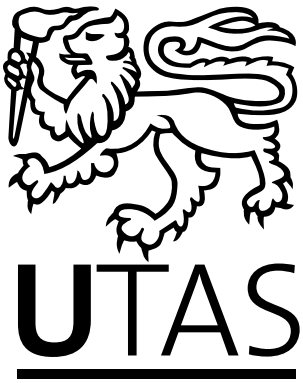
ADAPTIVE SIMULATION MODEL CONFIGURATION

by

Randall Gray, B.A. (Flinders University of South Australia)

Submitted in fulfilment of the requirements
for the Degree of Doctor of Philosophy

Department of Mathematics
University of Tasmania
October, 2016



I declare that this thesis contains no material which has been accepted for a degree or diploma by the University or any other institution, except by way of background information and duly acknowledged in the thesis, and, to the best of my knowledge and belief, no material previously published or written by another person, except where due acknowledgement is made in the text of the thesis; nor does the thesis contain any material that infringes copyright.

Signed: _____
Randall Gray

Date: _____

Signed: _____
Randall Gray

Date: _____

ABSTRACT

Models which are collections of largely independent, interacting submodels may spend significant amounts of real and simulated time running submodels in regions where other representations of the entities the submodels represent may perform better. This thesis demonstrates that even simple models may be improved by changing the representation of entities within such an aggregate model according to the the state of the system at a number of levels. *Introduction, Paper 1*

A commutative ring based based on a metric space with tree elements is developed as a tool for encoding the pertinent state information associated with a model configuration, the states of its components, and the real or expected efficiencies of the possible representations which may be used in the model as a whole.

An example model using a tree based technique is explored and used as a platform for discussion. *Paper 2*

The final chapter presents a fully formed model with detailed application of the techniques described and alluded to in preceding chapters. *model description*

Actual implementation of the critical parts of the model are included in the appendix.

ACKNOWLEDGEMENTS

My deepest thanks go to my very patient wife, Anne, who coped with far too many weeks with me firmly attached to my keyboard. Thanks also go my supervisor, Simon, who was never short of optimism and encouragement.

Chapter 1

Introduction

1.1 Overview and introduction

The primary aim of this thesis is to demonstrate an approach to constructing computational models which can be used to effect changes in the mathematical or algorithmic structure of model of complex systems in response to changes in their status. A simple uptake-depuration model of organisms which migrate through a contaminant field is presented in Chapter 2 to explore the fundamental principles and to demonstrate the utility of such a mechanism. The third chapter develops a mathematical structure with properties that make it a useful mechanism for representing the complex states of these structurally adaptible models. A number of useful functions are defined, and the basic properties of the structure are proved. Chapter 4 uses this mathematical structure for encoding the complex states of a model and its components in a "thought experiment" which illustrates the dynamics which might arise in a simple model.

Chapter A provides a detailed discussion of a real implementation of the thought model used in Chapter 4. The chapter briefly describes the modelling framework used to implement the model, and then addresses the details of the implementation. **Using ODD?** *T*he body of code is included in the Appendix. If we include it all it is 500pp

In this thesis, we will use *model* to refer to a (possibly trivial) collection of of components which, together, simulate a system. The term *submodel* will generally refer to a largely discrete component of a model (arguably a model in its own right). More specifically, we will consider submodels to be the abstract or *in potentia* representations, and *agents* to be the *in esse* instances of submodels.

Traditional models of complex systems incorporate alternative code paths or expressions to deal with fundamentally different dynamics which may be required within a component of the model by testing for conditons which arise during a simulation and executing an alternate code path within the component. Examples of this approach can be found in models implemented in a number of popular modelling frameworks [Wilensky, 1999, Minar et al., 1996, Collier and North, 2013, Luke et al., 2004] and in my previous work [Gray et al., 2006c,

Fulton et al., 2009b].

In contrast, the models used as examples in this thesis deal with such changes by changing the component used — a number of agents (instances of submodels)[1] associated with individual contact with a contaminant plume might be substituted for a population agent which has no code path of its own to deal with contaminant uptake or depuration.

Representing the state of the model, either as a whole or of its constituent parts, is not simple: not only may the components of the model vary through time, but their own dependences on other components may change as the state of the model changes: a model which contains a "whale spotting" tourism venture may follow the activities of whales at particular times of the year, but be utterly indifferent to them at times when there is little likelihood of their presence in an accessible location. Similarly, the association of entities represented by a population-based model needs to be maintained if the population disaggregates into super-individual-based or individual-based agents.

These factors make a simple vector-based encoding mechanism for the states of a model and its components awkward, thus we turn to a metric space over a rng[2] of trees with a finite number of weighted, labelled nodes. This simplifies the comparison of possible component mixes for a given global state, and makes available any algorithms (such as for clustering) which depend only on ring properties, excluding those that need a multiplicative unit.

Any attempt to assess the relative merit of a number of possible component mixes for a model relative to its current state must have some way of incorporating the necessary dependencies arising from the effects of the current state on the , and be able to calculate (in some sense) the incremental costs of change.

1.2 Historical work

`Probably need more historical work here`

Many individual-based models incorporate data which represents environmental characteristics and influences the behaviour of of the individuals simulated. Botkin et al. [1972c] and DeAngelis [1978] are important early example which model individuals in in the context of spatially explicit environmental gradients. In both cases, the subjects of the simulations are represented as individuals, and the environments are relatively static (though interactions, such as shading in JABOWA, also played a role). Even as formulated, we can consider the environments in these models as static models in their own right: from its early inception, the individual-based modelling approach implicitly couples different styles of model components.

The coastal marine ecosystem model in Gray et al. [2006a] used different representations for the agents which represented the organisms comprising the benthic habitat based on their life-stage, though the nature of this change was

[1]

[2]To avoid confusion, we take rng to denote a ring without a multiplicative indentity element.

configured during the compilation of the model. Here, juvenile biomasses could be represented by polygonal clouds which changed shape as they were advected, and adult stages could be represented by several different submodels which might be optimised either for speed or for spatial fidelity. This model was in most other ways similar to conventional agent-based modelling of the time.

Bobashev et al. [2007] describes a model of epidemic simulation in which the representation of populations or portions of populations are decided based on the number of infected individuals relative to a nominated trigger value. This model demonstrates that there is an advantage to changeing representation in terms of computational efficiency and the fidelity of the model. The model in Chapter 2 is similar: the rule governing the switching from one representation to another occurs when some monitored quantity (the number of infected individuals or the proximity to plume, for example) crosses a nominated boundary. The primary operational model underpinning the work in Fulton et al. [2009a] takes a step further, migrating whales change their representation according to their putative geographic location, specifically whether they were in the model domain or outside it. This model also had to consider the linkages between agents: agents representing predators, wildlife management and tourism operators had to be aware of the presence of "in domain" whales.

1.3 Structure and function

Chapter 2

Increasing model efficiency by dynamically changing model representations

Randall Gray[1]

CSIRO Division of Marine and Atmospheric Research

Simon Wotherspoon

University of Tasmania

Abstract

There are a number of strategies to deal with modelling large complex systems such as large marine ecosystems. These systems are often comprised of many submodels, each contributing to the overall trajectory of the system. The balance between the acceptable modelling error and the run-time often dictates the form of these submodels. There may be scope to improve the position of this balance point in both regards by structuring models so that submodels may change their algorithmic representation and state space in response to their local state and the state of the model as a whole.

This paper uses an example system consisting of a single population of animals which periodically encounters a diffuse contaminant in a localised region as an example of such a system, and discusses the key issues that arise from the approach.

[1]Published in *Environmental Modelling and Software*, 2012
Corresponding author: `Randall.Gray@limnal.net` (Randall Gray),
`Simon.Wotherspoon@utas.edu.au` (Simon Wotherspoon)

2.1 Introduction

There is a body of literature stretching back several decades which discusses individual-based modelling as a useful alternative to classical models. Early examples modelled forest canopy dynamics, notably JABOWA and its derivatives (Botkin et al. [1972b], Botkin et al. [1972c]). The number of significant papers and books has steadily increased since the 1980s. These works describe the use of individual-based models across a broad range of systems, and the relative strengths and weaknesses of the approach (such as Huston et al. [1988], DeAngelis and Gross [1992] and Grimm and Railsback [2005]). Classical models exploring populations and ecological systems are usually associated with modelling the dynamics of large groups and arguably appeared at the end of the eighteenth century with Malthus's ***An Essay on the Principle of Population*** (1798). The properties of these models are well understood and their state variables usually correspond to measurable quantities. Often, they are much faster than individual-based counterparts, and the analysis of model error may be much more straightforward. Classical and individual-based approaches represent the ends of a spectrum of aggregation in time, space and membership. Representations lying between these extrema, such as described by Scheffer et al. [1995], capitalise on the process-fidelity of an individual-based representation and gain some of the computational efficiency of a more aggregated classical approach, but an adaptive exploitation of the strengths of different representations is possible and worth exploring.

Ecosystem models are becoming broader in scope (Rose et al. [2010], DeAngelis et al. [1998], Harvey et al. [2003] Fulton et al. [2004], Gray et al. [2006a], webpage) and include more species with richer environments. The environmental response to climate change has also made anthropogenic pressure an important feature in many of these models. As this trend grows it seems less likely that a single model drawn from any particular region of this spectrum will be able to address all members and processes equally well. Simulation models often embed their subject in an "environment" comprised of primary data and other models and these components may occupy many places in the spectrum of representations. The model's actual implementation may be anything from a set of distinct models which are coupled together but retain their independence, to a corpus of code with the submodels so integrated that there is no real distinction between one "model" and the next.

The dynamics associated with biological and ecological systems can depend on the distributions and states of individuals in ways which are not amenable to equation-based modelling. The individual-based models described in Farolfi et al. [2010], and de Almeida et al. [2010] deal with systems of this sort. Versions of these models could be embedded in a common simulation environment in order to address more complex problems which span traditional domain boundaries, and such a model could address broader questions, such as how mosquito control strategies may best adapt to evolving agricultural practices and watershed conditions. Thiele and Grimm [2010] describes an extension to NetLogo which allows modellers to incorporate calls to R functions to aid in configuring the model to meet desirable mathematical conditions, to provide ongoing anal-

ysis, and to display the model's state through its run. This interface between R and NetLogo could be extended to support incorporating mathematical decision models written in R into the model's decision tree. The fusion of these three elements would form a system capable of simulating possible trajectories for the management of watersheds and human health in ways which would not be possible with a traditional monolithic modelling approach.

Models are including more functional groups and the interactions between components are becoming more detailed. It is costly in terms of computational load to address this increased demand for detail: individual-based models of populations may be very good at capturing vulnerability to exceptional events, but such simulations take a long time. Much of this time may be spent with the model in a largely unchallenging or uninteresting part of its state-space.

This paper explores the technique of changing the representation of a component of a model based on its location in its state-space. Modellers already do this to some degree: time-steps or spatial resolutions are changed, particular code paths may be by-passed to avoid pointless work, or additional calculations might be performed to reduce the error when the state is changing rapidly. These optimisations are largely optimisations of the *encoding* of the model or submodels, rather than an actual change in representation.

