The Impact of Programming Paradigms on the Efficiency of an Individual-based Simulation Model

David J. Barnes, Tim R. Hopkins

Computing Laboratory, University of Kent at Canterbury, Canterbury, Kent CT2 7NF, UK

Abstract

Individual-based models are a popular technique for simulating a wide range of ecological systems. However, to be successful, they must not only deliver an accurate representation of the system they are seeking to model, but must do so using viable amounts of computing resource. Models involving very large numbers of individuals will tend to have large memory requirements, while the need to vary parameter settings over multiple runs means that processor requirements must be kept within reasonable bounds. In order to address the issue of resource requirements, we assess the impact of using different programming paradigms for the implementation of an individual-based models. We do this by looking in detail at a number of implementations of a simulation of the spread of Barley Yellow Dwarf Virus. The model considers explicitly each individual plant and aphid, therefore it requires special care to reduce the amount of storage used whilst still producing a computationally efficient code. We present implementations of the model in both imperative and object-oriented programming languages, particularly noting aspects relating to ease of implementation and run-time performance. Finally, we attempt to quantify the cost of some of the decisions made in terms of their memory and processor time requirements.

Key words: Barley Yellow Dwarf Virus; Simulation; Programming Languages; Java; C++; Fortran 95.

1 Introduction

The increased power and availability of computers in recent years has led to the development of two new types of ecological simulation model: individual-based models and spatially-explicit models. In the former, each biological entity in the model is simulated at the level of the individual organism rather than the population level (see DeAngelis and Gross, 1992; Judson, 1994; Kawata and Toquenaga, 1994, for reviews). Among other things, this allows individual variation to be simulated (Huston et al., 1988).

The number of individual-based models reported in the literature has grown dramatically over the last decade (Grimm, 1999). Such models inevitably place significant demands upon the available computing resource; often requiring large amounts of both memory and processor time for realistic simulations. Studies of fish communities, for instance, could easily require of the order of 10^{12} individuals to be modelled (Shin and Cury, 2001). Even when the number of individuals being modelled is relatively small, as in Ahearn et al. (2001), run time must be kept within tractable limits to support the need for multiple runs with varying parameter combinations (Cowan et al., 1996). Where pragmatic implementation issues are considered within the literature, however, the focus tends to be either on the ease of modelling — for which the object-oriented paradigm is often adopted (Ahearn et al., 2001; Shin and Cury, 2001) — or simply keeping the run time within reasonable limits. With rare exceptions (Congleton et al., 1997), there is little acknowledgement that memory requirements play a significant part in the viability of large simulations (Scheffer et al., 1995), and have a direct impact upon run times through the use of virtual memory.

In the remainder of this paper we present an extended example to investigate how the use of the two main programming paradigms, procedural and object-oriented, affects the efficiency of implementation. The example is based on a reasonably simple but realistic model for the spread of Barley Yellow Dwarf Virus (BYDV). BYDV is an economically important pest of cereal crops and wild grasses worldwide (Power, 1996). It causes yellowing of the leaves and often results in significant yield losses in cereals. The approach used here is to model the individual plants in a cereal crop and the aphids that infect them. Such a model might be used in an attempt to obtain a better understanding of the population dynamics of the aphid and the mechanisms involved in the spread of the virus. However, that is not our principal aim in developing this simulation.

In section 3 we look in detail at the steps involved in the simulation and follow that with a careful consideration of the general storage requirements of the model. In section 5 we discuss the model's implementation in the structured programming language Fortran 95 (ISO/IEC, 1997). There we consider a number of ways in which we can reduce the amount of memory required to store data and discuss some of the implications these may have on the resultant run times. Section 6 provides some timing and performance measurements obtained from running the resultant Fortran 95 implementation on a medium and a large simulation.

