, as they generate the appropriately signed infinity. Any subsequent addition or subtraction involving the result of such a computation has the result Not-a-Number according to IEEE 754, and any further computation involving Not-a-Number has result Not-a-Number, meaning that the occurrence of overflow is likely to propagate forward to the final output from the model.

The most effective way to guard against underflow is to make use of the IEEE 754 stipulated floating point error traps, discussed further in Section 2.3.1. Higham (2002, p. 27) also recommends watching the numbers generated during the course of a simulation. Though agent-based models might well be expected to produce a vast amount of such output, too much to expect a researcher to inspect each and every number, modern computers have sufficient disk space to store it. The output can then be searched automatically for very small exponents, e.g. using a regular expression like `E-[1-9][0-9][0-9]` with the Unix command grep. Models generating small numbers that eventually result in underflow in the manner illustrated here are restricted in how many cycles they can be run for. Using extra precision should extend the limit.

2.2.3. Mathematically equivalent expressions

Floating point errors can also cause problems when replicating the work of other authors, even if implemented on the same platform. This section briefly looks at how different implementations of what should be mathematically equivalent results can change the behaviour of the program.

2.2.3.1. Illustration 4. This problem focuses on a slight variation of the SSA called the Supply-Unhappy-Agent, or SUA. The SUA is defined by the following conditions for an agent A and one of its neighbours, $N \in n(A)$:

$$\text{SUA-supply-awareness-condition}(A) \Leftrightarrow \text{wealth}(A) > \text{local-mean-wealth}(A) \quad (7)$$

$$\text{SUA-supply-approach-condition}(A, N) \Leftrightarrow \text{wealth}(N) < \text{local-mean-wealth}(N) \quad (8)$$

An SUA is said to be 'happy' when its wealth is greater than the mean wealth of its neighbours, and 'unhappy' when its wealth is less than the mean wealth of its neighbours. The SUA will donate an amount $g = G$ to unhappy neighbours when it is happy.

This specification of the SUA should be enough, but to show how it need not be, we consider four mathematically equivalent ways of assessing whether the wealth of an agent is greater or less than its local mean wealth.

The default, and most intuitive way is to compute the mean as it appears in most textbooks and compare it with that of agent B to determine B's happiness. Using w_a for the wealth of agent a, $N(B)$ to represent the set of neighbours of B, and n to represent the cardinality of $N(B)$:

$$w_B > \frac{1}{n+1} \sum_{a \in N(B) \cup \{B\}} w_a \Leftrightarrow \text{happy}_{\text{inclusive mean}}(B) \quad (9)$$

Multiplying both sides of the inequality by $n + 1$ gives a mathematically equivalent method of determining the happiness of B (which is the same as the SSA-supply-awareness-condition (1)):

$$(n+1)w_B > \sum_{a \in N(B) \cup \{B\}} w_a \Leftrightarrow \text{happy}_{\text{inclusive total}}(B) \quad (10)$$

The wealth of the agent B can also be taken out of the calculation of the mean, without affecting the truth of the inequality (mathematically speaking), as shown below:

$$\begin{aligned} w_B &> \frac{1}{n+1} \sum_{a \in N(B) \cup \{B\}} w_a \\ \Leftrightarrow w_B &> \frac{1}{n+1}\left(w_B + \sum_{a \in N(B)} w_a\right) \\ \Leftrightarrow w_B &\left(1 - \frac{1}{n+1}\right) > \frac{1}{n+1} \sum_{a \in N(B)} w_a \\ \Leftrightarrow \frac{n}{n+1} w_B &> \frac{1}{n+1} \sum_{a \in N(B)} w_a \\ \Leftrightarrow n w_B &> \sum_{a \in N(B)} w_a \\ \Leftrightarrow w_B &> \frac{1}{n} \sum_{a \in N(B)} w_a \end{aligned} \quad (11)$$

Hence, the two further variants of SUA assess their happiness as follows:

$$w_B > \frac{1}{n} \sum_{a \in N(B)} w_a \Leftrightarrow \text{happy}_{\text{exclusive mean}}(B) \quad (12)$$

$$n w_B > \sum_{a \in N(B)} w_a \Leftrightarrow \text{happy}_{\text{exclusive total}}(B) \quad (13)$$

There are thus four mathematically equivalent variants of SUA which should have the same behaviour. Fig. 8 shows the results of using the four variants in

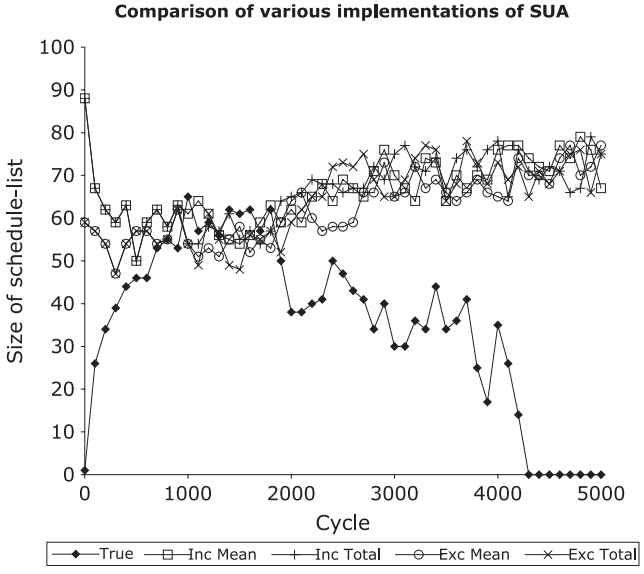

Fig. 8. Differences in behaviour between mathematically equivalent variants of the same model, shown using time-series graphs of the size of the schedule-list. The 'Inc Mean' line refers to Eq. (9), 'Inc Total' to Eq. (10), 'Exc Mean' to Eq. (12) and 'Exc Total' to Eq. (13). The solid diamond line shows the 'true' behaviour – that which should occur were it not for errors in floating point arithmetic ($G = 0.5$), when all variants behave in the same way. The other four lines show the differences between the implementation variants when $G = 0.7$. All runs used the same seed.

a 10×10 environment with initial wealth 0. Using $G = 0.5$ and a jackpot of 50, all four variants behave in exactly the same way, with the model converging by cycle 4300. However, when $G = 0.7$ and $J = 70$, each of the four variants produces a different behaviour by cycle 1100. The greatest difference in behaviour is between the exclusive and inclusive variants of SUA. Within these, smaller differences appear between the total and the mean variants.

The difference between the four differently behaving runs and the four runs using $G = 0.5$ is that 0.5 is exactly representable whilst 0.7 is not. An effect similar to that shown in Illustration 1 is thus occurring here. What the differently behaving runs show is that floating point calculations make it possible for different outputs to be generated from different implementations of mathematical expressions in the model design. How far this can lead to significantly different statistical signatures (Moss, 2001) is unknown.

2.3. Techniques

2.3.1. Detection of floating point errors

Platforms that implement IEEE 754 are required to provide facilities that allow the programmer to inspect various exception flags that are set when floating point errors occur (IEEE, 1985). These functions are provided in C in the ieeefp.h header file – fpgetsticky() and fpsetsticky()[3] being most relevant. The former function allows the programmer to inspect the state of the flags over all floating point operations since the flags were set to zero using the latter function. The authors are not aware of any such functionality being directly available in Java, though a superset of Java called Borneo, currently at the specification level only, does have such a capability (Darcy, 1998).

The five errors that the IEEE standard stipulates be detectable are:

1. Inexact result – the computation results in a number that cannot be represented by one of the numbers available under the given precision.
2. Underflow – the computation results in a number that is not exactly representable and is smaller than the smallest non-zero number representable using normalised numbers.[4]
3. Overflow – the computation results in a number larger than any representable number.[5]
4. Division by zero.
5. Invalid operation – e.g. $\sqrt{(-1)}$, $0/0$, $0 \times \infty$, ∞/∞ or any comparison with NaN.

[3] Unix manual page fpgetround(3C).
[4] For the sake of clarity, x is smaller than y iff $|x| < |y|$.
[5] Similarly, x is larger than y iff $|x| > |y|$.

A floating point error in a computation will not necessarily lead to an incorrect decision by an agent. Aborting the simulation as soon as an error occurs is therefore overly cautious. An alternative is to issue a warning in the event of a floating point error: the user may be able to judge from the context whether the error threatens the validity of the run. Detecting and warning about floating point errors at least means the user can be certain when a model run does *not* entail any floating point errors, set runs that produce such errors to one side for further inspection, and concentrate on those that do not. Whether focusing on runs that do not produce errors is a potential source of bias is a matter for further research.

If the software uses programming libraries that make floating point calculations that do not have an impact on agent decision making (e.g. a tcl/tk display), then a perfectly valid simulation could issue a floating point error warning. The only way to guard against this is to surround each relevant calculation by a call to fpsetsticky(0) beforehand to unset all the floating point error flags and a call to fpgetsticky() afterwards to check if any flag has been set by the calculation. This was the approach taken in the software used.

In the results, this technique is referred to as 'Warn'.

2.3.2. Tolerance windows

Floating point operations, whilst not strictly accurate, often achieve a level of accuracy within a relatively small tolerance, ε. Replacing the comparison operators as shown below can prevent incorrect decisions being made by agents:

- $x \tilde{>} y \Leftrightarrow x > [y+\varepsilon]_f$
- $x \tilde{\geq} y \Leftrightarrow x \geq [y-\varepsilon]_f$
- $x \tilde{<} y \Leftrightarrow x < [y-\varepsilon]_f$
- $x \tilde{\leq} y \Leftrightarrow x \leq [y+\varepsilon]_f$
- $x \tilde{=} y \Leftrightarrow (x \geq [y-\varepsilon]_f) \wedge (x \leq [y+\varepsilon]_f)$
- $x \tilde{\neq} y \Leftrightarrow (x < [y-\varepsilon]_f) \vee (x > [y+\varepsilon]_f)$

where ε is a non-negative floating point number.

The SSA-supply-awareness-condition (1) and SSA-supply-approach-condition (2) are then written thus:

$$\text{SSA-supply-awareness-condition}(A) \Leftrightarrow (\#N(A)+1)w_A \tilde{>} \sum_{a \in N(A) \cup \{A\}} w_a \tag{14}$$

$$\text{SSA-supply-approach-condition}(A, n(A)) \Leftrightarrow (\#N(A)+1)w_{n(A)} \tilde{<} \sum_{a \in N(A) \cup \{A\}} w_a \tag{15}$$

Correct termination of the model is then often possible, provided a good value has been chosen for ε: high enough to detect as equal two different numbers that

would be equal in the absence of floating point errors, but low enough to detect as different two numbers that would indeed be different without such errors. In CharityWorld, what constitutes a 'good value' for ε depends on the jackpot, J.