Vincenot et al. [2011a] make a clear case for considering what the authors term "hybrid-models." They present four reference cases which they use to describe ways in which equation-based models and individual-based models might be coupled to increase their utility. Their categories of hybrid-models are: individual-based models interacting with a single system dynamics model, system dynamics models embedded in individual-based models, individual-based models interacting with a number of system dynamics models, and models in which the representation swaps between individual-based and an equation-based form. They argue that a hybrid approach may provide a means of increasing the speed and accuracy of our models and Lyne et al. [1994b] and Gray et al. [2006a] have demonstrated that large models of ecosystems can be modelled this way. Vincenot et al. note that they found relatively few models which use both individual-based and equation-based submodels, and they present no existing models representing their fourth reference case. This final case, where models swap from equation-based to individual-based, is briefly described in general terms and is clearly intended to encompass models like the model of this paper.

This "mutating" or "switching" approach to the problem of managing complex simulations was developed using the experience from making several large scale human-ecosystem interaction models (Lyne et al.; Gray et al.; and a current, larger study of Ningaloo coastal region (*work in progress*)). In each of these studies a significant component of the model focused on simulating the interaction between organisms and contaminant plumes, though there is nothing that inherently limits the techniques to these sorts of studies. Lyne et al. assessed the potential of contaminants originating in industrial waste percolating through the food chain into commercially exploited fish stocks. Gray et al. developed a regional model to assess managment strategies for human activ-

ity which interacts with the biological systems along the Northwest Shelf of Australia.

Simulating contaminant interactions in an ecosystem is expensive in terms of run-time and memory use. The models described by Gray et al. and Lyne et al. include contaminant transport, uptake and depuration modelling, with behavioural sensitivity to contaminants. In Gray et al., the time taken to run a simulation with contaminants increased by roughly an order of magnitude, and in both studies a large amount of time was spent in regions where no interaction with contaminant plumes was possible. Monte [2009] presents a lucid discussion of analytic *contaminant migration-population effects* models. These models incorporate the movement of populations and their internal distribution, the transport of contaminants through the system via biotic and abiotic pathways, and the changes in behaviour and population dynamics associated with contamination. Monte discusses a method of coupling the equations which govern contaminant dispersion with the equations for population dynamics and migration. The technique depends on the equations of the location and the dispersion of members of a population satisfying an independence condition with respect to time and location which must hold. He states that the class of systems where the "movement of animals, the death and birthrates of individuals in $\mathbf{x}$ [location] at instant t [time] depend on previously occupied positions" is not generally amenable to the approach and suggests that repeated simulations of many individuals is an appropriate way of dealing with this situation.

It is unnecessary to run a complex model and carry the burden of maintaining its state when a simple model may perform better. If representations are switched appropriately, there is potential for improvements in run-time and accuracy. We need to consider four basic questions to do this:

1. What data need to persist across representations?
2. When should a model change representation?
3. How is the initial state for a new representation constructed?
4. How should the error associated with the loss of state information be managed?

The answer to these questions is specific to the set of submodels in question. Before expending resources and effort on a large scale model there needs to be a demonstration that the notion is worth pursuing, and some indication of how it might be accomplished. The aim of this paper is to provide this demonstration rather than to develop a comprehensive body of techniques supporting the approach. Many systems may benefit from similar techniques; obvious candidates are models of marginal populations, and the population dynamics of animals with behaviour where short periods of time have a significant influence on population levels (Wolff [1994] and Elderd et al. [2008], for example).

2.2 Overview: an *ODD* model description

The *ODD* protocol [Grimm et al., 2006] is used to describe the example model. We discuss the issues associated with making such a system, strategies and the reasons behind them in the Discussion section.

2.2.1 Purpose

This example model plays two roles. Its first is as an explicit demonstration, and the second is as a tool to explore the larger subject of changing a model's representation in response to its state. This example is overly simple, but it shares a number of features with plausible models and the analysis and development of the mutating model should be a reasonable template for other systems.

The model simulates organisms moving along a simple migratory path which intersects a region containing a field of fluctuating contamination (see Figure 2.2.1). This model exhibits fundamental attributes of larger studies of pollutant/ecosystem interactions (Lyne et al. and Gray et al.) and, while it is not intended to accurately represent any particular system, it might loosely correspond to some body of water influenced by contaminant loads associated with terrestrial runoff resulting from intense rainfalls.

Figure 2.1: Snapshots of individuals' locations at 28 day intervals superimposed on the migratory path. The plume's contact domain is marked by a grey ellipse near the position of individuals at day 28, with the track of a single individual approaching it. The domain of a population is is circumscribed around the individuals at day 196 for comparison.

The test models are composed of one or more submodels which run within a simple time-sharing system. Each submodel runs for a nominated period of time and passes control to the next submodel, very much like tasks running in many modern computer operating systems. In a *mutating* configuration, a trial will have different models take turns representing components of the system.

The population-based and individual-based submodels have been kept as similar as practicable in order to minimise the sources of divergence.

2.2.2 State variables and scales

There are essentially three distinct submodels in the simulation: an individual-based representation of the migrating group, population-based representation of the group, and a contaminant uptake-depuration model. We can think of the models which take the role of the group as candidates for filling a *niche*, which we can think of as the "sub-model shaped hole" in the middle of the program. Because the individual-based and population based models have fundamentally different spatial representations, each of these models include mechanisms to

evaluate their contact with a plume as they move through their environment. The spatial domain of the whole model system is a circular region with an arbitrary radius of somewhat more than 100km which encompasses both the area influenced by the contaminant source and the annual migratory path of the organisms. The plume can be viewed as a forcing function in the model and it has a maximum footprint area of approximately $43km^2$ which may be circular or elliptical and is centered on a point of the migratory circle. Both the elliptic and circular variants of the plume have the same area, and their intensities are adjusted so that the integral of the contaminant concentration over the region is the same.

The individual-based representation maintains a contaminant load associated with contact with the plume, a location, a direction and the next time at which it is scheduled to run. The population-based representation treats the group as homogeneous with respect to all state variables other than the contaminant load, and maintains only a record of its next time-to-run and an indication of contaminant load in the population. In the straight population-based representation, this is a single value, but in the mutating system the submodel maintains a list of contaminant loads which correspond to the non-zero loads of individuals. The plume model is deterministic with respect to time and location and maintains no state variables.

2.2.3 Process overview and scheduling

Simulations were run with 90 minute timesteps for a period representing twelve years. At each time-step, each instance of a submodel is rostered in a priority queue sorted on the "time-to-run" state variable, and when it comes to the top of the queue it executes.

Populations operate in a straightforward way: their path is deterministic, exposure to contaminants is calculated, and the resulting values are fed through the uptake-depuration equation. Individuals calculate their path (a segment of a directed random walk which follows the path of migration) and contact for the timestep and an apply the uptake-depuration equation. At the end of a timestep in non-mutating configurations, data is accumulated for output and each submodel reinserts itself in the priority queue. Otherwise, a heuristic is used to choose an appropriate representation for the niche in next timestep and that is inserted into the queue. Randomisation within a time-step is unnecessary, since the individual's or the population's contaminant updates are resolved for the contaminant contact across their time-step and are not dependent on the state of any other agents.

2.3 Design concepts

The central reason for the model is the mutability of the representation of the simulated organisms. Individuals and populations in the model are profoundly simple: no real scope is present for any of the trait categories mentioned in

Grimm et al. [2006], apart from their interaction with the contaminant plume, though this interaction is completely deterministic with respect to their path through the plume. In place of these traits, we have the basic heuristics associated with triggering a change from a population-based representation to individuals and a corresponding heuristic which indicates when an individual should join (or become) a population. The actual mechanism which turns a population into individuals or its converse is not necessarily a property of those models. Since the objective is to examine the impact of changing model representation in a fairly narrow situation, no attempt is made to optimise the submodels in the "non-contact" areas which constitute most of the model domain.

2.4 Details

2.4.1 Initialisation

Individuals and populations initially begin with no contaminant load, and individuals are positioned according to the two-dimensional normal distribution which characterises the population's assumed distribution. When a population mutates into an appropriate set of individuals, the individuals are positioned in the same fashion (centered on the centre of the population) with their corresponding contaminant loads either taken from the list of non-zero contaminant loads maintained by the population or initialised to be zero should the population's list fall short.

2.4.2 Input

Several characteristic features of the model are determined by the time and location represented. The contaminant intensity (and hence extent) at any point, $\mathbf{r}$, relative to the centroid of the plume, $\mathbf{m}_{plume}$, at a time, t, by the equation

$$I(t, \mathbf{r}) = \frac{1}{2}(1 + \cos(2\pi t/p)) \exp(-\psi\phi(\mathbf{r}, \mathbf{m}_{plume}))$$

where p is the period of 34 days, $\psi = 0.05$ is a decay exponent. We take ϕ to be a distance function, either $\phi(\mathbf{a}, \mathbf{b}) = |\mathbf{a} - \mathbf{b}|$, for a circular plume, or $\phi(\mathbf{a}, \mathbf{b}) = \sqrt{(\mathbf{a} - \mathbf{b}) \cdot (\sqrt{2}, \sqrt{1/2})}$, for an elliptical plume. The effective radius of the circular plume in the model is about 3.7% of the circular migratory path of the populations and individuals. The intensity of the elliptical plume is adjusted by scalar multiplication so that the integral of I for the two plumes over their domain is the same.

The individual-based and population-based models follow a circular migratory path about the origin. The path is traced annually and its location at any given time follows the equation $\mathbf{l}(t) = 10^5 (\cos(2t\pi/365.25), \sin(2t\pi/365.25))$.

2.4.3 Submodels

Individual-based representation

Individuals follow a directed random walk around the migratory circle described in the previous section. At each time-step the stride the individual takes is calculated according to its proximity to the "target" on the migratory path. There are a number of parameters associated with the movement of the individuals presented in Table 2.1.

Table 2.1: Parameters associated with individual movemement

Parameter	Value	Description
$\overline{V}$	4	A "variability" parameter associated with a Poisson-like process
q	0.5	A magnitude control parameter on directional change
μ_δ	1 day	Notional interval over which we calibrate individual's movement
μ	20km	Indicates the radius which is likely in a period of μ_δ
s	4ms^{-1}	nominal speed of the individuals

If we take δ to be the length of the current time step, and v to be a realisation of an event in a Poisson-like process with a mean of $\overline{V}$, we can take

$$Q = \left[1 - \exp\left(\frac{v}{\overline{V}} \log\left(1 - q\right)\right)\right]$$

to be a "variation" scalar which we use to evaluate an effective radial speed,

$$\nu_s = \left| -1 + \sqrt{1 + 4sQ^2 \frac{\delta}{\overline{V}}} \right| / 2Q^2.$$

Large values of Q correspond to long stretches of time without a change in direction, so we include Q in the calculation of α, the partial change in the individual's direction vector, by setting it to $\alpha = \pi rnd\left(-Q, Q\right)$. We can take their effective displacement over the 90 minute interval to be determined by a weighted sum of the normalised vector which joins them to their "target" location on the migratory path and a direction vector of length ν_s which is deflected by α.