A model defined primarily in terms of plants and aphids strongly suggests the use of an object-oriented design process (Booch, 1994) as an alternative to a structured programming approach. Section 7 discusses the implementation of the model in the Java (Arnold et al., 2000) and C++ (Stroustrup, 1997) programming languages along these lines for a small simulation. Because of the very large amounts of data involved, the naive approach used with the pure object-oriented implementations is doomed to failure due to storage considerations.

We conclude the paper with an evaluation of the alternative implementations.

2 Barley Yellow Dwarf Virus

Barley Yellow Dwarf Virus infects a wide range of grasses worldwide, including cereals. It is one of the most economically important diseases of grasses (Power, 1996). The virus can only be transmitted from one plant to another by aphids such as *Rhopalsiphum padi* (L.) feeding on the phloem of the host plant (Power, 1996). BYDV does not replicate within the aphid.

The disease is spread in two ways by the aphids (Morgan, 1989)

- Primary infection takes place when infective, winged (alate) aphids migrate onto the crop from reservoir populations of the virus elsewhere;
- Secondary infection results from dispersal of the offspring of the migrant aphids. Note that these aphids first have to acquire infection by feeding on an infected plant before they can pass it on to another plant.

Control measures aimed at halting the spread of BYDV are targeted at the secondary infection phase by reducing or eliminating the spread of BYDV by the offspring of the infective immigrant aphids. This is achieved by applying an aphicide spray to control the aphids. Comparatively little is known about the factors which determine the rate at which this secondary spread of the disease advances through the crop. However, a recent simulation model developed by McElhany et al. (1995) has indicated that factors such as the preference of the aphid vector for diseased or healthy hosts plants can be important in determining the rate of secondary spread. Other models of the spread of BYDV have been more or less successful in predicting the spatial and temporal dynamics of the disease and the factors which influence its spread (Morgan, 1989; Morgan et al., 1988).

3 The Simulation Model

We attempt to simulate the spread of BYDV in a cereal field by keeping track of the position and state of the individual aphids and by considering their effect on individual barley plants. A detailed description of the model may be found in Hopkins and Morse (1997). In brief, the simulation consists of a sequence of days during which the following events take place in the order indicated:

(1) Immigration of aphids: over a defined period of consecutive days at the beginning of the simulation period a predetermined number of winged aphids are randomly placed on the individual barley plants in the field.
(2) Based on given daily temperature data, the development and reproductive rates for all the aphids in the simulation are calculated.
(3) For each aphid in the field,
   (a) Its age is updated based on the daily development rate calculated in step 2 above.
   (b) Based on the daily reproductive rate, newly born aphids appear on the same plant as their parents.
   (c) Its position may change; movement occurs when a given probability threshold is exceeded. The current simulation allows a choice of two dispersal models
      (i) purely random: the aphid is transported to a random point in the field.
      (ii) movement is restricted to a nearest neighbour move with probabilities chosen to reflect the tendency of aphids to move between plants in the same row in a cereal field (Power, 1996). This movement preference reflects the fact that inter-plant spacing is closer within rows than it is between rows of plants.
(4) The virus states of the aphid and the plant on which the aphid is feeding are updated. An infected aphid always feeds upon its current plant, passing on the infection. A healthy aphid is always infected if it settles on an infected plant.

For the simulation reported a square grid of p plants in each direction (i.e., p^2 plants in total) with $0.002p^2$ immigrant aphids on each of the first four days was used. The immigrant aphids were all winged aphids aged 0.5 with a probability of 0.1 of being infected with BYDV. The simulation took place between days 70 and 100.

The timings reported were obtained on a SUN Ultra 60, Model 2450 with dual 450MHz processors and 512Mb of memory. Timing information was obtained using the machine as single user although it was connected to a local network. For Fortran 95 the built-in intrinsic function *system_clock* was used; for

C++ *clock_gettime* was used; and for Java *currentTimeMillis* was used. Clock resolution of better than 10% is repeatable.