Consider Illustration 1, for example. The parameter settings are $x = y = 3$, with agent class SSA, and $w_0 = 0$. Using the $G = 0.4$ case with $J = 72$ and $\varepsilon = 10^{-11}$, the model behaves in exactly the same way as when $G = 0.5$, $J = 90$ and $\varepsilon = 0$, converging with all agents having wealth within the range $\pm\varepsilon$ after 20 cycles. If $\varepsilon = 10^{-13}$, however, the model does not terminate and the agents never all have wealth within the range $\pm\varepsilon$. Using $J = 720$, we need to set $\varepsilon = 10^{-9}$. For very large $J = 72{,}000{,}000$, $\varepsilon = 0.1$ does not converge and correct behaviour is achieved with $\varepsilon = 3.5$.

2.3.3. Rewriting expressions to reduce floating point errors

Some of the literature on floating point arithmetic concerns algorithms for specific kinds of computation with reduced floating point errors. One such algorithm of particular relevance to CharityWorld is the Kahan Summation Algorithm (Kahan, 1965). The Kahan Summation Algorithm reduces the error in computing the sum of a series of numbers, by keeping track of a correction term throughout the sum. Implementing the Kahan Summation Algorithm when updating the wealth of an agent in Illustration 1 can, depending on the value of j, enable a simulation to converge that would not otherwise do so because of floating point errors. (In the results, this technique is referred to as 'Kahan'.) Even when this occurs, it is still possible that agents will make a decision to donate when they should not do so – so although the simulation terminates and converges, the sequence of donations is not necessarily the same as it would have been were it not for floating point errors.

2.3.4. Using strings

One suggestion when making comparisons is to write floating point numbers to strings with a specified number of decimal places. This appears in the perlop[6] manual page, for example. The use of strings can be extended by writing the result of a floating point operation to a string with the specified degree of accuracy for the floating point implementation used, and then converting that string back into a floating point number again. The idea is to maintain a consistent base 10 accuracy in the result before it is used in another calculation. It is rather like using a calculator, and writing down the numbers that appear on the screen so they can be used in a subsequent operation.

[6] http://www.perl.com/doc/manual/html/pod/perlop.html#Floating_point_arithmetic.

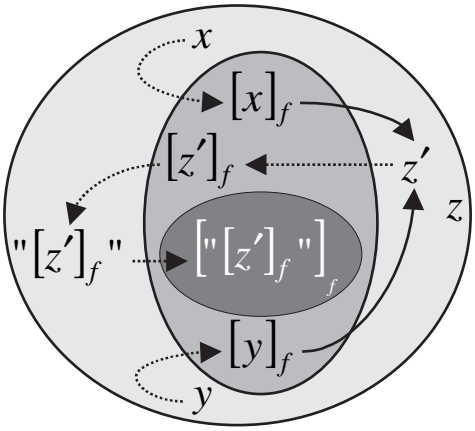

Fig. 9. Schematic diagram of the effect of using strings to convert the result of a calculation z' to the nearest representable number to the n significant decimal figures provided by the floating point implementation, $[''[z']_f'']_f$, which introduces another two potential sources of errors. The results of all calculations then belong to the subset of floating point numbers that are the nearest representable number to n significant figures of some floating point number, indicated by the set with the darkest grey background and thin border.

For example, in single precision IEEE 754 floating point, which guarantees at least 6 significant figures of accuracy, the closest representable number to 0.4 is 0.40000000596046447753906250. Similarly, the closest representable number to 0.1 is 0.100000001490116119384765625. The closest representable number to the product of the two representable numbers ($[[0.1]_f \times [0.4]_f]_f$) is 0.040000002831220626831054687 5, but $[0.04]_f$ (the closest representable number to 0.04) is 0.039999999105930328369140625. If the product is printed to a string with six significant figures of accuracy,[7] the string reads "4.00000E−2". If this string is then converted back to a `float`, the result is $[0.04]_f$ rather than $[[0.1]_f \times [0.4]_f]_f$, so it would seem that we have dealt quite nicely with any problem of accumulated errors from multiplying two non-representable numbers. Indeed, this method enables correct behaviour in Illustration 1.

However, not all calculations using this method work out so well. For example, 1.0 and 3.0 are both exactly representable, but when 1 is divided by 3, the nearest representable floating point number is 0.333333343267440795898437 5. This would convert to a string "3.33333E−1", with nearest representable number 0.333332985639572143554687 5. If the latter is multiplied back up by 3.0, then the result obtained is 0.999999 (to six significant figures), rather than the desired result of 1.000000 which is obtained when multiplying the former by 3.

[7] In C, this is achieved using `sprintf(str, "%.6E", num)`, where `str` is a buffer declared of type `char *`, and `num` is declared to have type `float` or `double`; in Java the same effect is achieved with `String str = (new DecimalFormat("0.000000E0")).format(num)`.

Converting to a string and back again introduces another two potential sources of error (Fig. 9). Although these errors can counteract earlier errors, they can also exacerbate them.

2.3.5. Using offsets

Another tactic for dealing with floating point errors is to use a large offset M when making comparisons. This is a piece of folklore among the programming community, an instance of which appears on a game development on-line discussion forum.[8]

- $x \hat{>} y \Leftrightarrow [x+M]_f > [y+M]_f$
- $x \hat{\geq} y \Leftrightarrow [x+M]_f \geq [y+M]_f$
- $x \hat{<} y \Leftrightarrow [x+M]_f < [y+M]_f$
- $x \hat{\leq} y \Leftrightarrow [x+M]_f \leq [y+M]_f$
- $x \hat{=} y \Leftrightarrow [x+M]_f = [y+M]_f$
- $x \hat{\neq} y \Leftrightarrow [x+M]_f \neq [y+M]_f$

Adding a sufficiently large term to the operands of a comparison operator is intended to eliminate all the least significant digits of the operands due to floating point errors, thereby enabling an accurate comparison to be made. When the operands are negative, however, fewer of the least significant bits will be lost on addition to M than when they are positive. Thus, there are many cases for which $x \hat{=} y$, but $-y \hat{>} -x$. One way of dealing with this would be to subtract M from negative operands and add M to positive operands. However, this requires accumulated floating point errors to always maintain the correct sign, which IEEE 754 floating point arithmetic does not guarantee. Another approach is to ensure that all numbers are always positive. This may not be possible for all models, but in CharityWorld it can be achieved by using an initial wealth greater than or equal to the ticket price.

For example, suppose we want to compare 0.1×0.4 with 0.04. As mentioned in Section 2.3.4, in single precision the closest representable number to the product of the closest representable numbers of the operands and the closest representable number to 0.04 are not equal. If we add 1024 to both, however, we get:

$$0.040000002831220626831054687 5 + 1024 = 1024.0400390625$$

and

$$0.039999999105930328369140625 0 + 1024 = 1024.0400390625$$

We can then correctly determine that $(0.1 \times 0.4) + 1024$ is equal to $0.04 + 1024$. However, in single precision, adding 1024 to the nearest representable number to 0.0002 gives the result 1024.000244140625, which is the same result as adding 1024 to the nearest representable number to 0.0003. Thus although single precision is quite capable of detecting that 0.0002 is not equal to 0.0003, using an offset of 1024 has them as equal. Seeing them as different would require a smaller offset, illustrating that for most models there is no compile-time setting for M that guarantees correct behaviour for all parameter settings. It is also possible for accumulations in floating point errors to be sufficient that there is no offset that will enable correct behaviour, as will be demonstrated for tolerance windows in Counter-example 2.

Conceptually, the idea behind offsets may seem appealing, and their use in CharityWorld can enable correct behaviour to occur when it would otherwise not. The theoretical basis for their use, however, is not particularly sound. Since IEEE floating point stipulates exactly rounded arithmetic operations, even the least significant bit of one of the numbers being compared can have an effect on their sum with M. For example, consider two numbers $a = 1 + 2^{-52}$ and $b = 1$, with $M = 2^{53}$. In double precision, $a + M$ is rounded up to $M + 2$, but $b + M = M$. Even with the smallest possible difference between a and b, therefore, using offsets would not detect them as being equal, though with the same value of M, 0 and 1 would be detected as being equal.

2.3.6. Using extended precision

The IEEE 754 standard allows for extended precision floating point formats. Double extended format stipulates at least 64 significant bits of precision and a minimum of 15 bits for the exponent. The actual precision offered is implementation specific. There are two ways extended precision could be used. One involves using double extended precision to store the variables and results of computations, but converting to double precision when making comparisons. The basis for this is the hope that rounding errors will be kept to the extra least significant bits of accuracy provided by double extended precision. When converting to double precision in comparison operations, these rounding errors should then be lost. The effect is similar to using offsets with double extended precision, but with a fixed compile-time choice of offset (though the problems with negative numbers would not apply). This is not recommended for the reasons given above.

The second method involves using double extended precision to gain extra accuracy. Using extended precision will not affect rounding errors due to conversion between binary and decimal formats, and it will also not necessarily enable computations to be made exactly that could not be made exactly in double precision. It is conceivable, however, that using double extended precision could enable a model to run for longer without generating floating point errors, particularly if combined with the use of interval arithmetic (see below).

[8] http://www.flipcode.com/cgi-bin/msg.cgi?showThread=00000581&forum=3dtheory&id=-1.

The catch with using extended precision is the increased flexibility that the IEEE 754 standard allows for these formats. This flexibility has the consequence that a model may run differently on one platform than on another given the same parameter settings and initial seed and thus impacts the ease with which any results can be replicated.

2.3.7. Interval arithmetic

A further stipulation of the IEEE 754 standard is that functionality be provided to change the rounding direction (which is round-to-nearest by default) (IEEE, 1985, p. 5). This is known as hardware rounding. If your platform allows you to do this, there is the possibility of representing each floating point number by an interval, the minimum of which applies to the result of rounding towards −infinity in an operation (the largest representable number not greater than the exact result), and the maximum to the result of rounding towards +infinity (the smallest representable number not less than the exact result). Where these facilities are not available, software rounding can be used.[9] Abrams et al. (1998) compare various approaches to rounding with examples in C. The true result of the computation is then known to lie within these bounds. Furthermore, by modifying the comparison operators so they are true if and only if they apply to all members of their operands, any action taken based on a decision depending on the comparison operators can be known to be correct. However, it is still possible for an action *not* to be taken because the comparison operator returned false when the correct result might have been true. It is possible to detect this by checking the opposite of a comparison operator when it returns false. If a comparison operator and its opposite both return false, then there is uncertainty about which action to take, and the agent could make the wrong decision. Having detected this, a warning could be issued, or the simulation could abort. If it did not happen too often, the simulation could fork, ensuring that at least one of the branches contains the correct result.