Population-based representation

The population-based model assumes that a radially symmetric, normal distribution of individuals is an appropriate representation. Trials using the move-

ment model of the individual-based model were run, and the positions of individuals relative to their "target" on the migratory circle at each time-step closely matched a 2D-normal distribution with a $\sigma^2 = 3136.25^2$. Using this value, we define the density of the population at the point $\mathbf{p} = (p_x, p_y)$, relative to the population's centre, to be

$$\rho(\mathbf{p}) = S_{_L} \frac{1}{2\pi\sigma^2} \exp\left(-\frac{p_x^2 + p_y^2}{2\sigma^2}\right)$$

$S_L = 1.015$ is a scaling parameter chosen so that the integral over the population's effective disk, $\mathbf{D} = \{\mathbf{q} \in Domain\,(\rho) : |\mathbf{q}| \leqslant 3\sigma^2\}$, gives

$$\int_{\mathbf{D}} \rho\,(\mathbf{p})\, d\mathbf{p} = 1.$$

Contaminant handling

Initially a contact value is calculated for the time step. For an individual, this value is the integral of the contaminant level over its path. Population contact is calculated in an analogous way over the domain of the population and it represents the average contact of the members of the population.

The mass of contaminant which is available for uptake, or contact is, for individuals, taken to be the result of integrating the intensity of the plume over its path, $\mathbf{P}_t$ to $\mathbf{P}_{t+\delta}$. Namely,

$$M = \int_{\mathbf{P}_t}^{\mathbf{P}_{t+\delta}} I(\mathbf{p})||\mathbf{p}||d\mathbf{p}$$

where our variable $\mathbf{p}$ is a vector with time and location and we assume that the motion from $\mathbf{P}_t$ to $\mathbf{P}_{t+\delta}$ is along a straight line segment. We take $\|\mathbf{p}\|$ to be the speed at which the individual is moving.

Population's contact occurs across its domain and we calculate the definite integral

$$M = \int_{\mathbf{P}_t}^{\mathbf{P}_{t+\delta}} 2 \int_{\mathbf{\Omega}} I(\mathbf{p} + \omega)\rho(\omega)d\omega d\mathbf{p}$$

where $\mathbf{\Omega}$ is an area over which we assess the effective area of the population and $\mathbf{p} + \omega$ denotes the area $\mathbf{\Omega}$ translated so that its centroid corresponds to $\mathbf{p}$. The contact equations are solved using a simple adaptive quadrature routine. This value corresponds to the most likely mean contact in the population.

For both models of our organisms, uptake and depuration is modelled by the ordinary differential equation

$$dC/dt = uM - \lambda C$$

where $u = 0.02$ is the uptake rate, a decay rate which is approximately λ =0.0059. The equation is solved numerically with a fourth order Runge-Kutta algorithm for the value of C given a contact mass, M, and an initial contaminant value or vector of values for C

Mutating sub-models

The individual-based representation requires no change to run in a mutating configuration, but the population-based representation must maintain a list of contaminant loads which are processed in exactly the same way a scalar might be processed in one of the simple configurations. In the mutating configuration, each instance of a model is assessed at the end of its timestep to determine whether a change in representation is appropriate.

When a population disaggregates into individuals, the set of individuals with contaminant loads corresponding to the entries in the list are created. Their locations are normally distributed within the population disk. Any shortfall in numbers is handled by creating individuals which have no contaminant load and positioning them in the same fashion. Once the individuals are created, the population model is allowed to terminate.

An individual joins a population by having its contaminant load added to the list the population maintains. The first step in the process is to determine if there is a population close enough to the individual. If not, an empty population is created. Once a population's contaminant load has been inserted into the population the individual is allowed to terminate. The population model itself does not really play a part in this transaction: the "import" call is never used directly by the population, rather it is the supervising scheduler which organises the transfer to and from individuals and populations.

2.5 Results

The data presented in section 2.5.1 are based on two sets of simulations representing forty individuals. The first set uses a circular plume and the second an elliptical plume. These data sets allow a comparison of run-times, the equivalence (or lack of equivalence) amongst the submodels, and thet provide data to examine the robustness of the representations to changes in the configuration of the plume. To ensure that run-time comparisons are meaningful all of the simulations in the first set of trials were run on the same computer.

The first set is comprised of forty trials of the homogeneous individual-based model, corresponding trials of the mutating model, and a single run of the population-based model. The second set is comprised of eighty trials of the mutating model and a single run of the population-based model. The data in the first set of trials establishes the equivalence of the homogeneous individual-based model and the mutating model. We pool the data from the mutating and homogeneous individual-based runs from the first set to match the eighty runs in the second to compare the effect of the plume's configuration. The results with an elliptical plume were not consistent across the model representations.

The individual-based representation produces a time series of contaminant levels for each individual, while the population submodel produces a "mean load" across a group of entities. The mutating submodel sits between the two, sometimes producing individual time series and sometimes mean time series for

varying parts of the population. We denote representations by a subscript $r \in \{i, m, p\}$, so that $C_{rkj}(t)$ is the contaminant load at t in time series, C, associated with individual j in trial k of representation r, $C_{rk}(t)$ is the mean at a time t over all the groups simulated in the indicated representation and trial, and $C_r(t)$ denotes the mean of $C_{rk}(t)$ across the k trials for the indicated representation. To compare the dynamics of the system we generate mean time series for each of the k trials in the individual-based and mutating sets, $C_{ik}(t)$ and $C_{mk}(t)$. We are careful to generate the correct mean in the mutating submodel from time steps which have a mixture of individual trajectories and mean trajectories from population-based representations. Each of the mean time series, $C_{rk}(t)$, corresponds to the mean contaminant load of the population, $C_p(t)$, produced by the population submodel; averaging them, that is constructing

$$C_r(t) = \frac{1}{k} \sum_{j=1}^{k} C_{rj}(t),$$

where r is one of 'i' or 'm', is equivalent to running many stochastic trials and averaging to fit the population submodel. Using $C_{ik}(t)$, $C_{mk}(t)$ and $C_p(t)$ we find the maximum value attained for each representation, $\hat{C}_r$. We are also interested in the mean value across time of each representation,

$$\bar{C}_r = \frac{1}{T} \sum_{t \in T} C_r(t)$$

2.5.1 Contaminant load correspondence between representations

Both sets, $\bar{C}_r$ and $\hat{C}_r$, are presented in Table 2.2.

Table 2.2: Maxima and Means

$Series_r$	$\hat{C}_r$	$\bar{C}_r$
C_i	0.1787	0.0390
C_m	0.1821	0.0392
C_p	0.1387	0.0350

These data suggest that the mutating representation is consistent with the homogeneous individual-based representation. The population-based representation seems to present markedly different mean and maximum values.

2.5.2 Contaminant load variability

We calculated measures of variability in the time series using the aggregated time series $C_{ik}(t)$ and $C_{mk}(t)$ and their respective means across the k trials,

$C_i(t)$ and $C_m(t)$. We will take T to be the total number of time steps taken, and we take

$$\hat{\sigma}_{ab} = \max_{t\in[1,T]} \left[\frac{1}{k} \sum_{j=1}^{k} (C_{ak}(t) - C_b(t))^2 \right]^{1/2}$$

and

$$\bar{\sigma}_{ab} = \left[\frac{1}{T} \sum_{t=1}^{T} \left[\frac{1}{k} \sum_{j=1}^{k} (C_{ak}(t) - C_b(t))^2 \right] \right]^{1/2},$$

to be the maximum root mean square error and the average root mean square error. Clearly we can write $\bar{\sigma}_{rr}$ as $\bar{\sigma}_r$ without introducing ambiguity, and similarly for $\hat{\sigma}_r$. The values for these measure of variability are presented in Table 2.3.

Table 2.3: Deviations amongst the model runs with respect to a given mean

r.m.s.e.	$r = i$	$r = m$	$r = p$
$\widehat{\sigma}_{ir}$	0.0083	0.0084	0.0534
$\bar{\sigma}_{ir}$	0.0024	0.0024	0.0096
$\widehat{\sigma}_{mr}$	0.0090	0.0090	0.0538
$\bar{\sigma}_{mr}$	0.0024	0.0024	0.0096

The data here indicate that the variability about the mean is consistent in the two representations which use simulated individuals to estimate contact and uptake. This is what we would expect since the mechanisms of uptake and contact are the same. In contrast, the population's values suggest that the contact and uptake are quite different, and that this model does not perform in quite the same way.

2.5.3 Sensitivity to the shape of the plume

We will use the same notation as Section 2.5.2 for the data derived from the circular plumes, while we will add a prime symbol to the data derived from the elliptical plumes. Thus, the mean value time series for the mutating submodel with elliptical plumes would be denoted C'_m and the mean value of that time series is $\bar{C}'$.

There is a good correspondence between the means and deviations associated with the mutating model in the circular and elliptical plume scenarios, but there is much poorer correspondence in the population based results in the two scenarios. The data for the circular plume and for the elliptical plume are presented in Tables 2.4 and 2.5 respectively.

The population based model is clearly more sensitive to the shape of the plume than the mutating model. It seems likely that the major driver of this difference

Table 2.4: Circular plume results

$Series_r$	$\hat{C}_r$	$\bar{C}_r$	StdDev	$r = m$	$r = p$
C_m	0.1738	0.0392	$\widehat{\sigma}_{mr}$	0.0088	0.0535
C_p	0.1387	0.0350	$\bar{\sigma}_{mr}$	0.0024	0.0098

Table 2.5: Elliptical plume results

$Series_r$	$\widehat{C'}_r$	$\overline{C'}_r$	StdDev	$r = m$	$r = p$
C'_m	0.1856	0.0394	$\widehat{\sigma}'_{mr}$	0.0087	0.0616
C'_p	0.1763	0.0445	$\bar{\sigma}'_{mr}$	0.0025	0.0092

is that the long axis of the plume (a region where the net contact will be higher) remains in close proximity to the centroid of population where the population density is greatest.

2.5.4 Run-time

Each run collected data regarding the amount of time spent in different parts of the submodel; predictably, most of the effort is in calculating contact and updating contaminant loads.

The optimisation of supressing the contact calculations when a population is outside the area of potential contact seemed to make very little difference to the run-time of population submodel (about 3%). It seems unlikely to make a great deal of difference to the mutating submodel. In the case of the purely individual-based submodel, this sort of optimisation is likely to play a much bigger role; any penalty would be multiplied by the number of animals simulated.

The population submodel ran for 98.7 cpu seconds. This submodel is deterministic and the amount of cpu time used is very stable, so only a single run is considered for comparison. The purely individual-based submodels took just over a mean time of 4205 cpu seconds with a standard deviation of approximately 16 seconds and the mean of the mutating submodel's run time was 1157 cpu seconds with a standard deviation of slightly over 11 cpu seconds.