4 General Storage Requirements of the Model

Hopkins and Morse (1997) argued that a realistic model might require between 27 Mbytes and 270 Mbytes of storage, depending on the size of the model. Their argument ran as follows. Assume a planting density of 300 plants/m^2 and a square field plot which is of the order of 200m in each direction. Further assume that there is an immigrant aphid population of between two and twenty thousand aphids per day during the immigration period. For 10^6 plants they specified an immigrant population of 2000 aphids/day for four days and for 10^7 plants, 20000 aphids/day for four days. The resultant populations after 30 days were 8×10^5 and 8×10^6 aphids respectively.

The simulation model must maintain data on the following variables:

- for each individual aphid, its
  - · current age (a real value in the range [0,1])
  - · life stages (one of newly born, wingless, winged or dead)
  - · position in the field (x, y coordinate – a pair of integers)
  - · BYDV status (one of infected, incubating, or uninfected)
  - · incubation period (the number of days the virus has been incubating – used to update the BYDV status from incubating to infected).
- and, for each individual plant, its
  - · BYDV status (one of infected, incubating, or uninfected)
  - · incubation period (the number of days before a plant, bitten by an infected aphid, itself becomes infected – used to update the BYDV status from incubating to infected).

A naive storage scheme might use five integers (one each for the life stage, the BYDV status, the x-coordinate, the y-coordinate and the incubation period) and a real number (the age) for each aphid and two integers for each plant (the BYDV status and the incubation period). (The coordinate information for the plants being implicitly obtained using a two dimensional array.) This gives conservative storage requirements of

10^6 plants $+8 * 10^5$ aphids $=$ 27 Mbytes
10^7 plants $+8 * 10^6$ aphids $=$ 270 Mbytes

where both integers and reals are assumed to require 32 bits. In an object-oriented programming language, the space requirements for an object are supplemented by the need to identify an object's runtime type in order to implement its correct behaviour. Typically, this will add an overhead of a further 4 bytes per object or 72Mbytes for the case of 10^7 plants, above.

There are several ways in which the naive approach can be further improved upon to reduce the memory requirements of the model (Hopkins and Morse, 1997). For example,

(1) The plant data can be directly mapped into a one dimensional array because the coordinates of each plant are only required for nearest neighbour calculations and any final visualisation of the data.
(2) The infection status of the plants may be stored as an array of bits and the coordinates of the incubating plants (those plants which have been infected but are not themselves infectious) stored as a circular list of linked lists. On completing the incubation period these plant lists would be used to update the bit array and the storage is reused.

In the following section we discuss these improvements in the context of an implementation in a structured programming language, Fortran 95. Further details of how these data structures were used may be found in Hopkins and Morse (1997). The storage space required is then reduced to $10^6/8+8n_1$ bytes for 10^6 plants and $10^7/8+8n_2$ bytes for 10^7 plants where n_1 and n_2 are the maximum number of plants incubating at any one time. With the model used here the maximum number of plants being stored was around 0.25% of the total number of plants.

5 A Fortran 95 Implementation

Fortran is generally regarded amongst the scientific programming community as being one of the languages of choice if highly efficient code is required; the long history of the language and the wealth of experience in writing optimising compilers has led to most commercial Fortran systems generating extremely good code.

Recent years have seen the development of Fortran 90 and Fortran 95 in an attempt to keep Fortran, at least partially, up-to-date with new programming language trends.

In Fortran 77 the array space required to store the aphid and plant data needs to be declared at compile time and, at best, separate arrays used to store the real and integer data associated with each plant and aphid. The alternative approach in Fortran 95 is to store teh associated aphid data as records (user defined types) where each record contains the age, life stage, position, BYDV status and incubation period of a single aphid. This approach is similar to the aggregation that an aphid class would provide in an object-oriented language.e availability of pointers then allows us to construct dynamically sized sets of these objects (for example, linked lists) which relieves the user of having to estimate the maximum size of the aphid population. We note here that the allocatable array available in Fortran 95 is not sufficiently flexible to provide an efficient alternative since it is not possible to extend such arrays once created; rather a new, larger array needs to be created, the old data copied over and the original space returned to the heap.