Various forms of interval arithmetic exist (Ullrich and von Gudenberg, 1990). The following uses the upper and lower bound approach (Alefeld and Herzberger, 1983). The arithmetic operators are outlined below for two intervals $[a_1, a_2]$ (where $a_1 \leq a_2$) and $[b_1, b_2]$ (where $b_1 \leq b_2$), using $[x \otimes y]_{f-}$ to represent the result of $x \otimes y$ rounded towards negative infinity, and $[x \otimes y]_{f+}$ to represent the result of $x \otimes y$ rounded towards positive infinity, where $\otimes$ stands for any of the arithmetic operators (Alefeld and Herzberger, 1983). The situation is rather more complex for multiplication and division than for addition and subtraction because of the various possibilities for the signs of the operands and how these affect the possible minimum and maximum outcomes of the operation.

- $[a_1, a_2] + [b_1, b_2] = [[a_1 + b_1]_{f-}, [a_2 + b_2]_{f+}]$
- $[a_1, a_2] - [b_1, b_2] = [[a_1 - b_2]_{f-}, [a_2 - b_1]_{f+}]$
- $[a_1, a_2] \times [b_1, b_2] = [\min\{[a \times b]_{f-} | a \in \{a_1, a_2\}, b \in \{b_1, b_2\}\}, \max\{[a \times b]_{f+} | a \in \{a_1, a_2\}, b \in \{b_1, b_2\}\}]$
- $[a_1, a_2] \div [b_1, b_2] = [\min\{[a \div b]_{f-} | a \in \{a_1, a_2\}, b \in \{b_1, b_2\}\}, \max\{[a \div b]_{f+} | a \in \{a_1, a_2\}, b \in \{b_1, b_2\}\}]$

In interval arithmetic there are two modes for each comparison operator corresponding to modal logic (Kripke, 1963): a 'possible' mode, in which the truth of the comparison operator holds for at least one pair of members of each operand, and a 'necessary' mode (represented using a square $\Box$), in which the truth of the comparison operator holds for all pairs of members of each operand (Alefeld and Herzberger, 1983). Using the necessary mode, although there may have been floating point errors in the computation of the operands, there is certainty that whatever the real numbers might have been, the required relationship holds.

- $[a_1, a_2] \Box > [b_1, b_2] \Leftrightarrow a_1 > b_2$
- $[a_1, a_2] \Box \geq [b_1, b_2] \Leftrightarrow a_1 \geq b_2$
- $[a_1, a_2] \Box < [b_1, b_2] \Leftrightarrow a_2 < b_1$
- $[a_1, a_2] \Box \leq [b_1, b_2] \Leftrightarrow a_2 \leq b_1$
- $[a_1, a_2] \Box = [b_1, b_2] \Leftrightarrow (a_1 = a_2) \wedge (b_1 = b_2) \wedge (a_1 = b_2)$
- $[a_1, a_2] \Box \neq [b_1, b_2] \Leftrightarrow (a_1 < b_2) \vee (a_2 > b_1)$

Using intervals enables the user to run a simulation until a potentially wrong decision is made because of floating point errors. It is possible, for example, that an agent-based model is only required to run for a relatively small number of cycles, and that during this time the floating point errors do not accumulate enough to create any uncertainty in the comparison operators.

2.4. Counter-examples

2.4.1. Counter-example 1

This counter-example was derived to illustrate the weaknesses with the 'warn' technique discussed in Section 2.3.1. Just as the SUA can have four variants for determining the supply-awareness-condition and supply-approach-condition, so can the SSA, though with the SSA the inclusive and exclusive variants differ in the set of neighbouring agents that satisfy the approach-condition. Here we consider just two, the mean and total inclusive variants, which should behave

[9] In C, the fpsetround() function is provided to control the rounding direction on the CPU (Unix manual page fpgetround(3C)). No direct provision is currently made for hardware rounding in Java, though utilities such as those available at http://www.dose-of-reality.org/?IEEE754, which provide functions that compute the next floating point number before and after the argument can be used to implement an interval arithmetic that will give wider bounds than those from using hardware rounding.

in the same way. The $\mathrm{SSA}_{\mathrm{total}}$ variant is the SSA exactly as described in (1) and (2) above, whilst the $\mathrm{SSA}_{\mathrm{mean}}$ variant computes the average as a fraction and compares it with the wealth of the agent:

$$\mathrm{SSA}_{\mathrm{mean}}\text{-supply-awareness-condition}(A)$$
$$\Leftrightarrow w_A > \frac{1}{\#N(A)+1}\sum_{a\in N(A)\cup\{A\}} w_a \tag{16}$$

$$\mathrm{SSA}_{\mathrm{mean}}\text{-supply-approach-condition}(A, n(A))$$
$$\Leftrightarrow w_{n(A)} < \frac{1}{\#N(A)+1}\sum_{a\in N(A)\cup\{A\}} w_a \tag{17}$$

Using similar settings as in Illustration 1 when the behaviour of the model was as predicted mathematically (20×20 environment, $g = 0.5, j = 5, J = xyjG = 1000$), and adding facilities to check for errors in precision, $\mathrm{SSA}_{\mathrm{total}}$ achieves convergence without any warnings, since all of the calculations for the approach and awareness conditions can be computed exactly. $\mathrm{SSA}_{\mathrm{mean}}$, by contrast, though it behaves in the same way as $\mathrm{SSA}_{\mathrm{total}}$, generates a warning almost every time the average wealth is computed. Using a Moore neighbourhood means that computing the average requires a division by 9, which yields an imprecise result in base 2 (unless the average happens to be a multiple of nine divided by an integer power of two). Thus, although $\mathrm{SSA}_{\mathrm{mean}}$ with these parameters produces mathematically correct output, errors are detected, demonstrating how a run with errors does not necessarily mean the output is incorrect.

2.4.2. Counter-example 2

This problem shows how there can be values of the parameters that preclude correct behaviour whatever value is used for ε when using tolerance windows described in Section 2.3.2. To show this we must introduce Proposition 1.

Proposition 1: *Let x and y be two floating point numbers and assume that summation and subtraction are exactly rounded (as the IEEE 754 standard stipulates). Let $D(x \pm y)$ be the error in computing $[x \pm y]_f$, expressed as $D(x \pm y) = [x \pm y]_f - (x \pm y)$. Then,*

$$|D(x \pm y)| \leq \min\{|x|, |y|\} \tag{18}$$

Proof: From the definition of $D(x \pm y)$, if $[x \pm y]_f = x$, then $|D(x \pm y)| = |y|$. Similarly, if $[x \pm y]_f = \pm y$, then $|D(x \pm y)| = |x|$. The exactly rounded solution $[x \pm y]_f$ will imply an error at most as large as the absolute error that we would obtain by assuming $[x \pm y]_f = x$, or $[x \pm y]_f = \pm y$, since x and $\pm y$ are floating point numbers and not necessarily the nearest to $(x \pm y)$. Therefore, $|D(x \pm y)| \leq \min\{|x|, |y|\}$. □

Now consider, as in Illustration 1, a model containing 9 SSAs on a 3×3 grid with initial wealth $w_0 = 0$. The agent in the middle of the grid wins the lottery and receives a jackpot of $J = 9jG$. Let $j = 250{,}000{,}000$ and $G = 0.4$. After the initialisation process, the model should converge in j cycles, in each of which the winner donates G to each of its 8 neighbours.

With these parameter settings, after $(j-1)$ cycles, the agents' wealth is as follows:

$$w_F \approx 32.06,\ w_R \approx -0.08635, \text{ and } \Sigma_a w_a \approx 31.37.$$

where

w_F is the winner's wealth.
w_R is the wealth of any of the agents who is not the winner.
$\sum_a w_a$ is the sum of the nine agents' wealth.

At this point, for the model to behave correctly, both the SSA-supply-awareness-condition (14) and the SSA-supply-approach-condition (15) should be satisfied for the winner. For our particular case, these two conditions are as follows, from the definition of D in Proposition 1:

$$\text{SSA-supply-awareness-condition}(F)$$
$$\Leftrightarrow 9w_F > \left[\sum_a w_a + \varepsilon\right]_f$$
$$\Leftrightarrow 9w_F > D\left(\sum_a w_a + \varepsilon\right) + \sum_a w_a + \varepsilon$$
$$\Leftrightarrow \varepsilon < 9w_F - \sum_a w_a - D\left(\sum_a w_a + \varepsilon\right) \tag{19}$$

$$\text{SSA-supply-approach-condition}(F, R)$$
$$\Leftrightarrow 9w_R < \left[\sum_a w_a - \varepsilon\right]_f$$
$$\Leftrightarrow 9w_R < D\left(\sum_a w_a - \varepsilon\right) + \left(\sum_a w_a - \varepsilon\right)$$
$$\Leftrightarrow \varepsilon < \sum_a w_a - 9w_R + D\left(\sum_a w_a - \varepsilon\right) \tag{20}$$

For the jackpot winner to donate money at the last cycle (cycle j), it is necessary that:

$$\varepsilon < 32.15 + D(31.37 - \varepsilon), \text{ from (20)} \tag{21}$$

From Proposition 1, $|D(31.37 - \varepsilon)| \leq 31.37$. So the following condition is necessary to correctly undertake the last cycle:

$$\varepsilon < 32.15 + D(31.37 - \varepsilon) < 32.15 + 31.37 = 63.52 \tag{22}$$

If the last cycle has been correctly undertaken, Eq. (22) holds and so, of course, does $\varepsilon < 63.52$. The agents' wealth is then the following:

$w_F \approx 28.86$, $w_R \approx 0.3137$, and $\Sigma_a w_a \approx 31.37$.

At this point we have to make sure that the winner does not satisfy the SSA-supply-awareness-condition, otherwise the model would not achieve termination as it should. From Eq. (19), preventing the jackpot winner from satisfying the SSA-supply-awareness-condition would require:

$$\varepsilon \geq 228.37 - D(31.37+\varepsilon) \tag{23}$$

Again, from Proposition 1, $|D(31.37 + \varepsilon)| \leq 31.37$. So the following condition is also necessary for the model to achieve termination at the right time:

$$\varepsilon \geq 228.37 - D(31.37+\varepsilon) \geq 228.37 - 31.37 = 197 \tag{24}$$

But, as is evident from Eq. (22), for the winner to donate money at the last cycle (cycle j) we needed $\varepsilon < 63.52$. There is thus no value of ε that enables correct behaviour.