2.6 Discussion

In the example our objective is to produce time-series data associated with the contaminant load of the group. Our individual-based model is taken as the best model for capturing the contact that real organisms have with an intermittent plume, and the population based representation has a computational efficiency that the individuals lack. The case for swapping in the example model is reasonably clear: there is a distinct improvement in run-time with no apparent deterioration in the fidelity of the dynamics. It seems likely that the naïve population distribution may be introducing a systematic divergence from what we see as an accurate, but computationally intense, individual-based model.

The general case is not limited to the polar extremes of switching between individual-based models and populations. A niche may have many representations, each of which has a particular set of strengths and weaknesses. This adaptive approach would present the same scope for improvement in purely equation-based models, where rules of thumb might be replaced by first-order approximations, or by complex systems of differential equations. In a purely individual-based example, the depth of the representation of the individual might vary from a simple mass and location through to a level of detail which included the individual's metabolic rates and breeding characteristics.

2.6.1 State spaces

To make our population-based model compatible with the individual-based model we have to extend the population's state space and maintain additional information to preserve the essential parts of the individual representation that makes it valuable to us. This is basically posing the first of the enumerated question from section 2.1. The *significant* information which the individual-based representation possesses is embodied in the contaminant loads amongst the individuals which comprise the group. We assume that the role of their relative locations about the population's centre is not important over a large portion of the global state-space and that we can discard it when we move from individuals to populations.

The union of submodels' state-spaces can generally be decomposed into *processing sets* of state-variables. Partitioning the state variables in this way – particularly in advance – makes it easier to analyse and minimise the boundary effects associated with the transition from one representation to another. Within a representation, the state variables which are unique to it form a special subset. The subset can be divided into the variables which need to be maintained by other representations, which we call V_r, and the variables which do not which we will call U_r. In principle, the variables in V_r might be maintained by a routine which is common to them all. The variables in U_r are more complex: when some other representation is mutating to representation r, the values assigned to the variables in U_r should reflect the state implied by the state of the old representation.

In the example model, an individual's relative location is a member of this set.

Variables which are maintained and used by more than one representation are the third major group. This group can be divided into the set which is used consistently across the submodels (W_r), and the group of variables which have different dynamics in the various representations (X_r). In our example case, the contaminant load level of an individual would belong to the set V_{ind}. Its location and velocity would be in U_{ind}, the current time for both individuals and populations belongs to W_r, and a list of contaminant loads for populations belongs to V_{pop}.

2.6.2 Heuristics

The example model has very simple dynamics: the plumes are always in the same place, the migration is very predictable, and the spatial domain an individual may explore is well contained. Implementing a heuristic for the model which efficiently decides when to move from one representation to another is very straightforward: *If we are close enough that an individual might encounter the plume if the plume were at its maximum, switch a population to a group of individuals. Conversely, if there is no chance that an individual heading straight toward the plume (backwards) will encounter it, move the individual to a "close enough" population, or create a new population to accomodate the individual.* The model was constructed so that any number of populations could be run, and the heuristic was framed so that there were no assumptions about the number of population agents and the size of the groups they represented.

In this model, the decision process was shared between the representations themselves and the controlling scheduler. The representations reported their "robustness" to the scheduler which would then decide how to act on the advice, either triggering a change in representation or not.

The goal is to have a complex ensemble of niches which will change representations under the aegis of the scheduler to optimise the global outcome. For this more general approach additional information is needed: the scheduler would incorporate information about what properties each of the current representations required from other niches in order to decide what the mix of submodels filling the niches ought to be.

2.6.3 Transitions

The transition from individuals to population and population to individuals involves the loss and reconstruction of fine-scale position data, which may be a source of error. In our example, we assume that we can reconstruct a plausible position for each individual from the population's distribution function because we know the typical distribution of individuals and there is no behavioural change associated with contaminant load. A contaminant that made an organism sluggish would skew the distributions of both the population as a whole and the distribution of intoxicated organisms within the population. In the example model, individuals were randomly located in this way with enough time to randomise their velocities and blur any artifacts resulting from the se-

lection of their locations. This corresponds to the perception that the velocities and relative locations of the individuals are comparatively unimportant except as they related to the distribution of the population. When values of state variables are generated in the process of changing to another representation, they need to conform to the distributions of the representation they are leaving. If for some strange reason (like behavioural change) the distribution of individuals, for example, does *not* conform to the distribution associated with a coherent population, then additional steps need to be taken accomodate this when changing to a population-based representation.

In section 2.6.2, transitions between representations in the example model are mediated by the scheduler. Decisions to change representation must be based, in part, on whether the transition will increase or decrease the efficacy of the suite of representations as a whole. If the example model were more than a pedagogic tool, it would have been useful to assess the mean error introduced in the transition from population to individual and individual to population. Our simulation was aimed at producing contaminant load results, but in this context our aim would be to track the mean position, variance and extrema of the distribution of individuals relative to the population. To do this we would perform a comparison similar to that of section 2.5 but using positional data rather than contaminant load. The results would indicate if there might be significant transition effects associated with the change in representation. This sort of testing should ideally be performed at a number of scales (temporal or spatial, for example) since the knowledge of how long it takes for the transition boundary effects to settle (if they do) should feed into the high-level managing scheduler. In a sophisticated system, a representation might be spun up in advance so that the boundary effects have settled before it takes over from a less efficient representation.

2.6.4 Errors

Estimating error and confidence is extremely hard in complex models. Not only are the abstract processes deeply connected, but there may be hundreds of thousands of lines of code[2] which may introduce error of their own. Confronted with the code of a large-scale marine ecosystem model, one might turn around and contemplate joining a monastery rather than try and track the error propagation through the system. When we look at making an aggregate model out of niches, we can make the set of each of the representations which fill the niche simpler than some chimera which tries to take the best bits of each of the candidate representations. With well isolated transition mechanisms, we side-step nests of conditional code-paths and can contain the potential sources of error we must analyse. Since transitions can be recorded (like other useful data), we can generate an indication of the likely level of confidence based on our understanding of the representations used in a simulation.

[2]Ningaloo-InVitro is currently more than 233000 lines of C++, NWS-InVitro is slightly more than 118000 lines of C++ and Atlantis is more than 145000.

2.7 Conclusion

Interesting and unanticipated results have come from this experiment. The discrepancy between the data concerning elliptical and circular plumes suggests that, at least in contaminant work, we need to pay closer attention to the movement dynamics of individuals and the density functions of populations. More predictably, the run-times show that mutating configurations provide a reasonable means of increasing computational speed without sacrificing fidelity in appropriate situations. The simple example demonstrated that a model which changes the representation of the system according to its location in its state space could provide much better computational efficiency than a model with a constant representation with no loss of accuracy.

Increasing population and resource use often reduce our environment's resilience and there is a growing need to model larger, more complex parts of the system we live in. Techniques for this have been iteratively moving toward a more systematic approach: many models will optimise their run-time and accuracy by suppressing unnecessary calculation, models will split their timesteps according to what they are simulating, or perhaps even adaptively set an appropriate timestep or spatial scale.

This paper proposes that actually changing the representations to suit the different regions of the state space of the model could provide a better balance between computational efficiency and error.

In our experience of large scale marine ecosystem modelling, the size of the system considered is growing much faster than computational capacity. Even for small systems the possibility of adjusting the representation of submodels to optimise the accuracy of the model as a whole has great appeal. Mutating models may provide an effective means of concentrating the use of computational capacity where it is most needed.

The authors would like to thank the three reviewers whose advice and suggestions have made this paper much clearer and to the point. Their care and insight are deeply appreciated.

Chapter 3

A commutative rng over a metric space with tree elements

3.1 Introduction

The subject of this chapter arose in course of devising a mechanism to use responses to survey questions to incorporate the "attitudes" of populations of simulated entities in a way which allows events and the attitudes of other entities to influence the attitudes of each entity. The specific goal was to be able to simulate the way public opinion changes in response to policy actions, and changes in the economy and environment. This structure also has application in controlling simulation models which permit the representation of submodels to change in response to their own local state and the local states of other submodels.

More generally we will be considering trees comprised of labelled nodes with associated values. Each such node may have an arbitrary number of children.

3.2 Conventions and preliminary definitions

Generally, we will use lower case, boldface symbols to denote a node (or tree); upper case, boldface symbols to denote sets (particularly sets of nodes); and other symbols (such as x) will typically refer to numbers or rational polynomials. Elements of a node, $\boldsymbol{u}$ will be identified using an appropriate subscript, namely $\boldsymbol{u}_v$, for the node's value, $\boldsymbol{u}_{\mathsf{P}}$ for its label and $\boldsymbol{u}_{\boldsymbol{E}}$ for its set of extensions. We will take $\mathsf{P}_{\boldsymbol{A}}$ to be the field of rational multivariate polynomials with variables taken from a set of symbols $\boldsymbol{A}$.

Definition 3.2.1. *A node, $\boldsymbol{u}$, is a representative of a recursive structure of the form $(v, p, \boldsymbol{E})$ where $v \in \mathbb{F}, p \in \mathsf{P}_{\boldsymbol{A}}$, and its extension set, $\boldsymbol{E} \subset \boldsymbol{T}$, contains no two elements with the same label. Without necessarily limiting ourselves, we will take $\mathbb{F}$ to be $\mathbb{R}$. Nodes or trees with $\boldsymbol{E} = \emptyset$, the empty set, will be called* simple nodes, simple trees, *or* leaf nodes, *and simple nodes which also have*

scalar polynomials as their labels may be referred to as scalar nodes *or* scalar trees*. The domain of trees,* $\boldsymbol{T}$*, is the collection of (acyclic) trees with a finite number of nodes of this form. We will make use of a special element in* $\boldsymbol{T}$*,* $\mathbf{O} = (0, 0, \emptyset)$*, an analogue of zero which we will call the* zerotree*.*

The set of valid labels is taken to be a ring of polynomials, but elements of any *(commutative) ring could serve equally well. For other applications the choice of polynomials may not be appropriate; here, polynomials provide a simple example which is easily manipulated and printed.*

Any two nodes are said to be *compatible* if they have the same label, or at least one of the nodes is the $\mathbf{O}$. Two trees are compatible if their root nodes are compatible. This property is largely analogous with compatibility in matrices. When there is no risk of ambiguity, we will use the same symbol to refer to a set, $\boldsymbol{T}$ for example, and a vector space based on that set.

Definition 3.2.2. *For* $\boldsymbol{u} \in \boldsymbol{T}$ *we define the function*

$$\text{depth}(\boldsymbol{u}) = \begin{cases} 0 & \text{if } \boldsymbol{u} = (0, 0, \emptyset) = \mathbf{O} \\ 1 & \text{if } \boldsymbol{u} \text{ is a simple node} \\ 1 + \max(\{\text{depth}(\boldsymbol{v}) : \forall\, \boldsymbol{v} \in \boldsymbol{u}_{\boldsymbol{E}}\}) & \text{otherwise} \end{cases}$$

which gives us the depth of the tree.