In addition, the module facility available in the new Fortran standard allows the use of information hiding that parallels the class encapsulation features of object-oriented languages. This means that it is possible to build the simulation software so that details of the underpinning data structures are hidden from higher levels. Indeed the effects of any change to the data structure are localised within a single module and, since all access to this module is at a subroutine level, no changes are required elsewhere in the simulation code.

For example, the aphid-control module contains a number of publically callable routines which allow

(1) an aphid's record to be unpacked into its components and repacked into a, possibly compressed, record. Fortran 95 defines new intrinsic functions for the simple manipulation of bit strings.
(2) newly born aphids to be added to the list and dead ones deleted;
(3) a count to be kept of the number of aphids at each life stage, and so on.

A similar approach may be used for manipulating the plant data, both for storing the incubating plants and for recording those infected.

Finally there are a number of ways in which Fortran code may be ported to parallel architectures either automatically (for example, using the automatic parallelisation provided by the Lahey/Fujitsu Fortran 95 compiler (Lahey, 2000)) or by using additional libraries (OpenMP, 2000, for example,)).

6 Performance of the Fortran 95 Simulation Software

We considered a number of different implementations of the model in order to ascertain what effects the underlying data structures would have on run times.

Version 1 used a linked list to store the individual aphid details, this involved a record of the form

```
TYPE aphid
```

```
    PRIVATE
    REAL (sp) :: aphid_age
    INTEGER :: life_stage, coord, bydv_status, time_to_infection
    TYPE (aphid), POINTER :: next_aphid
  END TYPE aphid
```

Version 2 was the same as Version 1 but stored the data associated with each aphid in packed format as described in Hopkins and Morse (1997).

Version 3 used an array of type `aphid`, where each element of the array was a record of the form described in Version 1 but without the `POINTER` field. The required array space was allocated once and, therefore, fixed at some maximum size, at the start of the simulation.

Version 4 used a set of disjoint arrays to store the relevant details of the individual aphids. Although this sounds extremely crude it does allow a Fortran 77 implementation to be generated very easily. This provided an additional comparison of the efficiency of the two language compilers on the same hardware.

The Fortran 95 code for all four versions was successfully compiled and executed using the following systems

(1) Edinburgh Portable Compilers Fortran 90 version 1.5.2.6,
(2) Sun WorkShop 6 update 1 Fortran 95,
(3) NAGWare Fortran 95 compiler Release 4.0a(309)

without any source code changes.

The results reported in this section refer to simulations as described in section 3 with $p = 100$, 500, 1000, 1500 and 2000.

Table 6 shows timings for the four different implementations in Fortran 95 run using the Sun f95 compiler with the *-fast* compiler flag which attempts to optimise for speed of execution. This option is claimed to provide close to maximum attainable performance for many realistic applications.

By the effective use of modules the code changes required to use linked lists instead of arrays are restricted to a small number of short routines within a single module. The vast majority of the software (approximately 80% in terms of lines) remained unchanged. These timings therefore show that, for the Sun f95 compiler, access to data through user controlled pointers rather than array indexing increases the run time by around 25%. The main benefit to the user is that there is no requirement to guess in advance the size of the final aphid population in order to reserve enough array space. Using the current implementation of versions 3 and 4, underestimating this maximum value results in the simulation being aborted part way through; this problem could be overcome by reallocating more array space and copying the old data into the new space as required. Such a scheme was not implemented as it was

Table 1
Execution times in seconds for Fortran 95 implementations using the Sun f95 compiler

p	Unpacked Linked List	Packed Linked List	Array of Records	Separate Arrays
100	0.10	0.10	0.08	0.18
500	2.43	2.38	1.86	1.89
1000	10.20	9.76	7.70	7.69
1500	24.31	22.48	17.49	17.35
2000	46.38	42.63	33.30	33.64

felt that the costs of temporarily doubling the memory requirements and then copying the data were prohibitive.