As the above examples show, ε cannot be set to any compile-time constant, and leave the user assured that the program will behave correctly.

2.4.3. Counter-example 3

This counter-example uses the Portion Supply Agent (PSA) from Illustration 3, to demonstrate an issue with using the 'strings' technique (Section 2.3.4). Here, the initialisation is slightly different from elsewhere, in that the jackpot J is given directly to the winning agent, with none of the agents having to pay a ticket price to enter the lottery. Agents not receiving the jackpot retain their initial wealth w_0, and the winning agent's wealth after initialisation is $w_0 + J$.

Using $x = y = 3$, $w_0 = 0$, $J = 1$, $G = 1$, and the jackpot winner at location (2, 2), with real numbers, the model should converge after 1 cycle, with each agent having wealth 1/9. Errors in the floating point arithmetic mean that the winning agent has a slightly different wealth from the other agents, and correct behaviour does not occur using normal floating point arithmetic. Using a technique such as tolerance windows (Section 2.3.2), the correct behaviour can be generated.

Using the string conversion process described in Section 2.3.4, however, the model does not terminate. Initially, the winning agent has wealth 1 and the other agents all have wealth 0. Since $G = 1$, the winning agent donates $g = 1/9$ to each of the other agents in turn. In what follows, $x\ldots_n\ldots x$ is used as shorthand for the digit x repeated n times. When written to a string, the donation g is "1.1…$_{14}$…1E−1" to the 15 significant figures of accuracy guaranteed by double precision, which when converted back to a double is ["1.1…$_{14}$…1E−1"]$_f$. As it donates to each neighbour in turn, the winning agent's wealth goes from ["1.0…$_{14}$…0E0"]$_f$ to ["8.8…$_{13}$…89E−1"]$_f$ to ["7.7…$_{13}$…78E−1"]$_f$, and so on, until after the last donation, it has wealth ["1.1…$_{13}$…12E−1"]$_f$ and all the other agents wealth ["1.1…$_{14}$…1E−1"]$_f$. In the next cycle, the winning agent begins with wealth that is greater than the average, and so it donates again. This time, the amount to donate is the nearest number to a ninth of ["1.0…$_{14}$…0E−15"]$_f$, which is, to 15 significant figures ["1.1…$_{14}$…1E−16"]$_f$. The winning agent faithfully attempts to donate this small quantity to each of its neighbours, but the nearest representable number to the sum of ["1.1…$_{14}$…1E−1"]$_f$ and ["1.1…$_{14}$…1E−16"]$_f$ is ["1.1…$_{14}$…1E−1"]$_f$ to 15 significant figures, and thus the neighbours receive nothing! Similarly, the winning agent's wealth of ["1.1…$_{13}$…12E−1"]$_f$ is also untouched by this false act of generosity. In the following cycle, the winning agent is thus again prepared to donate, and the process repeats endlessly.

2.5. Software

The software used to conduct the experiments is written in Objective C and makes use of the Swarm libraries (version 2.1.1 or 2.2). The CharityWorld model is written in a manner that follows a fairly standard simple Swarm model implementation, with the exception that all floating point numbers are represented using objects rather than the standard `double` C data type. These objects all belong to the class DoubleSimple, which contains a `double` instance variable, and methods to replace the arithmetic operators {+, −, *, /} and the comparison operators. Subclasses of DoubleSimple implement the various techniques discussed above, and DoubleSimple features creation methods that cause all new floating point objects to belong to one of these subclasses rather than DoubleSimple itself (Fig. 10). The user can therefore specify which subclass of DoubleSimple they wish to use throughout a particular simulation, and hence which technique will be used to manage floating point issues.

Ideally, the DoubleSimple classes would have been implemented using C++ classes in Objective-C++, which, through operator overloading, would have made code using these classes more readable, and reduced the memory management burden on the programmer. However, at the time of programming, an Objective-C++ compiler compatible with Swarm was not available. From the gcc mailing list, Objective-C++ will hopefully be made available in gcc version 3.5 (Laski, 2004).

The parameter file to the software has a Scheme syntax (Dybvig, 2003), typical of Swarm models.

The following shows an example parameter file, which applies to Illustration 1:

```
(list
(cons 'modelSwarm
(make-instance 'PaperModelSwarm
```

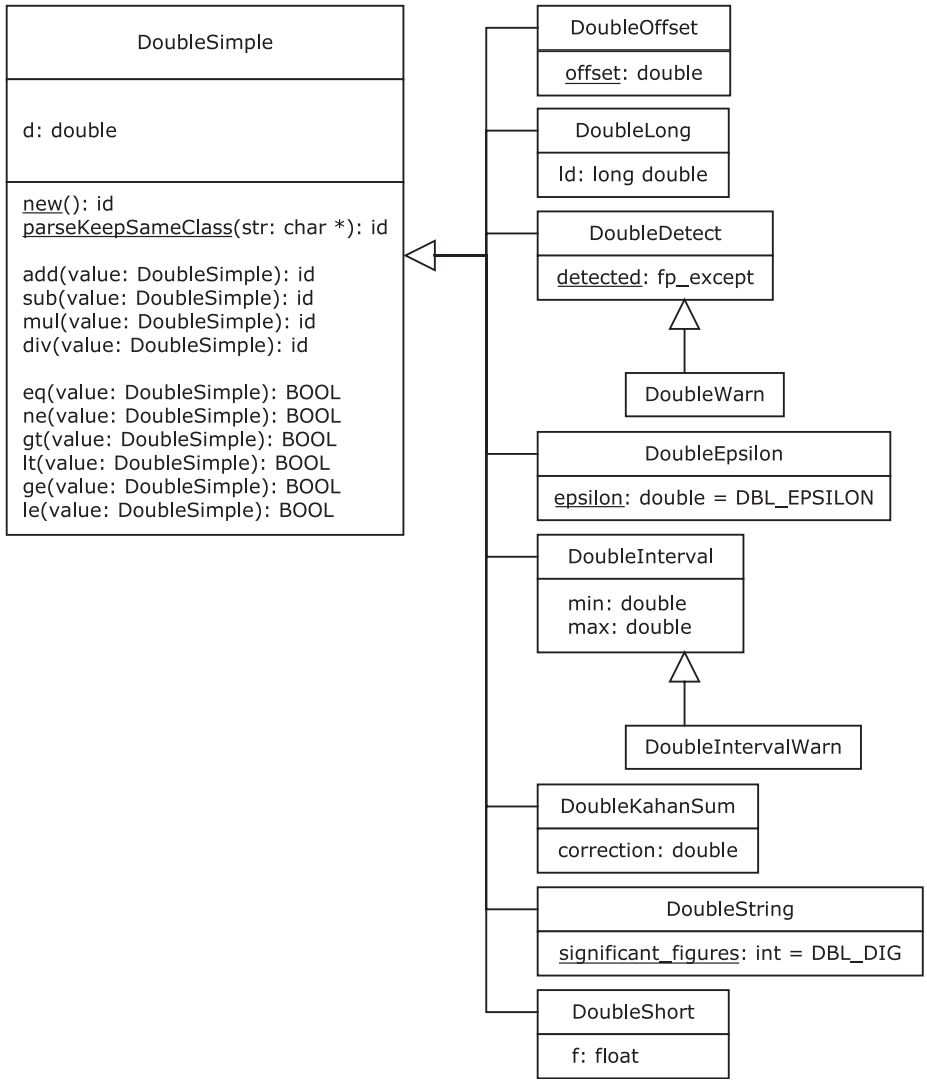

Fig. 10. UML Class Diagram depicting the implementation of the various techniques for dealing with floating point arithmetic issues compared in this paper. The constants DBL_DIG and DBL_EPSILON appear in the C header file float.h, and the fp_except type is defined in ieeefp.h.

```
#:xSize 3
#:ySize 3
#:jackpotStr DoubleWarn=4.5!ALL
#:initialWealthStr DoubleWarn=0.0
#:agentClassStr SSA
#:generosityStr DoubleWarn=0.5
#:winnerCoord "(2,2)"
#:loserChooser all)))
```

The parameters for the model are entered in the parameter file on the lines beginning '#:'. Parameters xSize and ySize are integers, and refer to the size of the environment, x and y as described in Section 2.1. The class of agent to use is entered as the value for agentClassStr, and may be one of **`SSA`**, **`SSAMean`**, **`SSATotal`**, **`PSA`**, **`SUAInclusiveMean`**, **`SUAInclusiveTotal`**, **`SUAExclusiveMean`**, **`SUAExclusiveTotal`**. The winnerCoord parameter is used to specify an agent in the space to win the lottery, either using the word **`random`** to signify an arbitrary choice, or in the format "(*wx*,*wy*)", where *italics* should be replaced with the required setting, and **`bold`** should appear as written. The loserChooser parameter value should be either the word **`all`**, or, for Counter-example 3, the word **`none`**.

The jackpot J, initial wealth w_0, and supply parameter G are entered as values for jackpotStr, initialWealthStr and generosityStr, respectively, using a string with the following format, where *class* is replaced with one of the classes shown in Fig. 10:

```
class=value
```

Certain classes require or permit configuration. Configuration need only appear the first time the class appears in the parameter file (e.g. in jackpotStr above),

and is appended to the end of the *value* without whitespace. The following details the configuration of those classes that require it:

DoubleWarn: Use **!***trap*, where *trap* is one of five strings representing the IEEE 754 stipulated exceptions: **INV** (invalid operation), **OFL** (overflow), **UFL** (underflow), **IMP** (imprecision), **DZ** (division by zero); or **ALL**, to trap all of them. Multiple traps can also be enabled through a sequence of **!***trap*s, e.g. `DoubleWarn = 0.4!OFL!UFL` – trap overflow and underflow only.
DoubleEpsilon: Use **~***epsilon*, where *epsilon* is the size of the tolerance window required.
DoubleOffset: Use **~***offset*, where *offset* is the size of offset required.
DoubleString: Use **s***sf*, where *sf* is the number of significant figures required.

The DoubleWarn class is used to implement the technique described in Section 2.3.1, DoubleEpsilon for that in Section 2.3.2, DoubleKahanSum for Section 2.3.3, DoubleString for 2.3.4, DoubleOffset for 2.3.5, DoubleLong for 2.3.6, and DoubleIntervalWarn for 2.3.7. DoubleShort is used in Illustration 1 (Section 2.2.1). DoubleSimple is used as a control.