Definition 3.2.3. *We will also define for* $\boldsymbol{u} \in \boldsymbol{T}$*,*

$$\text{trim}(\boldsymbol{u}) = \begin{cases} \mathbf{O} & \text{if } \boldsymbol{u} = \mathbf{O} \\ \mathbf{O} & \text{if } \boldsymbol{u} \text{ is simple} \\ (\boldsymbol{u}_v, \boldsymbol{u}_{\mathsf{P}}, \{\text{trim}(\boldsymbol{e}) : \forall\, \boldsymbol{e} \in \boldsymbol{u}_{\boldsymbol{E}}\} \setminus \{\mathbf{O}\}) & \text{otherwise.} \end{cases}$$

Trimming essentially removes all simple nodes from the tree. A recursive application of trimming will be denoted trim_k, indicating that the tree $\boldsymbol{u}$ will be trimmed k times. Note that $\text{trim}_{\text{depth}(\boldsymbol{u})}\ \boldsymbol{u} = 0$ and $\text{depth}(\text{trim}_{\text{depth}\ \boldsymbol{u}-1}\ \boldsymbol{u}) = 1$.

Definition 3.2.4. *The cardinality of a tree is the number of nodes it contains. We define it formally as*

$$\|\boldsymbol{u}\|_{\mathsf{T}} = \begin{cases} 0 & \text{if } \boldsymbol{u} = \mathbf{O} \\ 1 + \sum_{\boldsymbol{e} \in \boldsymbol{u}_{\boldsymbol{E}}} \|\boldsymbol{e}\|_{\mathsf{T}}. \end{cases}$$

Simple nodes are the only nodes which have a cardinality of one, and $\mathbf{O}$ *is the only node or tree with a cardinality of zero.*

Definition 3.2.5. *The* overlap *between two trees is defined*

$$\text{overlap}(\boldsymbol{u}, \boldsymbol{v}) = \begin{cases} 0 & \text{if } \boldsymbol{u} = \mathbf{O} \text{ or } \boldsymbol{v} = \mathbf{O} \text{ or } \boldsymbol{u}_{\mathsf{P}} \neq \boldsymbol{v}_{\mathsf{P}} \\ 1 + \sum\limits_{\substack{\boldsymbol{e} \in \boldsymbol{u}_{\boldsymbol{E}} \\ \boldsymbol{f} \in \boldsymbol{v}_{\boldsymbol{E}}}} \text{overlap}(\boldsymbol{e}, \boldsymbol{f}) & \text{otherwise} \end{cases}$$

Clearly two trees, $\boldsymbol{u}$ *and* $\boldsymbol{v}$*, are compatible if and only if* $\text{overlap}(\boldsymbol{u}, \boldsymbol{v}) \neq 0$ *; they will be said to* completely overlap *if* $\|\boldsymbol{u}\|_{\mathsf{T}} = \|\boldsymbol{v}\|_{\mathsf{T}} = \text{overlap}(\boldsymbol{u}, \boldsymbol{v})$*.*

3.3 Addition and Scalar Multiplication

We will define scalar multiplication of the trees in $\boldsymbol{T}$, then we will define a few useful mappings which will help keep the expressions simple. Our aim, in this section, is to define addition, and to show that the defined scalar multiplication and addition make this a vector space.

Definition 3.3.1. *Given $a \in \mathbb{F}$ and $\boldsymbol{u} \in \boldsymbol{T}$, we define*

$$a\,\boldsymbol{u} = \begin{cases} \mathbf{O} & \text{if } a = 0 \text{ or } \boldsymbol{u} = \mathbf{O} \\ (a\,\boldsymbol{u}_{v},\, \boldsymbol{u}_{\mathsf{P}}, a\,\boldsymbol{u}_{\boldsymbol{E}}) & \text{otherwise} \end{cases}$$

and we observe that the notation $-1\,\boldsymbol{u} \equiv -\,\boldsymbol{u}$ is consistent with common use.

Definition 3.3.2. *We will define a few useful notation or relations on sets of nodes in $\boldsymbol{T}$. Take $\boldsymbol{U}$ and $\boldsymbol{V}$ be such sets and $\boldsymbol{a}$ be a node in $\boldsymbol{T}$; then*

$$\boldsymbol{L}(\boldsymbol{U}) = \{\, \boldsymbol{e}_{\mathsf{P}} : \forall\, \boldsymbol{e} \in \boldsymbol{U}\}$$

$$\boldsymbol{U}|_{\boldsymbol{L}(\boldsymbol{V})} = \{\, \boldsymbol{f} \in \boldsymbol{U} :\, \boldsymbol{f}_{\mathsf{P}} \in \boldsymbol{L}(\boldsymbol{V})\}$$
$$\boldsymbol{U}|_{\overline{\boldsymbol{L}}(\boldsymbol{V})} = \{\, \boldsymbol{f} \in \boldsymbol{U} :\, \boldsymbol{f}_{\mathsf{P}} \notin \boldsymbol{L}(\boldsymbol{V})\}$$

and

$$a\boldsymbol{U} = \{a\,\boldsymbol{u} : \forall\, \boldsymbol{u} \in \boldsymbol{U}\}$$

Definition 3.3.3. *For convenience, we define analogues of several of the above relations for nodes to implicitly refer to the extension sets of those nodes.*

Let $\boldsymbol{u}$ and $\boldsymbol{v}$ be arbitrary nodes in $\boldsymbol{T}$. Then we define the following

$$\boldsymbol{L}(\boldsymbol{u}) \equiv \boldsymbol{L}(\, \boldsymbol{u}_{\boldsymbol{E}})$$
$$\boldsymbol{u}|_{\boldsymbol{L}(\,\boldsymbol{v})} \equiv\ \boldsymbol{u}_{\boldsymbol{E}}|_{\boldsymbol{L}(\,\boldsymbol{v}_{\boldsymbol{E}})}$$
$$\boldsymbol{u}|_{\overline{\boldsymbol{L}}(\,\boldsymbol{v})} \equiv\ \boldsymbol{u}_{\boldsymbol{E}}|_{\overline{\boldsymbol{L}}(\,\boldsymbol{v}_{\boldsymbol{E}})}$$

Definition 3.3.4. *For compatible nodes $\boldsymbol{u}$ and $\boldsymbol{v} \in \boldsymbol{T}$,*

$$\boldsymbol{u} + \boldsymbol{v} = \begin{cases} \boldsymbol{u} & \text{if } \boldsymbol{v} = \mathbf{O} \\ \boldsymbol{v} & \text{if } \boldsymbol{u} = \mathbf{O} \\ \Big(\boldsymbol{u}_{v} + \boldsymbol{v}_{v},\, \boldsymbol{u}_{\mathsf{P}}, (\,\boldsymbol{u}|_{\overline{\boldsymbol{L}}(\,\boldsymbol{v})} \cup \boldsymbol{v}|_{\overline{\boldsymbol{L}}(\,\boldsymbol{u})} \\ \quad \cup\{\, \boldsymbol{r} + \boldsymbol{s} :\, \boldsymbol{r} \in \boldsymbol{u}|_{\boldsymbol{L}(\,\boldsymbol{v})} \text{ and } \boldsymbol{s} \in \boldsymbol{v}|_{\boldsymbol{L}(\,\boldsymbol{u})} \text{ and } \boldsymbol{r}_{\mathsf{P}} = \boldsymbol{s}_{\mathsf{P}}\}) \setminus \{\, \mathbf{O}\}\Big) & \text{otherwise} \end{cases}$$

Definition 3.3.5 (Notation)**.** *We define a mapping which takes two sets, each comprised of nodes in $\boldsymbol{T}$, to another set of nodes.The resulting set contains the nodes of the first and second operands and the set obtained by applying the operator to compatible pairs from the first and second set. This notation is used to express the restriction of addition to compatible nodes in the extensions of the root nodes. Let*

$$\boldsymbol{B} \boxplus \boldsymbol{C} = (\,\boldsymbol{B}|_{\overline{\boldsymbol{L}}(\,\boldsymbol{C})} \cup \boldsymbol{C}|_{\overline{\boldsymbol{L}}(\,\boldsymbol{B})} \cup \{\, \boldsymbol{r} + \boldsymbol{s} :\, \boldsymbol{r} \in \boldsymbol{B}|_{\boldsymbol{L}(\,\boldsymbol{C})} \text{ and } \boldsymbol{s} \in \boldsymbol{C}|_{\boldsymbol{L}(\,\boldsymbol{B})} \text{ and } \boldsymbol{r}_{\mathsf{P}} = \boldsymbol{s}_{\mathsf{P}}\}) \setminus \{\, \mathbf{O}\}.$$

3.4 Vector space

Proposition 3.4.1. *$\boldsymbol{T}$, with scalar multiplication and addition is a vector space.*

Proof. We will assume that $\boldsymbol{p}$, $\boldsymbol{q}$, $\boldsymbol{u}$, $\boldsymbol{v}$, $\boldsymbol{w} \in \boldsymbol{T}$ and $a, b \in \mathbb{F}$, and that addition $\boldsymbol{u}$, $\boldsymbol{v}$ and $\boldsymbol{w}$ are compatible.

Additive identity element

This is explicit in the definition of addition between elements of $\boldsymbol{T}$.

Inverse elements with respect to addition

The element $\mathbf{O}$ is its own inverse, since $\mathbf{O} + \mathbf{O} = \mathbf{O}$ by definition.

So, we consider the case of $\boldsymbol{u}$, where $\text{depth}(\boldsymbol{u}) = 1$, then

$$\begin{aligned} \boldsymbol{u} + -\boldsymbol{u} &= (\boldsymbol{u}_v, \boldsymbol{u}_\mathsf{P}, \emptyset) + (-1\,\boldsymbol{u}_v, \boldsymbol{u}_\mathsf{P}, \emptyset) \\ &= (\boldsymbol{u}_v + -\boldsymbol{u}_v, \boldsymbol{u}_\mathsf{P}, \emptyset) \\ &= \mathbf{O} \end{aligned}$$

so for any simple node, $\boldsymbol{u}$, $-\boldsymbol{u}$ is its inverse.

Let $\boldsymbol{v}$ be a non-null node which is not simple, but has simple extension nodes, that is to say $\text{depth}(\boldsymbol{v}) = 2$). Then

$$\begin{aligned} \boldsymbol{v} + -\boldsymbol{v} &= (\boldsymbol{v}_v - \boldsymbol{v}_v, \boldsymbol{v}_\mathsf{P}, \{\boldsymbol{e} + -\boldsymbol{e} : \forall\, \boldsymbol{e} \in \boldsymbol{v}_{\boldsymbol{E}}\} \setminus \{\mathbf{O}\}) \\ &= (0, \boldsymbol{v}_\mathsf{P}, \emptyset) \\ &= \mathbf{O} \end{aligned}$$

since the simple leaf nodes are all added to their own additive inverse.