No timing penalty resulted from using an array of aphid records as opposed to separate arrays of data; i.e., the cost of accessing five pieces of data via array indexing is the approximately the same as extracting the individual fields from a record. Although in the uncertainty levels of the timer the execution times were consistently between 2 and 5% faster using type arrays. The use of arrays of records makes the code easier to read and less prone to inconsistent array indexing errors; a cost free benefit of using the extra facilities available in Fortran 95.

Thirdly when comparing the two linked list versions we found that the packing of data not only lessened the storage requirements but, perhaps surprisingly, also decreased the run time slightly; again this reduction is slight but consistent.

Although different in absolute terms the other compilers used showed very similar execution time behaviours for the four implementations. There was one peculiarity that is worth mentioning. Unlike the EPC and Sun compilers where the ratio of execution times for the linked list version over the array version stayed almost constant, the NAG compiler showed an increase from 1.34 to 1.84 as p increased from 1000 to 2000; apparently the overhead of using linked lists grows with the length of the list.

Finally, using the Sun f77 compiler (Sun WorkShop 6 update 1 FORTRAN 77 5.2) with the -fast flag, the Fortran 77 version of version 4 ran almost 20% faster than the Fortran 95 separate arrays version. This version did not allocate arrays and held all arrays in labelled common blocks; include files, which were not standard for Fortran 77, were used to minimise the number of textual changes required when changing declared array lengths. In addition, it was necessary to store the plant information as an integer array whose length was the number of plants in the field and whose elements store only a single bit of information (infected or uninfected). This could have been compressed to a bit level but would have required either non-standard or inefficient means of setting and retrieving individual plant information. There is thus a trade-off between increased run-time efficiency and increased storage requirements.

7 An Object-Oriented Design

The desire to model the behaviour of each individual plant and aphid strongly suggests an object-oriented design for the simulation. Four major classes fall naturally from the model: *Plant*, *Field*, *Climate* and *Aphid*. A plant object maintains its BYDV status and a dynamically expanding collection of references to the aphids currently occupying it. A field object contains a fixed-size array of the plant objects and provides methods to report on the status of the plants and aphids at the end of each day. A singleton climate object (Gamma et al., 1995) provides the data for calculating daily ageing and reproductive rates. Most complex is the correct representation of aphids in an object-oriented design. The requirement for some distinct life-stage behaviour for infants, morphs and winged aphids suggests the use of a class hierarchy rather than a single class. An object-oriented language allows an aphid super class to implement those aspects of state and behaviour that are common to all life stages — such as age, BYDV status and a reference to the plant an aphid occupies. Further sub classes of this super class then only need to implement the distinguishing characteristics of each life stage, such as the ability to reproduce or fly. The fact that an individual aphid changes its behaviour as it moves through different life stages suggests the use of the state pattern (Gamma et al., 1995) to capture this effect. Using the state pattern, each aphid is represented to the simulation by an object of a separate context class whose interface makes it look as if it is an aphid. A context object contains a reference to an object of one of the genuine life-stage classes, to which it delegates all the interaction it receives from the simulation. When an aphid moves from one life stage to another, the object representing the previous life stage is replaced within the context object by a new object of the next life-stage class. All details of these changes are hidden from the simulation by the use of the state pattern.

Implementations of a design based on these classes were created in two of the most popular current object-oriented languages; Java and C++. Table 7 shows comparative timings for these two versions, plus a further C++ version whose details are described in section 7.2.