Output from the model is driven by verbosity settings, which are specified in a file given to the model executable as an argument to the **+V** flag. Each verbosity setting details a particular situation in which to print a message, and a level of verbosity at which to print it, with 0 being used to signify that the model should always print the message. By default, a message is never printed. When comparing output of the model between various techniques, we checked that in each cycle, the sequence of donations and the agents appearing on the schedule-list match between run-generated outputs. This requires a verbosity file set up as follows:

```
Cycles 0
DonorContents 0
TransfersNoMoney 0
```

The executable for CharityWorld should thus be called from the command line as follows:

charity +V *verbosity_file parameter_file*

The output from running the model with the parameter file and verbosity file given above looks something like the following:

```
../../charity - Flags:
1: +V ../check.verby
Seed: 321654789
Money transferred from (2, 2) to (2, 2)
Money transferred from (1, 2) to (2, 2)
Money transferred from (3, 1) to (2, 2)
Money transferred from (1, 3) to (2, 2)
Money transferred from (2, 1) to (2, 2)
Money transferred from (2, 3) to (2, 2)
Money transferred from (1, 1) to (2, 2)
Money transferred from (3, 2) to (2, 2)
Money transferred from (3, 3) to (2, 2)
Content of donors list:
[(2, 2)]
-------------------------[ Cycle 000000001
Money transferred from (2, 2) to (3, 1)
Money transferred from (2, 2) to (3, 2)
Money transferred from (2, 2) to (1, 2)
Money transferred from (2, 2) to (2, 1)
Money transferred from (2, 2) to (2, 3)
Money transferred from (2, 2) to (1, 1)
Money transferred from (2, 2) to (1, 3)
Money transferred from (2, 2) to (3, 3)
Content of donors list:
[]
1: No beggars or donors left!
Terminated
Converged - certain
```

The transfers of money before Cycle 1 are to the winner of the jackpot from all the agents participating in the lottery.

As well as checking the sequence of donations and the agents on the schedule-list, we also checked for termination (which occurs when there are no agents on the schedule-list) and convergence at termination. Convergence occurs when all agents have equal wealth at termination, where equality is defined by the particular technique being used. The certainty of the convergence is checked in case a technique allows both its 'not-equal' and 'equal' operators to return 'false' at the same time. This could happen when, for example, two non-singleton intervals with non-empty intersection are being compared.

The models were all run using the same default seed, given in the example output above. This is to ensure the closest possible comparability between techniques. To use a different seed, the **-s** flag can be given to the command line after the parameter file. A full list of available flags and options can be obtained by running the executable without any command line options.

The source code is available on-line at http://www.macaulay.ac.uk/fearlus/floating-point/charity-world/, along with all parameter files used to generate results in this paper.

3. Results

The results are given in Table 1, which shows those techniques not involving a parameter to configure them

Table 1
Results of various methods of coping with errors in floating point arithmetic that do not require a configuration parameter, together with the control

Problem		Control	Warn (Section 2.3.1)	Kahan (Section 2.3.3)	Extended (Section 2.3.6)		Interval (Section 2.3.7)
					Cygwin	Solaris	
I1	$G = 0.5$	N	N	N	N	N	N
	$G = 0.4$	1	1_0	N	1	1	1_1
I2	$J = 700$	N	N	N	N	N	N
	$J = 70$	0	0_0	N	N	0	N_0
I3		537	537_{537}	537	8223	8247	537_{537}
I4	$SUA_{exc\text{-}total}$	0	0_0	15	12	0	15_0
	$SUA_{inc\text{-}total}$	0	0_0	15	12	0	15_0
	$SUA_{exc\text{-}mean}$	0	0_0	15	12	0	89_0
	$SUA_{inc\text{-}mean}$	0	0_0	15	12	0	89_0
C1	SSA_{total}	N	N	N	N	N	N
	SSA_{mean}	N	N_0	N	N	N	N
C2	$G = 0.5, j = 2.5e8$	N	N	N	N	N	N
	$G = 0.4, j = 2.5e8$	2.5e8 + 1	$2.5e8 + 1_0$	2.5e8 + 1	2.5e8 + 1	2.5e8 + 1	$2.5e8 - 29_{2.5e8-29}$
C3		1	1_0	1	1	1	1_1

Entries show the cycle at which the model deviates from correct behaviour, with 0 for the initialisation cycle and N if for the purpose of the problem, there is no deviation. If a warning is issued, the cycle at which this occurred appears as a subscript.

and Table 2, for those techniques that do. Table 3 shows the parameters used for each problem. In Table 1, each result is given as a pair d_w, where w is the cycle at which the method gives a warning of potential problems with floating point errors, and does not appear if no warning is given; and for each problem, d is the cycle at which the method shows a difference in behaviour in terms of the sequence of donations or the agents on the schedule-list, or 'N' if no difference in behaviour is observed (except that in Illustration 2, d is 'N' if the model does not diverge from the correct behaviour during initialisation – see Fig. 7). Initialisation is represented using cycle number zero. The control is the default behaviour using double precision for all floating point variables and no techniques aimed at coping with floating point errors.

The results in Table 2 are given as a range of configuration parameter settings of the technique for which the model behaves as though it was working with real numbers, or 'None' if no such settings are possible. For tolerance windows, the results show the range of values for ε that enable correct behaviour. In the case of strings, the configuration parameter is the number of significant base-10 digits to write to the intermediate string. The range is found by testing numbers of significant figures from 1 to 25, and then 50. A 25+ for the maximum of the range implies that both 25 and 50 significant figures gave the correct behaviour. For offsets, the only configuration parameters tested were integer powers of two. Thus a maximum of 2^n means that n is the maximum power of two offset for which the model behaved correctly, and hence although 2^{n+1} definitely does not work, offset values between 2^n and 2^{n+1} may or may not work.

Table 1 shows that all techniques with no parameter deviate from the correct behaviour in at least one Illustration or Counter-example, whilst Table 2 shows

Table 2
Results of three methods of dealing with errors in floating point arithmetic that require a configuration parameter

Problem		Tolerance windows (Section 2.3.2)	Strings (Section 2.3.4)	Offsets (Section 2.3.5)	
				$w_0 = 0$	$w_0 > 0$
I1	$G = 0.5$	[0, 4.5)	[2, 25+]	$[0, 2^{55}]$	$[0, 2^{55}]$
	$G = 0.4$	$[2^{-50}, 3.6)$	[2, 15]	$[2^4, 2^{54}]$	$[2^{-54}, 2^{54}]$
I2	$J = 700$	[0, 5.59e3]	[2, 25+]	$[0, 2^{65}]$	$[0, 2^{65}]$
	$J = 70$	[4.45e−16, 5.60e2]	[2, 15]	$[2^4, 2^{62}]$	$[0, 2^{62}]$
I3		None	None	None	None
I4	$SUA_{exc\text{-}total}$	[1.78e−14, 0.7)	[3, 15]	$[2^7, 2^{51}]$	$[2^{10}, 2^{51}]$
	$SUA_{inc\text{-}total}$	[1.07e−14, 0.7)	[3, 15]	$[2^9, 2^{51}]$	$[2^9, 2^{51}]$
	$SUA_{exc\text{-}mean}$	[2.89e−15, 8.75e−2)	[3, 15]	$[2^7, 2^{48}]$	$[2^7, 2^{48}]$
	$SUA_{inc\text{-}mean}$	[2.89e−15, 7.78e−2)	[3, 15]	$[2^7, 2^{48}]$	$[2^7, 2^{48}]$
C1	SSA_{total}	[0, 0.5)	[4, 25+]	$[0, 2^{51}]$	$[0, 2^{51}]$
	SSA_{mean}	[0, 5.56e−2)	[4, 25+]	$[0, 2^{48}]$	$[0, 2^{48}]$
C2	$G = 0.5, j = 2.5e8$	[0, 4.5)	[10, 25+]	$[0, 2^{55}]$	$[0, 2^{55}]$
	$G = 0.4, j = 2.5e8$	None	[10, 15]	None	None
C3		[1.60e−16, 1)	None	$[2^3, 2^{53}]$	n/a

Table 3
Summary of model parameters used in the various problems, where $\lceil x \rceil$ is the smallest integer that is not less than x

Problem	$x \times y$	J	w_0		Agent class	G
			Offset			
I1	3×3	$9G$	0	$\lceil G \rceil$	SSA	Varies
I2	10×10	70 or 700	0	1 or 7	SSA	Any exact
I3	2×1	1	0	1	PSA	0.75
I4	10×10	70	0	1	$\text{SUA}_{\text{various}}$	0.7
C1	20×20	1000	0	3	$\text{SUA}_{\text{various}}$	0.5
C2	3×3	$9Jg$	0	$\lceil jG \rceil$	SSA	Varies
C3	3×3	1	0	0	PSA	1

Note that offsets use a different value for the initial wealth than other approaches, and that Counter-example 3 involves a different initialisation process than the other problems. Also, Counter-example 2 uses a cut down version of the software since j is large, at 250,000,000.

that all techniques with a parameter setting have at least one parameter value for each Illustration or Counter-example that causes the model to behave incorrectly. Thus, none of the techniques guarantee correct behaviour. However, interval arithmetic always provides a warning before or during the cycle at which the model deviates from correct behaviour. Further, for those Counter-examples and Illustrations where a configuration parameter exists that enables correct behaviour, tolerance windows are effective for $1.78 \times 10^{-14} \leq \varepsilon < 0.0556$, offsets for $10 \leq \log_2 M \leq 48$, and strings when the number of significant figures is between 10 and 15.

In Illustration 1, when $G = 0.4$, the Kahan summation algorithm is able to achieve termination and convergence without deviating from the correct behaviour observed when $G = 0.5$. None of the other techniques are able to guarantee this, though the parameterised techniques in Table 2 are able to achieve the correct behaviour for a wide range of parameter settings. The Kahan summation algorithm also works well in Illustration 2 when $J = 70$; and in Cygwin on an Intel PC where extended precision uses 96-bit floating point numbers (according to the size of operator), the model behaves correctly, though not in Solaris on a Sparc box, with 128-bit extended precision. Interval arithmetic also behaves correctly here, though it issues a warning during the initialisation cycle.

In Illustration 3, extended precision is effective in prolonging the length of time the simulation can be run before underflow occurs, something that is obviously not achievable by any of the techniques requiring a parameter. Again, a slight difference is observed between the extended precision on an Intel chip and a Sparc chip. This difference is less than might be expected given the latter uses an extra 32 bits in its extended precision format.