Having established this, we can generalise to trees with a depth greater than two. Assuming that the proposition holds for elements of $\boldsymbol{T}$ with depth n, we consider an element, $\boldsymbol{u}$, where $\text{depth}(\boldsymbol{u}) = n + 1$ added to the element $-\boldsymbol{u}$.

$$\boldsymbol{u} + -\boldsymbol{u} = (0, \boldsymbol{u}_\mathsf{P}, \{\boldsymbol{e} + -\boldsymbol{e} : \forall\, \boldsymbol{e} \in \boldsymbol{u}_{\boldsymbol{E}}\})$$

Since the extension nodes of the root node of $\boldsymbol{u}$ are, by assumption, added to their additive inverses, they then become

$$\begin{aligned} &= (\boldsymbol{v}_v + -\boldsymbol{v}_v, \boldsymbol{v}_\mathsf{P}, \emptyset) \\ &= \mathbf{O} \end{aligned}$$

for each $\boldsymbol{v} \in \boldsymbol{u}_{\boldsymbol{E}}$ and, by induction, our inverse holds for all members of $\boldsymbol{T}$.

Multiplicative identity element

First observe that $1\,\mathbf{O}$ is by definition $\mathbf{O}$.

Now consider an arbitrary non-null tree in $\boldsymbol{T}$, $\boldsymbol{u}$; $\boldsymbol{u}$ is either simple, or it has extension nodes. In the case of a simple $\boldsymbol{u}$, it is obvious that $1 \in \mathbb{F}$ acts as an identity.

$$\begin{aligned}\boldsymbol{u} &= (\,\boldsymbol{u}_{v},\, \boldsymbol{u}_{\mathsf{P}}, \emptyset) \\ &= (1\,\boldsymbol{u}_{v},\, \boldsymbol{u}_{\mathsf{P}}, \emptyset) \\ &= 1(\,\boldsymbol{u}_{v},\, \boldsymbol{u}_{\mathsf{P}}, \emptyset) \\ &= 1\,\boldsymbol{u}\end{aligned}$$

Thus, for all leaf nodes on $\boldsymbol{u}$, 1 is the multiplicative identity. Now we take $\boldsymbol{u}$ to be some non-simple node,

$$\begin{aligned}\boldsymbol{u} &= (\,\boldsymbol{u}_{v},\, \boldsymbol{u}_{\mathsf{P}},\, \boldsymbol{u}_{\boldsymbol{E}}) \\ &= (1\,\boldsymbol{u}_{v},\, \boldsymbol{u}_{\mathsf{P}}, 1\,\boldsymbol{u}_{\boldsymbol{E}}) \\ &= (1\,\boldsymbol{u}_{v},\, \boldsymbol{u}_{\mathsf{P}}, \{(1\,\boldsymbol{e}_{v},\, \boldsymbol{e}_{\mathsf{P}},\, \boldsymbol{e}_{\boldsymbol{E}}) : \forall\, \boldsymbol{e} \in\, \boldsymbol{u}_{\boldsymbol{E}}\}),\end{aligned}$$

and, if the extensions are all simple nodes,

$$\begin{aligned}&= 1(\,\boldsymbol{u}_{v},\, \boldsymbol{u}_{\mathsf{P}},\, \boldsymbol{u}_{\boldsymbol{E}}) \\ &= 1\,\boldsymbol{u}.\end{aligned}$$

so it is the case that 1 is the multiplicative inverse for nodes which have a depths of less than three.

Now suppose that 1 is the multiplicative inverse of all members of $\boldsymbol{T}$ with a depth of n or less for some natural number n, and we consider $\boldsymbol{v}$ which has a depth of $n+1$. Then

$$\begin{aligned}\boldsymbol{v} &= (\,\boldsymbol{v}_{v},\, \boldsymbol{v}_{\mathsf{P}},\, \boldsymbol{v}_{\boldsymbol{E}}) \\ &= (1\,\boldsymbol{v}_{v},\, \boldsymbol{v}_{\mathsf{P}}, 1\,\boldsymbol{v}_{\boldsymbol{E}})\end{aligned}$$

.

But each of the elements in the set $1\,\boldsymbol{v}_{\boldsymbol{E}}$ either has a depth of n or less and so the the extension set of $1\,\boldsymbol{v}$ is merely $\boldsymbol{v}_{\boldsymbol{E}}$, and so $1\,\boldsymbol{v} = \boldsymbol{v}$.

By induction, we can demonstrate that 1 is the multiplicative identity for nodes of arbitrary (finite) depth.

Commutativity

Let us consider compatible nodes $\boldsymbol{u}$ and $\boldsymbol{v}$.

The commutativity of addition involving $\mathbf{O}$ is guaranteed by the definition of addition. so we first address the case where both addends are simple.

Take $\boldsymbol{u}$ and $\boldsymbol{v}$ to be simple nodes; then

$$\begin{aligned}\boldsymbol{u}+\boldsymbol{v} &= (\boldsymbol{u}_v, \boldsymbol{u}_\mathsf{P}, \emptyset) + (\boldsymbol{v}_v, \boldsymbol{v}_\mathsf{P}, \emptyset)\\ &= (\boldsymbol{u}_v + \boldsymbol{v}_v, \boldsymbol{u}_\mathsf{P}, \emptyset)\\ &= (\boldsymbol{v}_v + \boldsymbol{u}_v, \boldsymbol{u}_\mathsf{P}, \emptyset)\\ &= \boldsymbol{v}+\boldsymbol{u}.\end{aligned}$$

Now suppose that there is some number n for which $\text{depth}(\boldsymbol{u}) \leq n$ and $\text{depth}(\boldsymbol{v}) \leq n \implies \boldsymbol{u}+\boldsymbol{v} = \boldsymbol{v}+\boldsymbol{u}$.

Then if we take $\boldsymbol{u}$ and $\boldsymbol{v}$ to be nodes with depths of $n+1$ or less,

$$\begin{aligned}\boldsymbol{u}+\boldsymbol{v} &= (\boldsymbol{u}_v, \boldsymbol{u}_\mathsf{P}, \boldsymbol{u}_{\boldsymbol{E}}) + (\boldsymbol{v}_v, \boldsymbol{v}_\mathsf{P}, \boldsymbol{v}_{\boldsymbol{E}})\\ &= \Big(\boldsymbol{u}_v + \boldsymbol{v}_v, \boldsymbol{u}_\mathsf{P}, (\{\boldsymbol{r}+\boldsymbol{s} : \boldsymbol{r} \in \boldsymbol{u}|_{\boldsymbol{L}(\boldsymbol{v})} \text{ and } \boldsymbol{s} \in \boldsymbol{v}|_{\boldsymbol{L}(\boldsymbol{u})} \text{ and } \boldsymbol{r}_\mathsf{P} = \boldsymbol{s}_\mathsf{P}\}\\ &\qquad \cup \{\boldsymbol{r} : \boldsymbol{r} \in \boldsymbol{u}_{\boldsymbol{E}} \text{ and } \boldsymbol{r}_\mathsf{P} \notin \boldsymbol{L}(\boldsymbol{v})\} \cup \{\boldsymbol{s} : \boldsymbol{s}_{\boldsymbol{E}} \in \boldsymbol{v} \text{ and } \boldsymbol{s}_\mathsf{P} \notin \boldsymbol{L}(\boldsymbol{u})\}) \setminus \{\mathbf{O}\}\Big)\\ &= \Big(\boldsymbol{u}_v + \boldsymbol{v}_v, \boldsymbol{u}_\mathsf{P}, (\{\boldsymbol{r}+\boldsymbol{s} : \boldsymbol{r} \in \boldsymbol{u}_{\boldsymbol{E}} \text{ and } \boldsymbol{r}_\mathsf{P} \in \boldsymbol{L}(\boldsymbol{v}) \text{ and } \boldsymbol{s} \in \boldsymbol{v}_{\boldsymbol{E}} \text{ and } \boldsymbol{s}_\mathsf{P} \in \boldsymbol{L}(\boldsymbol{u})\}\\ &\qquad \cup\ \boldsymbol{u}|_{\overline{\boldsymbol{L}}(\boldsymbol{v})} \cup \boldsymbol{v}|_{\overline{\boldsymbol{L}}(\boldsymbol{u})}) \setminus \{\mathbf{O}\}\end{aligned}$$

So,

$$\begin{aligned}\boldsymbol{u}+\boldsymbol{v} = \Big(\boldsymbol{v}_v + \boldsymbol{u}_v, \boldsymbol{u}_\mathsf{P}, (\{\boldsymbol{r}+\boldsymbol{s} : \boldsymbol{r} \in \boldsymbol{u}_{\boldsymbol{E}} \text{ and } \boldsymbol{r}_\mathsf{P} \in \boldsymbol{L}(\boldsymbol{v}) \text{ and } \boldsymbol{s} \in \boldsymbol{v}_{\boldsymbol{E}} \text{ and } \boldsymbol{s}_\mathsf{P} \in \boldsymbol{L}(\boldsymbol{u})\}\\ \cup\ \boldsymbol{u}|_{\overline{\boldsymbol{L}}(\boldsymbol{v})} \cup \boldsymbol{v}|_{\overline{\boldsymbol{L}}(\boldsymbol{u})}) \setminus \{\mathbf{O}\}\end{aligned}$$

since addition in $\boldsymbol{T}$ is commutative. If we can demonstrate that the expression for the extension set is independent of order, then it must be the case that sum of the addends, $\boldsymbol{u}$ and $\boldsymbol{v}$, must also be order independent.

The set $\{\boldsymbol{r}+\boldsymbol{s} : \boldsymbol{r} \in \boldsymbol{u}_{\boldsymbol{E}} \text{ and } \boldsymbol{r}_\mathsf{P} \in \boldsymbol{L}(\boldsymbol{v}) \text{ and } \boldsymbol{s} \in \boldsymbol{v}_{\boldsymbol{E}} \text{ and } \boldsymbol{s}_\mathsf{P} \in \boldsymbol{L}(\boldsymbol{u})\}$ must be order independent since each of the candidate $\boldsymbol{r}$ and $\boldsymbol{s}$ addends must have a depth of n or less. Since set union is commutative, the order of $\boldsymbol{u}|_{\overline{\boldsymbol{L}}(\boldsymbol{v})}$ and $\boldsymbol{v}|_{\overline{\boldsymbol{L}}(\boldsymbol{u})}$ doesn't affect the result, thus, addition must be commutative for all $\boldsymbol{u}$ where $\text{depth}(\boldsymbol{u}) \leq n+1$. By induction, this must be true for all $n \geq 0$.

Associativity

Let us consider compatible nodes $\boldsymbol{u}$, $\boldsymbol{v}$ and $\boldsymbol{w}$ in $\boldsymbol{T}$.

First consider the situation where the depths of $\boldsymbol{u}$, $\boldsymbol{v}$ and $\boldsymbol{w}$ are all less than or equal to one. If they all have a depth of zero, the sum is almost trivially the null tree. Similarly, if only one is the null tree, it rapidly degenerates to simple addition. So we take $\boldsymbol{u}$, $\boldsymbol{v}$, and $\boldsymbol{w}$ to be simple.