Table 2
Execution times in seconds for Java and C++ implementations

p	Java Full plants	C++ Full plants	C++ Partial plants
100	1.84	0.43	0.15
500	60.72	13.93	4.67
1000	211.92	62.13	22.71

7.1 Performance of the Object-Oriented Implementations

The Java implementation was developed using version 1.3.1 of the Java 2 Platform, Standard Edition (J2SE)(Sun Microsystems Inc., 2001). The C++ version was developed using version 2.95.2 of the GNU C++ compiler (GNU Project, 2001). The Java version was developed first and took approximately one day to code from a detailed design, reinforcing the view that the model lent itself well to an object-oriented design. Java's good exception handling features enabled the source of common runtime errors, such as array bound accesses and invalid reference (i.e. 'pointer') accesses, to be quickly identified and corrected during its development. Standard collection classes supported the need for a dynamic data structure to hold the growing numbers of aphids on a single plant, and avoided concerns with low-level manipulation of fixed-size arrays or hand-crafted dynamic data structures. The C++ version closely followed the Java version in style.

A comparison of the execution times for these two similar versions shows significant differences. Java is essentially an interpreted language whereas C++ is a compiled language. While technologies exist for Java to improve its performance — such as runtime compilation of performance-critical code — it would appear that interpretation still had a significant impact on the execution speed of the simulation.

In addition, quite large amounts of memory were required by the Java version. A field with $p = 500$ required a 64Mbyte memory allocation pool while a $p =$ 1000 field required a 512Mbyte memory pool. The large memory requirement (compared with the minimum values calculated in section 4) is a reflection, in part, of the large memory requirement of objects in the Java runtime system. These figures should not be too surprising, however, given the fact that the design and implementation did not attempt to optimise representation of the model at the expense of program clarity.

7.2 Partial Plant Creation

In an effort to seek further improvements to the CPU and memory requirements of the object-oriented versions, an optimisation was made to the structure of the C++ version. In the original version, each plant in the field is created as a separate object at the start of the simulation. Each day, the simulation iterates over the plants in the field in order to advance the incubation stage of plants that have been bitten, and ages the aphids according to the climate model. One object per plant in the field is not strictly necessary because plants are almost entirely passive objects. They only affect the outcome of the simulation when they are either infected by an aphid or are required to pass on an infection to their aphid population. In the modified version, an object to represent a plant was only created when needed; that is, when an aphid landed on it. Only the simulation loop and the *Field* class were affected, and no modifications were required to either the *Plant* or *Aphid* classes. When a new aphid arrived in the field, or an existing one moved to another plant, the host plant was created in the field if it did not already exist there. This on-demand version represented a significant improvement in run time.

Clearly, further modifications to the C++ version are possible, such as those described in section 4. However, there comes a point where such modifications effectively mean that the implementation no longer possesses those characteristics that distinguish it as object-oriented, and the merits of implementing it in a specifically object-oriented language no longer apply.

8 Summary and Conclusions

We have developed both object-oriented and structured programming language implementations of a relatively simple individual-based model for the spread of BYDV within a cereal field. The model was well suited to a naive implementation of an object-oriented design, but both the memory and CPU overhead of this approach are prohibitive for other than small scale simulations. An alternative Fortran 95 version produced a significantly more efficient implementation. The new pointer facilities available in Fortran 95 allowed us to experiment with data structures that grew with the aphid population and the bit level intrinsic functions provided a portable, memory efficient means of storing the state (infected/uninfected) of each plant. These same intrinsic functions also meant that we could compress the data describing each individual aphid as far as possible and this allowed much larger problems to be solved with a negligible overhead from packing and unpacking the data.

The use of linked lists rather than arrays increases the run times by around

20% but relieves the user from having to supply an upper bound for the final aphid population. Using records to store individual aphid data rather than separate arrays has no effect on run time while improving the readability of the code.

References

Ahearn, S. C., Smith, J. L. D., Joshi, A. R., Ding, J., 2001. TIGMOD: an indivual-based spatially explicit model for simulating tiger/human interaction in multiple use forests. Ecological Modelling 140, 81–97.