In Illustration 4, the parameterised techniques all have more than one value for the parameter that enables correct behaviour in all four agent classes, whilst of the non-configurable techniques, interval arithmetic runs for the longest period before incorrect behaviour is observed, though it issues a warning during initialisation. For tolerance windows, correct behaviour is observed in all four classes for $1.78 \times 10^{-14} \leq \varepsilon < 0.0778$, for strings, any number of significant figures from 3 to 15 is effective, whilst for offsets when $w_0 = 0$, the same is achieved when $9 \leq \log_2 M \leq 48$, with the lower bound increasing to 10 when $w_0 > 0$. Using a non-zero initial wealth increases the range of values for M that produce correct behaviour in Illustrations 1 and 2, but has no effect in Counter-example 1, and decreases it in Illustration 4. Using an initial wealth greater than or equal to the ticket price to ensure that all numbers are positive when using offsets is not therefore always effective in improving the offset technique.

Counter-example 1, aimed at the warning technique in Section 2.3.1 has little effect on the other non-parameterised techniques. For offsets and tolerance windows, the use of SSA_{mean} rather than $\text{SSA}_{\text{total}}$ decreases the range of configuration parameters that produce correct behaviour. The strings technique is the only technique able to generate correct behaviour in Counter-example 2; extended precision is interestingly not effective in this case. Counter-example 3 shows a weakness in the string technique, but the other configurable techniques are able to produce correct behaviour for some of their parameter settings. None of the non-configurable techniques are able to produce correct behaviour for Counter-examples 2 and 3, though interval arithmetic issues a warning in the same cycle as the deviation is observed.

4. Discussion

4.1. Evaluation of techniques

The use of tolerance windows is possibly one of the most popular techniques for coping with floating point arithmetic errors, and they are discussed by Knuth (1998, p. 233), who argues that the additive tolerance windows of the kind used here assume a particular order of magnitude for the operands, and suggests comparing the difference between the operands with a multiple of the larger operand. His suggested comparison operators have been implemented by Theodore Belding in a small package available on the SourceForge website.[10] It is not clear, however, from Knuth's notation, whether he intends the difference between the operands to be computed exactly or in floating point arithmetic, and so these comparison operators were not included here. Assuming he intended floating point arithmetic, in the

[10] http://fcmp.sourceforge.net/.

case of Illustration 1 we found that whilst using $\varepsilon = 1$ allowed correct behaviour with Knuth's comparison operators, $\varepsilon = 0.5$ did not. Thus if our assumption about his intentions is valid, it would seem there is little difference between issues with the tolerance windows here and those of Knuth in terms of choosing values of configuration parameters.

The popularity of tolerance windows is understandable, since they are relatively easy to implement, and work fairly well. Some fairly extreme circumstances are required to generate situations for which no value of ε enables correct behaviour. In Illustration 3, any finite floating point representation will eventually result in underflow, whilst in Counter-example 2, 250 million cycles are used to create a situation in which accumulated floating point error is sufficiently large that tolerance windows cannot work. Models using large numbers of cycles are not unheard of, however. LeBaron et al. (1999) ran the Artificial Stock Market for 250,000 cycles before analysing time series to allow the agents time to learn (p. 1499), and made one run of 1 million cycles (p. 1507).

Whilst tolerance windows, and the other techniques with a configuration parameter, are effective in many cases, they cannot inform the user when they have not worked. In a typical agent-based modelling context this is an issue. A key concept associated with agent-based modelling is that of emergent effects. Although there is no agreed definition of emergence, many authors refer to the element of 'surprise' (Mihata, 1997; Liebrand et al., 1998; Ronald et al., 1999). Whilst the association of surprise with emergence has been argued against (Gilbert, 1996), clearly we are in something of a luxurious position in the examples above: we know what to expect, and when it does not happen, that something is wrong. In general, this may not be the case. If so, then when two configuration parameters for the floating-point handling technique result in different emergent outcomes, there is no way to choose which is correct. Even if all configuration parameters cause the same emergent effect, it is possible that none of them are correct, albeit that some fairly extreme examples were required in CharityWorld to create such a situation.

The Kahan Summation Algorithm is effective in the case of Illustrations 1 and 2, at least for the particular parameters used. This demonstrates the merits of familiarising oneself as far as bearable with the floating point literature in case there are ways of implementing particular mathematical expressions that either eliminate or reduce the impact of rounding error. As Langlois (2004) points out, however, whilst rewriting expressions can reduce error, there is still no guarantee that the result is a good enough approximation.

Of all the techniques demonstrated here, only interval arithmetic and warning on detection of errors make any use of the extra functionality stipulated by the IEEE 754 standard, the former making use of the configurable rounding direction facilities, and the latter the capability to interrogate the flags on the floating point unit that are set when an error occurs. Both are pessimistic, in that they may issue a warning when the model has not yet done anything wrong, but of the two, interval arithmetic is less so. For interval arithmetic, there is the further complication that for non-monotonic functions, the minimum and maximum of the result may not occur at the bounds of the interval. The bounds of interval arithmetic may also over-estimate the potential floating point error of an expression. In some situations it may be possible to ameliorate these drawbacks through careful rewriting of the expressions, but even if it is not, both interval arithmetic and issuing a warning have the critical advantage over the other techniques that they never lull the user into a false sense of security.

If floating point parameters cause so many problems, there might be a temptation to use integers instead, a technique employed by the PS-I simulation platform (Lustick, 2002; Dergachev, 2004, pers. comm.). Whilst this may prove effective for the kinds of model PS-I is used to build, in general the problems with floating point arithmetic derive from using a discrete parameter to represent a non-discrete phenomenon, and using integers instead of floating point numbers can be expected to make matters worse. The integer division operation, for example, is effectively defined to round towards zero. If a parameter involves an accumulation of results of a series of division operations, errors will also accumulate from the missed remainders. With integers, there is also no access to standard mathematical library functions, such as square root, power and trigonometric functions. Floating point arithmetic is the mathematics of approximating continuous phenomena with discrete parameters, and, used carefully with the full facilities provided by IEEE 754, it achieves this remarkably well. Trying to achieve the same using integers would be an act of hubris.

Significantly, the worst performing technique is the control—the case where no remedial action is taken to address floating point issues. Using extended precision (which merely involves changing the datatype of floating point variables) likewise performs poorly, only improving on the control in Illustration 3 through being able to run for longer before underflow occurs. Whilst this might be effective in a case where the experimental design treated more than a certain number of cycles as being infinity, it is clear that ignoring these issues is in general not the best way of ensuring that a computer model is behaving as it is designed to.

4.2. Robustness

It might be argued that a model that is sensitive to floating point errors somehow lacks robustness. Perhaps

it does not matter if an agent takes the 'wrong' decision because of floating point errors, or if it does matter, then there is something wrong with the model. For example, if a variable is very close to the threshold at which an agent takes one course of action rather than another, maybe it should be unimportant which of the two is finally selected. If the model shows systematically different behaviour when run with a floating point valued variable from the way it would behave if reals were used, then it could be that the model is too sensitive to that variable.

There are a number of arguments against this view. Firstly, the way that variables sensitive to floating point arithmetic behave is somewhat different to the way that other sensitive variables in the model might behave. From the example in Illustration 1, if $G = 0.5$, 0.51 or 0.52, the model converges, if 0.53, it does not, but for 0.54, 0.55 and 0.56 it does, then at 0.57, 0.58 and 0.59 it does not. Considering three significant figures, the model converges if $G = 0.502$, 0.504, 0.506 and 0.508, but not 0.501, 0.503, 0.505, 0.507 or 0.509. A real-world system seems unlikely to display this kind of sensitivity, with the exception of fractal basin boundaries. Related to this matter, Corless (1994) discusses confusion between genuine chaotic dynamics in real-world systems and artefacts of floating point arithmetic.

Secondly, a model can be shown mathematically to be completely robust in its behaviour, but such robustness does not transfer to the computer simulation. In the case of CharityWorld, for any G, if $J = jxyG$, we can prove mathematically that the model should converge using SSAs. By putting this constraint on the parameters, the convergence of the model should not be sensitive to G at all. CharityWorld is thus an example of a model arguably designed to be as robust as possible relative to the parameter G in the realm of arithmetic with real numbers. It is surely not reasonable to expect such a model to also be robust in a realm where the associative laws of addition and multiplication, and the distributive law between multiplication and addition no longer necessarily apply (Knuth, 1998, p. 231). It is not even clear whether designing a model to be robust in floating point arithmetic rather than real number arithmetic would be desirable.

As has already been mentioned, errors in data can have a more significant effect on the behaviour of a model than errors in floating point arithmetic. Yet this does not mean that floating point errors can be ignored; neither can it be assumed that floating point errors and data errors can be treated in the same way. Data errors occur once, before the model even begins, whilst floating point errors occur throughout the simulation. Some authors, such as Vignes and La Porte (1974), do argue for treating floating point errors in a probabilistic fashion, much as one might treat data errors. The deliberate introduction of noise to computations is a common modelling practice to assess the robustness of a model. Kahan (1998), however, argues that analyses of floating point errors probabilistically generally require the assumptions that rounding errors are random, weakly correlated, and have a continuous distribution, when in fact, these errors are not random, are often correlated, and often behave like discrete variables. Further, he argues that randomised perturbations of arithmetic operations, by ignoring known theorems about harmless rounding errors, can exaggerate the effects of rounding error without offering any guarantee of correctness (Kahan, 1998).

Finally, it cannot be best practice to allow floating point errors to determine the robustness of a model or sensitivity of variables, not least because it has been shown that errors in floating point arithmetic are not random (Kahan, 1998). For example, in IEEE round-to-nearest-even mode, adding any representable number $2 \leq z < 3.7$ to $[0.3]_f$ will cause the computer to round down.[11] Moreover, there are differences between IEEE 754 compliant platforms in the results delivered for some computations (Anon., 2001). In general, control over assessing the sensitivity of the model should ideally lie entirely with the user, and should not depend on the platform on which the model is run.