Then

$$\begin{aligned}(\boldsymbol{u}+\boldsymbol{v})+\boldsymbol{w} &= ((\boldsymbol{u}_v,\boldsymbol{u}_{\mathsf{P}},\emptyset)+(\boldsymbol{v}_v,\boldsymbol{u}_{\mathsf{P}},\emptyset))+(\boldsymbol{w}_v,\boldsymbol{u}_{\mathsf{P}},\emptyset)\\ &= (\boldsymbol{u}_v+\boldsymbol{v}_v,\boldsymbol{u}_{\mathsf{P}},\emptyset)+(\boldsymbol{w}_v,\boldsymbol{u}_{\mathsf{P}},\emptyset)\\ &= ((\boldsymbol{u}_v+\boldsymbol{v}_v)+\boldsymbol{w}_v,\boldsymbol{u}_{\mathsf{P}},\emptyset)\\ &= (\boldsymbol{u}_v+(\boldsymbol{v}_v)+\boldsymbol{w}_v),\boldsymbol{u}_{\mathsf{P}},\emptyset)\\ &= \boldsymbol{u}+(\boldsymbol{v}+\boldsymbol{w}).\end{aligned}$$

Let us consider the case where these may be non-simple trees. Suppose there is an integer n such that associativity holds for any three trees $\boldsymbol{u}$, $\boldsymbol{v}$ and $\boldsymbol{w}$, whose depth is less than or equal to n, that is if $\text{depth}(\boldsymbol{u}) \leq n, \text{depth}(\boldsymbol{v}) \leq n$ and $\text{depth}(\boldsymbol{w}) \leq n$, then it must be the case that

$$(\boldsymbol{u}+\boldsymbol{v})+\boldsymbol{w} = \boldsymbol{u}+(\boldsymbol{v}+\boldsymbol{w})$$

.

Now suppose one or more of these trees has a depth of $n+1$.

$$\begin{aligned}(\boldsymbol{u}+\boldsymbol{v})+\boldsymbol{w} &= ((\boldsymbol{u}_v,\boldsymbol{u}_{\mathsf{P}},\boldsymbol{u}_{\boldsymbol{E}})+(\boldsymbol{v}_v,\boldsymbol{u}_{\mathsf{P}},\boldsymbol{v}_{\boldsymbol{E}}))+(\boldsymbol{w}_v,\boldsymbol{u}_{\mathsf{P}},\boldsymbol{w}_{\boldsymbol{E}})\\ &= (\boldsymbol{u}_v+\boldsymbol{v}_v,\boldsymbol{u}_{\mathsf{P}},\boldsymbol{u}_{\boldsymbol{E}}\boxplus\boldsymbol{v}_{\boldsymbol{E}})+(\boldsymbol{w}_v,\boldsymbol{u}_{\mathsf{P}},\boldsymbol{w}_{\boldsymbol{E}})\end{aligned}$$

Recall that

$$\boldsymbol{u}_{\boldsymbol{E}}\boxplus\boldsymbol{v}_{\boldsymbol{E}} = (\boldsymbol{u}|_{\overline{\boldsymbol{L}}(\boldsymbol{v})}\cup\boldsymbol{v}|_{\overline{\boldsymbol{L}}(\boldsymbol{u})}\cup\{\boldsymbol{p}+\boldsymbol{q}:\boldsymbol{p}\in\boldsymbol{u}|_{\boldsymbol{L}(\boldsymbol{v})}\text{ and }\boldsymbol{q}\in\boldsymbol{v}|_{\boldsymbol{L}(\boldsymbol{u})}\text{ and }\boldsymbol{p}_{\mathsf{P}}=\boldsymbol{q}_{\mathsf{P}}\})\setminus\{\mathbf{O}\}$$

so, letting

$$\boldsymbol{B} = \boldsymbol{u}_{\boldsymbol{E}}\boxplus\boldsymbol{v}_{\boldsymbol{E}}$$

we get

$$\begin{aligned}(\boldsymbol{u}+\boldsymbol{v})+\boldsymbol{w} &= \Big((\boldsymbol{u}_v+\boldsymbol{v}_v)+\boldsymbol{w}_v,\boldsymbol{u}_{\mathsf{P}},(\boldsymbol{B}|_{\overline{\boldsymbol{L}}(\boldsymbol{w})}\cup\boldsymbol{w}|_{\overline{\boldsymbol{L}}(\boldsymbol{B})}\cup\boldsymbol{B}\boxplus\boldsymbol{w}_{\boldsymbol{E}})\setminus\{\mathbf{O}\}\Big))\\ &= \Big(\boldsymbol{u}_v+(\boldsymbol{v}_v+\boldsymbol{w}_v),\boldsymbol{u}_{\mathsf{P}},(\boldsymbol{B}|_{\overline{\boldsymbol{L}}(\boldsymbol{w})}\cup\boldsymbol{w}|_{\overline{\boldsymbol{L}}(\boldsymbol{B})}\cup\boldsymbol{B}\boxplus\boldsymbol{w}_{\boldsymbol{E}})\setminus\{\mathbf{O}\}\Big))\end{aligned}$$

since addition in $\boldsymbol{T}$ is associative

Notice that the elements of all the sets which comprise the extension set, those in $\boldsymbol{B}$ and in $\boldsymbol{w}_{\boldsymbol{E}}$, must have a depth of n or less; any addition which occurs amongst the elements of these sets must be associative by our inductive assumption. Hence

$$(\boldsymbol{u}+\boldsymbol{v})+\boldsymbol{w} = \boldsymbol{u}+(\boldsymbol{v}+\boldsymbol{w}).$$

Compatibility of scalar multiplication and multiplication in $\mathbb{F}$

Observe first that $a\,\mathbf{O} = \mathbf{O}, \forall a \in \mathbb{F}$. We also dispose with the case of simple nodes:

$$\begin{aligned} a(b\,\boldsymbol{u}) &= (a(b\,\boldsymbol{u}_v),\ \boldsymbol{u}_\mathsf{P}, \emptyset) \\ &= (ab\,\boldsymbol{u}_v,\ \boldsymbol{u}_\mathsf{P}, \emptyset) \\ &= ((ab)\,\boldsymbol{u}_v,\ \boldsymbol{u}_\mathsf{P}, \emptyset) \\ &= (ab)\,\boldsymbol{u}; \end{aligned}$$

So assuming that multiplication is compatible with nodes with depths of n or less, we consider $\boldsymbol{u}$, where $\text{depth}(\boldsymbol{u}) = n + 1$,

$$a(b\,\boldsymbol{u}) = a(b\,\boldsymbol{u}_v,\ \boldsymbol{u}_\mathsf{P}, b\,\boldsymbol{u}_{\boldsymbol{E}})$$

since $\text{depth}(\boldsymbol{e}) \leq n \forall\, \boldsymbol{e} \in \boldsymbol{u}_{\boldsymbol{E}}$, multiplication of these elements is compatible, and

$$= (ab\,\boldsymbol{u}_v,\ \boldsymbol{u}_\mathsf{P}, a(b\,\boldsymbol{u}_{\boldsymbol{E}}))$$

becomes

$$\begin{aligned} &= ((ab)\,\boldsymbol{u}_v,\ \boldsymbol{u}_\mathsf{P}, (ab)\,\boldsymbol{u}_{\boldsymbol{E}}) \\ &= (ab)\,\boldsymbol{u} \end{aligned}$$

Thus the scalar and field multiplication operators are compatible.

Distribution of scalar multiplication with respect to vector addition

Let us consider compatible trees, $\boldsymbol{u}$ and $\boldsymbol{v}$.

First, note that

$$\forall\, \boldsymbol{u} \in \boldsymbol{T}, a(\,\mathbf{O} + \boldsymbol{u}) = a\,\boldsymbol{u} = a\,\mathbf{O} + a\,\boldsymbol{u},$$

and that

$$\forall\, \boldsymbol{u}, \boldsymbol{v} \in \boldsymbol{T}, 0(\,\boldsymbol{u} + \boldsymbol{v}) = \mathbf{O} = 0\,\boldsymbol{u} + 0\,\boldsymbol{v}.$$

The property holds for simple nodes,

$$\begin{aligned} a(\,\boldsymbol{u} + \boldsymbol{v}) &= a((\,\boldsymbol{u}_v,\ \boldsymbol{u}_\mathsf{P}, \emptyset) + (\,\boldsymbol{v}_v,\ \boldsymbol{v}_\mathsf{P}, \emptyset)) \\ &= a(\,\boldsymbol{u}_v + \boldsymbol{v}_v,\ \boldsymbol{u}_\mathsf{P}, \emptyset) \\ &= (a(\,\boldsymbol{u}_v + \boldsymbol{v}_v),\ \boldsymbol{u}_\mathsf{P}, \emptyset) \\ &= (a\,\boldsymbol{u}_v + a\,\boldsymbol{v}_v,\ \boldsymbol{u}_\mathsf{P}, \emptyset) \\ &= (a\,\boldsymbol{u}_v,\ \boldsymbol{u}_\mathsf{P}, \emptyset) + (a\,\boldsymbol{v}_v,\ \boldsymbol{u}_\mathsf{P}, \emptyset) \\ &= a\,\boldsymbol{u} + a\,\boldsymbol{v} \end{aligned}$$

.

So, suppose that the equation $a(\boldsymbol{p}+\boldsymbol{q}) = a\,\boldsymbol{p}+a\,\boldsymbol{q}$ holds for all compatible nodes $\boldsymbol{p}$ and $\boldsymbol{q}$ such that $depth(\boldsymbol{p}) \leq k$, and $\text{depth}(\boldsymbol{q}) \leq j$.