Arnold, K., Gosling, J., Holmes, D., 2000. The Java Programming Language, 3rd Edition. Addison Wesley, Reading, MA, Java is a trademark of Sun Microsystems Inc.

Booch, G., 1994. Object-Oriented Analysis and Design with Applications, 2nd Edition. Addison Wesley, Reading, MA.

Congleton, W. A., Pearce, B. R., Beal, B. F., 1997. A C++ implementation of an individual/landscape model. Ecological Modelling 103, 1–17.

Cowan, Jr., J. H., Houde, E. D., Rose, K. A., 1996. Size-dependent vulnerability of marine fish larvae to predation: an individual-based numerical experiment. ICES Journal of Marine Science 53, 23–37.

DeAngelis, D., Gross, L., 1992. Individual-based models and approaches in ecology. Chapman & Hall, London.

Gamma, E., Helm, R., Johnson, R., Vlissides, J., 1995. Design Patterns — Elements of Reusable Software. Addison Wesley, Reading, MA.

GNU Project, 2001. Gnu compiler collection.
URL `http://www.gnu.org/software/gcc/gcc.html`

Grimm, V., 1999. Ten years of individual-based modelling in ecology: what have we learned and what could we learn in the future? Ecological Modelling 115, 129–148.

Hopkins, T., Morse, D., 1997. The implementation and visualization of a large spatial individual-based model using Fortran 90. In: Denzer, R., Swayne, D., Schimak, G. (Eds.), Environmental Software Systems, Volume 2. Chapman & Hall, London, pp. 284–291.
URL `http://www.cs.ukc.ac.uk/pubs/1997/527`

Huston, M., Deangelis, D., Post, W., 1988. New computer-models unify ecological theory. Bioscience 38 (10), 682–691.

ISO/IEC, 1997. Information Technology – Programming Languages – Fortran - Part 1: Base Language (ISO/IEC 1539-1:1997). ISO/IEC Copyright Office, Geneva.

Judson, O., 1994. The rise of the individual-based model in ecology. Trends in Ecology & Evolution 9 (1), 9–14.

Kawata, M., Toquenaga, Y., 1994. From artificial individuals to global patterns. Trends in Ecology & Evolution 9 (11), 417–421.

Lahey, 2000. Lahey/Fujitsu Fortran 95 User's Guide Linux Edition. Lahey Computer Systems, Inc., Incline Village, NV, Revision B Edition.

McElhany, P., Real, L., Power, A., 1995. Vector preference and disease dynamics — a study of barley yellow dwarf virus. Ecology 76 (2), 444–457.

Morgan, D., 1989. A simulation model of BYDV epidemiology. Proceedings of CYMMIT Workshop on Barley Yellow Dwarf Virus — 1987 , 300–304.

Morgan, D., Carter, N., Jepson, P., 1988. Modelling principles in relation to the epidemiology of barley yellow dwarf virus. Bulletin IOBC/WPRS 11, 27–32.

OpenMP, Nov. 2000. OpenMP Fortran Application Program Interface. OpenMP Architecture Review Board.
URL `http://www.openmp.org/specs/mp-documents/fspec20.ps`

Power, A., 1996. Competition between viruses in a complex plant-pathogen system. Ecology 77 (4), 1004–1010.

Scheffer, M., Baveco, J., DeAngelis, D., Rose, K., van Nes, E. H., 1995. Superindividuals a simple solution for modelling large populations on an individual basis. Ecological Modelling 80, 161–170.

Shin, Y.-J., Cury, P., 2001. Exploring fish community dynamics through size-dependent trophic interactions using a spatialized individual-based model. Aquatic Living Resources 14 (2).

Stroustrup, B., 1997. The C++ Programming Language, 3rd Edition. Addison-Wesley, Reading, MA.

Sun Microsystems Inc., 2001. Java 2 platform.
URL `http://java.sun.com/j2se/`