4.3. Recommendations to agent-based modellers

There are two key approaches that agent-based modellers can employ to deal with floating point errors. On the design side, work can be done to reduce the element of surprise associated with emergent effects. Though agent-based models are often used in situations that are mathematically intractable, mathematical analysis can complement computer simulation work, as illustrated by authors such as Gotts et al. (2003a) and Brown et al. (2004). Analysis of simplified versions of the full model could be used to generate expectations of the likely output from the computer simulation beforehand, or used afterwards to confirm that a particular emergent effect is consistent with the model design. Mathematical analysis could also be used to identify useful phenomena, such as quantities that should remain constant throughout the simulation

[11] The decimal number 0.3 is the recurring binary fraction 0.0 1001 1001… In double precision floating point this is stored as $1.0011\ldots0011 \times 2^{-2}$ (where the ellipsis is replaced with eleven lots of 0011). Both z and $[z + [0.3]_f]_f$ have exponent 1, meaning that the last three bits of $[0.3]_f$ cannot be included in the result. These last three bits are 011, less than half the unit in the last place of the result, and therefore not sufficient to cause an upward rounding.

and could be displayed to confirm that they are behaving correctly.

On the programming side, an awareness of floating point arithmetic issues and the limitations of techniques for dealing with them can prevent careless mistakes. Some of the techniques above can be combined, such as interval arithmetic with extended precision, or tolerance windows with carefully chosen expressions that minimise rounding error. It is also worth reviewing the numerical analysis literature before implementing standard mathematical expressions or procedures. In general, however, as Higham (2002, p. 26) points out, "There is no simple recipe for designing numerically stable algorithms." Nevertheless, the following are a few of the guidelines he does offer (p. 27):

- Avoid subtracting quantities contaminated by error if possible.
- Minimise the magnitude of intermediate quantities in expressions relative to the final solution.
- Look for different formulations of a computation that are mathematically equivalent but more numerically stable.
- Avoid overflow and underflow where possible.
- Look at the numbers generated during a simulation.

5. Conclusion

We have demonstrated that floating point errors should not be ignored in computer implementations of agent-based models, through demonstrating a simple agent-based model with an emergent outcome that is affected by floating point errors during the simulation. Although this is a worst-case scenario, it should serve as a warning.

Of the various remedies to the problem of errors in floating point arithmetic explored here, none of them guarantee that the principles on which the model was designed are rigorously adhered to throughout any particular simulation run. Some remedies at least indicate when these principles have not been rigorously adhered to, and all involve extra computation that will slow simulation times. However, the belief underlying the methods presented in this paper is that it is better to have a slow model that can tell you whether floating point errors have affected the correctness of a run, than a fast model that cannot. Of those approaches studied, interval arithmetic seems the most promising – it is at least the safest. Knuth (1998) is also in favour of interval arithmetic: "The prospects for effective use of interval arithmetic look very good, so efforts should be made to increase its availability and to make it as user-friendly as possible" (p. 241). Perhaps programming languages should include data-types for interval arithmetic. They should certainly make available all of the functionality stipulated by the IEEE 754 standard.

Future work could look at the extent to which changing the way floating point expressions are coded can affect the statistical signatures from the model; whether focusing on runs parameterised such that they have no floating point errors (should this be possible) introduces a bias when using a model; the extent to which interval arithmetic can be effectively applied in other agent-based models; and on other more elaborate techniques for doing arithmetic with real numbers in computers, such as lazy arithmetic (Michelucci and Moreau, 1997).

Acknowledgements

This work was funded by the Scottish Executive Environment and Rural Affairs Department. The authors are also grateful to Joe Darcy for some useful help and advice.

Appendix 1

If $j = J/(xyg)$ is a natural number, then simulations involving SSAs will eventually converge with probability 1.

Proof: Since the initial wealth (the same for every agent) is a reference point which is not relevant for the proof, we will assume for clarity that it is zero.

We can think of the situation where $j = J/(xyg)$ is a natural number as setting up an environment where one SSA (F) has $(xy-1)j$ quanta of positive wealth (which will be called 'pecunions') and the other $(xy-1)$ SSAs have j quanta of negative wealth (or 'impecunions'). One pecunion has value g whilst an impecunion has value $-g$.

When an SSA with positive wealth donates to an SSA with negative wealth, a pecunion from the donating SSA annihilates with an impecunion from the recipient SSA. When an SSA with positive wealth donates to any other SSA, the pecunion from the donating SSA can be thought of as moving from the donor to the recipient. It is also possible for an SSA with zero or negative wealth to make a donation, since the SSA-supply-awareness-condition pertains only to the local average wealth. This would effectively imply that an impecunion from the recipient SSA moves to the donating SSA. Note that the SSA-supply-awareness-condition and SSA-supply-offering-condition ensure that an SSA cannot donate to a neighbour with the same or greater wealth. Table A1 summarises these results.

Table A1
Relationship between the wealths of donating and recipient SSAs, and their consequences for pecunions and impecunions

		Recipient simple supply agent		
		Wealth < 0	Wealth $= 0$	Wealth > 0
Donating simple supply agent	Wealth < 0	Impecunion movement	Not possible	Not possible
	Wealth $= 0$	Impecunion movement	Not possible	Not possible
	Wealth > 0	Pecunion–impecunion annihilation	Pecunion movement	Pecunion movement

In the general case, the initial wealth should be substituted for zero in the row and column headings.

Each donation can thus be seen as involving the movement of a pecunion or impecunion, or the annihilation of a pecunion–impecunion pair – the last of these bringing the simulation closer to termination. The simulation converges when all pecunions have annihilated with impecunions.

The proof of convergence, which is given somewhat informally, is based on the fact that a situation where there is a certain number p of pecunion–impecunion pairs can be seen as a discrete-time absorbing Markov chain with a finite state-space (Grinstead and Snell, 1997, p. 415). An absorbing Markov chain is a Markov chain that has at least one absorbing state (a state that cannot be left) and from every non-absorbing state it is possible to reach an absorbing state in a finite number of steps. We define the only absorbing state as the situation where an annihilation of wealth quanta takes place (i.e. the number of pecunion–impecunion pairs decreases). The rest of the possible (transient) states are characterised by the following properties:

- A certain distribution of pecunions and impecunions over the grid. The number of pecunions in each cell (i.e. owned by each agent) is bounded by p whereas the number of impecunions in each cell is bounded by the minimum of p and j.
- A schedule-list of potential donors (the number of possible states for which is bounded by $(2^{xy}-1)$ and the order of which is not relevant since agents are selected from the schedule-list at random).

Since each property has a bounded integral number of possible values, the number of possible transient states is finite. Each of these possible transient states satisfies two constraints:

- The total number of pecunion–impecunion pairs is p.
- Any SSA that satisfies the supply-awareness-condition is on the schedule-list (the fact that the stepping agent and all its neighbours are rescheduled after the redistribution cycle ensures that no SSA satisfying the supply-awareness-condition is left off the schedule-list).

Any situation that can arise in a simulation where there are p pecunion–impecunion pairs will correspond to one of these transient states. Clearly, for any of these transient states we could draw at least one direct route for (at least) one pecunion to meet (at least) one impecunion, i.e. a route towards the absorbing state (Fig. A1).

This route has a certain probability of happening since any SSA in the route would be able to donate (and therefore would be on the list) and they have a certain probability of being chosen since agents are selected from the schedule-list at random. Therefore it is always possible to reach the absorbing state from any transient state in a finite number of steps.

As the number of stages approaches infinity in a finite absorbing chain, the probability of being in a non-absorbing state approaches 0 (Grinstead and Snell, 1997, p. 417) – which in our case, with only one absorbing state, means that the probability of reaching the absorbing state approaches one. Therefore, for any number p of pecunion–impecunion pairs, the probability of a pecunion and an impecunion annihilating approaches 1 as the number of stages approaches infinity. This effectively means that any simulation (which starts with $(xy-1)j$ pecunion–impecunion pairs) will eventually terminate ($p = 0$) with probability 1. □

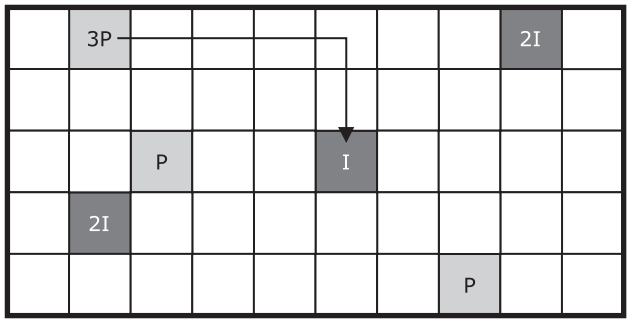

Fig. A1. A possible distribution of pecunions (P) and impecunions (I) on a grid. The arrow shows one of many possible routes for a pecunion to meet an impecunion.

References

Abrams, S.L., Cho, W., Hu, C.-Y., Maekawa, T., Patrikalakis, N.M., Sherbrooke, E.C., Ye, X., 1998. Efficient and reliable methods for rounded interval arithmetic. Computer-Aided Design 30 (8), 657–665.

Alefeld, G., Herzberger, J., 1983. Introduction to Interval Computation. Academic Press, New York, NY.

Anon., 2001. Differences among IEEE 754 implementations. In Sun Microsystems Numerical Computation Guide, July 2001, pp. 237–251. Available from: http://docs.sun.com/db/doc/806-7996.

Axelrod, R., 1997. Advancing the art of simulation in the social sciences. In: Conte, R., Hegselmann, R., Terna, P. (Eds.), Simulating Social Phenomena. Springer, Berlin, pp. 21–40.

Booch, G., Rumbaugh, J., Jacobson, I., 1999. The Unified Modeling Language User Guide. Addison-Wesley, Boston, MA.

Bousquet, F., Bakam, I., Proton, H., Le Page, C., 1998. CORMAS: common-pool resources and multi-agent systems. Lecture Notes in Artificial Intelligence 1416, 826–838.

Bousquet, F., Le Page, C., 2004. Multi-agent simulations and ecosystem management: a review. Ecological Modelling 176 (3–4), 313–332.

Brown, D.G., Page, S.E., Riolo, R., Rand, W., 2004. Agent-based and analytical modeling to evaluate the effectiveness of greenbelts. Environmental Modelling and Software 19 (12), 1097–1109.

Coe, T., Mathisen, T., Moler, C., Pratt, V., 1995. Computational aspects of the Pentium affair. IEEE Computational Science & Engineering 2 (1), 18–30.

Corless, R.M., 1994. What good are numerical simulations of chaotic dynamical systems? Computers & Mathematics with Applications 28 (10–12), 107–121.

Darcy, J.D., 1998. Borneo 1.0.2: adding IEEE 754 floating point support to Java. MS thesis, Computer Science Division, University of California at Berkeley. Available from: http://www.sonic.net/~jddarcy/Borneo/borneo.pdf.

Darcy, J.D., 2 May 2003 and 29 July 2003. Personal communications.

Dergachev, V., 31 May 2004. Personal communication.

Dybvig, R.K., 2003. The Scheme Programming Language. third ed. MIT Press, Cambridge, MA, Available from: http://www.scheme.com/tspl3/.

Edmonds, B., Hales, D., 2003. Replication, replication and replication: some hard lessons from model alignment. Proceedings, Model to Model International Workshop, GREQAM, Vielle Charité, 2 rue de la Charité, Marseille, France. 31 March – 1 April, 2003, 133–150.

Epstein, J.M., Axtell, R., 1996. Growing Artificial Societies: Social Science from the Bottom Up. Brookings Institution Press, Washington, DC.

Fernandez, J.-J., Garcia, I., Garzon, E.M., 2003. Floating point arithmetic teaching for computational science. Future Generation Computer Systems 19 (8), 1321–1334.

Feuillette, S., Bousquet, F., Le Goulven, P., 2003. SINUSE: a multi-agent model to negotiate water demand management on a free access water table. Environmental Modelling & Software 18, 413–427.

Fox, L., 1971. How to get meaningless answers in scientific computation (and what to do about it). IMA Bulletin 7 (10), 296–302.

Gilbert, N., 1996. Holism, individualism and emergent properties. In: Hegselman, R., Mueller, U., Troitzsch, K.G. (Eds.), Modelling and Simulation in the Social Sciences from the Philosophy of Science Point of View. Kluwer, pp. 1–12 (Chapter 1).

Gleick, J., 1988. Chaos: Making a New Science. Heinemann, London, UK.

Goldberg, D., 1991. What every computer scientist should know about floating point arithmetic. ACM Computing Surveys 23 (1), 5–48. Reproduced in Sun Microsystems Numerical Computing Guide (Appendix D). Available from: http://portal.acm.org/citation.cfm?doid=103162.103163.

Gosling, J., Joy, B., Steele, G., Bracha, G., 2000. The Java Language Specification. second ed. Addison-Wesley, Available from: http://java.sun.com/docs/books/jls/index.html.

Gotts, N.M., Polhill, J.G., Adam, W.J., 2003a. Simulation and analysis in agent-based modelling of land use change. Presented at the First Conference of the European Social Simulation Association, Groningen, The Netherlands, 18–21 September 2003, Conference proceedings. Available from: http://www.uni-koblenz.de/~kgt/ESSA/ESSA1/proceedings.htm.

Gotts, N.M., Polhill, J.G., Law, A.N.R., 2003b. Aspiration levels in a land use simulation. Cybernetics & Systems 34 (8), 663–683.

Gotts, N.M., Polhill, J.G., Law, A.N.R., 2003c. Agent-based simulation in the study of social dilemmas. Artificial Intelligence Review 19, 3–92.

Grinstead, C.M., Snell, J.L., 1997. Introduction to Probability: Second Revised Edition. American Mathematical Society, Available from: http://www.dartmouth.edu/~chance/teaching_aids/books_articles/probability_book/book.html.

Hare, M., Deadman, P., 2004. Further towards a taxonomy of agent-based simulation models in environmental management. Mathematics and Computers in Simulation 64, 25–40.

Harper, S.J., Westervelt, J.D., Shapiro, A.-M., 2002. Management application of an agent-based model: control of cowbirds at the landscape scale. In: Gimblett, H.R. (Ed.), Integrating Geographic Information Systems and Agent-Based Modeling Techniques for Simulating Social and Ecological Processes. Oxford University Press, New York, NY, pp. 105–123.

Higham, N.J., 2002. Accuracy and Stability of Numerical Algorithms, second ed. SIAM, Philadelphia, USA.

Hoffmann, M., Kelley, H., Evans, T., 2002. Simulating land-cover change in South-Central Indiana: an agent-based model of deforestation and afforestation. In: Janssen, M.A. (Ed.), Complexity and Ecosystem Management: The Theory and Practice of Multi-Agent Systems. Edward Elgar, Cheltenham, UK, pp. 218–247.

IEEE, 1985. IEEE Standard for Binary Floating-Point Arithmetic, IEEE 754-1985. Institute of Electrical and Electronics Engineers, New York, NY.

Izquierdo, L.R., Gotts, N.M., Polhill, J.G., 2004. Case-based reasoning, social dilemmas, and a new equilibrium concept. Journal of Artificial Societies and Social Simulation 7 (3), Available from: http://jasss.soc.surrey.ac.uk/7/3/1.html.

Johnson, P.E., 2002. Agent-based modeling: what I learned from the artificial stock market. Social Science Computer Review 20, 174–186.

Kahan, W., 1965. Further remarks on reducing truncation errors. Communications of the ACM 8 (1), 40.

Knuth, D.E., 1998. The Art of Computer Programming, third ed. Seminumerical Algorithms, vol. 2. Addison-Wesley, Boston, MA.

Kripke, S., 1963. Semantical considerations on modal logic. Acta Philosophica Fennica 16, 83–94.

Langlois, P., 2004. More accuracy at fixed precision. Journal of Computational and Applied Mathematics 162 (1), 57–77.

Lansing, J.S., Kremer, J.N., 1993. Emergent properties of Balinese water temple networks: coadaptation on a rugged fitness landscape. American Anthropologist 95 (1), 97–114.

Lawson, C.L., Hanson, R.J., 1995. Solving Least Squares Problems. SIAM, Philadelphia, PA.

LeBaron, B., Arthur, W.B., Palmer, R., 1999. Time series properties of an artificial stock market. Journal of Economic Dynamics & Control 23, 1487–1516.

Liebrand, W.B.G., Nowak, A., Hegselbaum, R., 1998. Computer Modeling of Social Processes. Sage Publications, London, UK.

Luke, S., Catalin Balan, G., Panait, L., Cioffi-Revilla, C., Paus, S., 2003. MASON: a Java multi-agent simulation library. Proceedings

of the Agent 2003 Conference. Available from: http://cs.gmu.edu/~eclab/projects/mason/papers/Agent2003.6.pdf.

Lustick, I., 2002. PS-I: a user-friendly agent-based modeling platform for testing theories of political identity and political stability. Journal of Artificial Societies and Social Simulation 5 (3), Available from: http://jasss.soc.surrey.ac.uk/5/3/7.html.

McCullough, B.D., Vinod, H.D., 1999. The numerical reliability of econometric software. Journal of Economic Literature 37 (2), 633–665.

Michelucci, D., Moreau, J.M., 1997. Lazy arithmetic. IEEE Transactions on Computers 46 (9), 961–975.

Mihata, K., 1997. The persistence of "Emergence". In: Eve, R.A., Horsfall, S., Lee, M.E. (Eds.), Chaos, Complexity and Sociology. Sage Publications, Thousand Oaks, CA, pp. 30–38 (Chapter 3).

Moss, S., 2001. Competition in intermediated markets: statistical signatures and critical densities. Centre for Policy Modelling Report No. 01-79. Available from: ftp://cfpm.org/pub/papers/statsig2a.pdf.

Moss, S., Gaylard, H., Wallis, S., Edmonds, B., 1998. SDML: a multi-agent language for organizational modelling. Computational & Mathematical Organization Theory 4 (1), 43–69.

Pahl-Wostl, C., 2005. Information, public empowerment, and the management of urban watersheds. Environmental Modelling and Software 20 (4), 457–467.

Polhill, J.G., Gotts, N.M., Law, A.N.R., 2001. Imitative versus nonimitative strategies in a land use simulation. Cybernetics & Systems 32 (1–2), 285–307.

Polhill, J.G., Izquierdo, L.R., Gotts, N.M., 2005. The ghost in the model (and other effects of floating point arithmetic). Journal of Artificial Societies and Social Simulation 8 (1), Available from: <http://jasss.soc.surrey.ac.uk/8/1/5.html>.

Ronald, E.M.A., Sipper, M., Capcarrère, M.S., 1999. Design, observation, surprise! A test of emergence. Artificial Life 5 (3), 225–239.

Rouchier, J., Bousquet, F., Barreteau, O., Le Page, C., Bonnefoy, J.-L., 2001. Multi-agent modelling and renewable resource issues: the relevance of shared representations for interacting agents. In: Moss, S., Davidsson, P. (Eds.), Multi-Agent-Based Simulation: Second International Workshop MABS 2000. Springer, Berlin, pp. 181–197.

Stallman, R.M., 2001. Using and Porting the GNU Compiler Collection. Free Software Foundation, Boston, MA.

Ullrich, C., von Gudenberg, J.W., 1990. Different approaches to interval arithmetic. In: Wallis, P.J.L. (Ed.), Improving Floating Point Programming. John Wiley & Sons, Chichester, UK (Chapter 5).

Vignes, J., La Porte, M., 1974. Error analysis in computing. Information Processing 74, 610–614.

Wallis, P.J.L., 1990a. Basic concepts. In: Wallis, P.J.L. (Ed.), Improving Floating Point Programming. John Wiley & Sons, Chichester, UK (Chapter 1).

Wallis, P.J.L., 1990b. Machine arithmetic. In: Wallis, P.J.L. (Ed.), Improving Floating Point Programming. John Wiley & Sons, Chichester, UK (Chapter 2).

Winkler, J.R., 2003. A statistical analysis of the numerical condition of multiple roots of polynomials. Computers & Mathematics with Applications 45 (1–3), 9–24.

Internet references

Collier, N., 2000. RePast. Available from: <http://repast.sourceforge.net/>.

Kahan, W.,1998. The improbability of probabilistic error analyses for numerical computations. Originally presented at the UCB Statistics Colloquium, 28 Feburary 1996. Revised and updated version (10 June 1998, 12:36 version used here). Available from: <http://www.cs.berkeley.edu/~wkahan/improber.pdf>.

Kahan, W., Darcy, J.D., 2001. How Java's floating point hurts everyone. Originally presented at the ACM 1998 Workshop on Java for High-Performance Network Computing, Stanford University. Revised and updated version (5 Nov 2001, 08:26 version used here). Available from: <http://www.cs.berkeley.edu/~wkahan/JAVAhurt.pdf>.

Langton, C., Burkhart, R., Minar, N., Askenazi, M., Ropella, G., Daniels, M., Lancaster, A., Lee, I., Jojic, V., 1995. Swarm. Available from: <http://wiki.swarm.org/>.

Laski, Z., 2004. Available from: <http://gcc.gnu.org/ml/gcc/2004-02/msg01202.html>.

Parker, M., 1998. Ascape. Available from: <http://www.brook.edu/es/dynamics/models/ascape>.

Sun Microsystems, 2001. Numerical Computation Guide. Lincoln, NE: iUniverse Inc. Available from: <http://www.docs.sun.com/db/doc/806-7996>.