Take $n = \min(j,k)$, $a \in \mathbb{F}$, and nodes $\boldsymbol{u}$ and $\boldsymbol{v}$ such that $\text{depth}(\boldsymbol{u}) = n+1$, and $\text{depth}(\boldsymbol{v}) = n+1$. Note that n must be greater than zero since the property holds for simple nodes. Then

$$\begin{aligned}
a(\boldsymbol{u}+\boldsymbol{v}) &= a((\boldsymbol{u}_v, \boldsymbol{u}_\mathsf{P}, \boldsymbol{u}_{\boldsymbol{E}}) + (\boldsymbol{v}_v, \boldsymbol{v}_\mathsf{P}, \boldsymbol{v}_{\boldsymbol{E}}) \\
&= a(\boldsymbol{u}_v + \boldsymbol{v}_v, \boldsymbol{u}_\mathsf{P}, \{\boldsymbol{u}|_{\overline{\boldsymbol{L}}(\boldsymbol{v})} \cup \boldsymbol{v}|_{\overline{\boldsymbol{L}}(\boldsymbol{u})} \\
&\qquad \cup \{\boldsymbol{r}+\boldsymbol{s} : \boldsymbol{r} \in \boldsymbol{u}|_{\boldsymbol{L}(\boldsymbol{v})} \text{ and } \boldsymbol{s} \in \boldsymbol{v}|_{\boldsymbol{L}(\boldsymbol{u})}\}\} \setminus \{\mathbf{O}\}) \\
&= \Big(a(\boldsymbol{u}_v + \boldsymbol{v}_v), \boldsymbol{u}_\mathsf{P}, \{\{a\,\boldsymbol{e} : \boldsymbol{e} \in \boldsymbol{u}|_{\overline{\boldsymbol{L}}(\boldsymbol{v})}\} \cup \{a\,\boldsymbol{e} : \boldsymbol{e} \in \boldsymbol{v}|_{\overline{\boldsymbol{L}}(\boldsymbol{u})}\} \\
&\qquad \cup \{a(\boldsymbol{r}+\boldsymbol{s}) : \boldsymbol{r} \in \boldsymbol{u}|_{\boldsymbol{L}(\boldsymbol{v})} \text{ and } \boldsymbol{s} \in \boldsymbol{v}|_{\boldsymbol{L}(\boldsymbol{u})}\}\} \setminus \{\mathbf{O}\})\Big) \\
&= \Big(a\,\boldsymbol{u}_v + a\,\boldsymbol{v}_v, \boldsymbol{u}_\mathsf{P}, \{\{a\,\boldsymbol{e} : \boldsymbol{e} \in \boldsymbol{u}|_{\overline{\boldsymbol{L}}(\boldsymbol{v})}\} \cup \{a\,\boldsymbol{e} : \boldsymbol{e} \in \boldsymbol{v}|_{\overline{\boldsymbol{L}}(\boldsymbol{u})}\} \\
&\qquad \cup \{a(\boldsymbol{r}+\boldsymbol{s}) : \boldsymbol{r} \in \boldsymbol{u}|_{\boldsymbol{L}(\boldsymbol{v})} \text{ and } \boldsymbol{s} \in \boldsymbol{v}|_{\boldsymbol{L}(\boldsymbol{u})}\}\} \setminus \{\mathbf{O}\})\Big)
\end{aligned}$$

.

Notice that the component sets of the extension to $\boldsymbol{u}+\boldsymbol{v}$, namely $\{a\,\boldsymbol{e} : \boldsymbol{e} \in \boldsymbol{u}|_{\overline{\boldsymbol{L}}(\boldsymbol{v})}\}$, $\{a\,\boldsymbol{e} : \boldsymbol{e} \in \boldsymbol{v}|_{\overline{\boldsymbol{L}}(\boldsymbol{u})}\}$ and $\{a(\boldsymbol{r}+\boldsymbol{s}) : \boldsymbol{r} \in \boldsymbol{u}|_{\boldsymbol{L}(\boldsymbol{v})} \text{ and } \boldsymbol{s} \in \boldsymbol{v}|_{\boldsymbol{L}(\boldsymbol{u})}\}$ can only contain nodes with a depth of n or less; Thus, we can proceed inductively, increasing the least upper bound, $(\min(j,k)$, for the set of trees that cooperate with distribution of scalar multiplication over vector addition, to any value we wish.

Distribution of scalar multiplication with respect to addition in $\mathbb{F}$

The property is clearly true when $\boldsymbol{u} = \mathbf{O}$, since $(a+b)\,\mathbf{O} = \mathbf{O} = a\,\mathbf{O}+b\,\mathbf{O}$.

We first consider simple nodes:

$$\begin{aligned}
(a+b)\,\boldsymbol{u} &= (a+b)((a+b)\,\boldsymbol{u}_v, \boldsymbol{u}_\mathsf{P}, \emptyset) \\
&= ((a+b)\,\boldsymbol{u}_v, \boldsymbol{u}_\mathsf{P}, \emptyset) \\
&= (a\,\boldsymbol{u}_v, \boldsymbol{u}_\mathsf{P}, \emptyset) + (b\,\boldsymbol{u}_v, \boldsymbol{u}_\mathsf{P}, \emptyset) \\
&= a\,\boldsymbol{u} + b\,\boldsymbol{u}.
\end{aligned}$$

Nodes with a depth of two are slightly more complicated,

$$\begin{aligned}
(a+b)\,\boldsymbol{u} &= (a+b)((a+b)\,\boldsymbol{u}_v, \boldsymbol{u}_\mathsf{P}, (a+b)\,\boldsymbol{u}_{\boldsymbol{E}}) \\
&= (a\,\boldsymbol{u}_v + b\,\boldsymbol{u}_v, \boldsymbol{u}_\mathsf{P}, \{(a+b)\,\boldsymbol{e} : \boldsymbol{e} \in \boldsymbol{u}_{\boldsymbol{E}}\})
\end{aligned}$$

but $\boldsymbol{u}_{\boldsymbol{E}}$ is composed of simple nodes, so,

$$\begin{aligned}
&= (a\,\boldsymbol{u}_v + b\,\boldsymbol{u}_v, \boldsymbol{u}_\mathsf{P}, \{a\,\boldsymbol{e} + b\,\boldsymbol{e} : \boldsymbol{e} \in \boldsymbol{u}_{\boldsymbol{E}}\}) \\
&= (a\,\boldsymbol{u}_v + b\,\boldsymbol{u}_v, \boldsymbol{u}_\mathsf{P}, a\,\boldsymbol{u}_{\boldsymbol{E}}) + (b\,\boldsymbol{u}_v, \boldsymbol{u}_\mathsf{P}, b\,\boldsymbol{u}_{\boldsymbol{E}}) \\
&= a\,\boldsymbol{u} + b\,\boldsymbol{u}.
\end{aligned}$$

Now suppose the property holds for nodes with a depth of n. Then we consider node $\boldsymbol{u}$ with a depth of $n+1$:

$$\begin{aligned}(a+b)\,\boldsymbol{u} &= (a+b)(\,\boldsymbol{u}_v,\,\boldsymbol{u}_{\mathsf{P}},\,\boldsymbol{u}_{\boldsymbol{E}})\\ &= ((a+b)\,\boldsymbol{u}_v,\,\boldsymbol{u}_{\mathsf{P}},(a+b)\,\boldsymbol{u}_{\boldsymbol{E}}),\\ &= (a\,\boldsymbol{u}_v + b\,\boldsymbol{u}_v,\,\boldsymbol{u}_{\mathsf{P}},\{a\,\boldsymbol{e}+b\,\boldsymbol{e}:\boldsymbol{e}\in\boldsymbol{u}_{\boldsymbol{E}}\})\end{aligned}$$

since depth$(\boldsymbol{e}) = n$

$$= a\,\boldsymbol{u} + b\,\boldsymbol{u}.$$

By induction, the property must hold for all $n >= 0$ □

3.5 Seminorms, norms and metrics

We will now construct a seminorm on the vector space $\boldsymbol{T}$. This will induce a norm on a quotient space of $\boldsymbol{T}$ which we can use as a tool for assessing the similarity of trees and, ultimately, provide both a means of clustering trees and selecting trees with particular properties.

3.5.1 $\boldsymbol{T}$ and its seminorm

Definition 3.5.1. *We define the absolute value of a node to be*

$$\langle\,\boldsymbol{u}\rangle = \begin{cases}0 & \text{if } \boldsymbol{u} = \mathbf{O}\\ |\,\boldsymbol{u}_v| & \text{if } \boldsymbol{u}_{\boldsymbol{E}} = \emptyset\\ |\,\boldsymbol{u}_v| + \sum_{\boldsymbol{e}\in\boldsymbol{u}_{\boldsymbol{E}}}\langle\,\boldsymbol{e}\rangle & \text{otherwise.}\end{cases}$$

The absolute magnitude is only based only on the values of the nodes of trees. Note that each node in a tree can only contribute a non-negative quantity to the absolute value of the tree, it is obvious that $\langle\,\boldsymbol{u}\rangle \geq 0$ for all $\boldsymbol{u}\in\boldsymbol{T}$ and that equality only occurs if the value of each node in the tree $\boldsymbol{u}$ is zero.

Proposition 3.5.1. *For $a\in\mathbb{F}$ and $\boldsymbol{u}\in\boldsymbol{T}$, $|a|\langle\,\boldsymbol{u}\rangle = \langle a\,\boldsymbol{v}\rangle$.*

Proof. The magnitude of the empty tree is trivially zero, so $|a|\langle\,\mathbf{O}\rangle = \langle a\,\mathbf{O}\rangle = 0$.

Consider simple nodes in $\boldsymbol{T}$:

$$\begin{aligned}|a|\langle\,\boldsymbol{u}\rangle &= |a|(|\,\boldsymbol{u}_v| + 0)\\ &= |a||\,\boldsymbol{u}_v|\\ &= \langle a\,\boldsymbol{u}\rangle.\end{aligned}$$

Now suppose that there is $n\geq 1$ such that the proposition is true for all trees with a depth of n or less. Then, taking $\boldsymbol{u}\in\boldsymbol{T}$ where depth$(\boldsymbol{u}) = n+1$, we

have

$$\begin{aligned}|a|\langle\, \boldsymbol{u}\rangle &= |a|((|\, \boldsymbol{u}_{v}| + \sum_{\boldsymbol{e}\in\, \boldsymbol{u}_{\boldsymbol{E}}} \langle\, \boldsymbol{e}\rangle)) \\ &= |a||\, \boldsymbol{u}_{v}| + \sum_{\boldsymbol{e}\in\, \boldsymbol{u}_{\boldsymbol{E}}} |a|\langle\, \boldsymbol{e}\rangle\end{aligned}$$

but all the elements in $\boldsymbol{u}_{\boldsymbol{E}}$ have a depth of k or less

$$\begin{aligned}&= ||a|\, \boldsymbol{u}_{v}| + \sum_{\boldsymbol{e}\in\, \boldsymbol{u}_{\boldsymbol{E}}} \langle|a|\, \boldsymbol{e}\rangle \\ &= |a\, \boldsymbol{u}_{v}| + \sum_{\boldsymbol{e}\in\, \boldsymbol{u}_{\boldsymbol{E}}} \langle a\, \boldsymbol{e}\rangle \\ &= \langle a\, \boldsymbol{u}\rangle.\end{aligned}$$

By induction, the proposition must be true for all $n \geq 0$. □

Proposition 3.5.2. *For* $\boldsymbol{u}$ *and* $\boldsymbol{v} \in \boldsymbol{T}, \langle\, \boldsymbol{u} + \boldsymbol{v}\rangle \leq \langle\, \boldsymbol{u}\rangle + \langle\, \boldsymbol{v}\rangle$.

Proof. We start by considering trees of depths zero and one. The case for null trees is trivial: $\langle\, \mathbf{O} + \mathbf{O}\rangle = |0 + 0| = 0$, and if only one of the trees has a depth of one, we get either $\langle\, \boldsymbol{u} + \mathbf{O}\rangle = \langle\, \boldsymbol{u}\rangle$ or $\langle\, \mathbf{O} + \boldsymbol{u}\rangle = \langle\, \boldsymbol{u}\rangle$.

For $\boldsymbol{u}$ and $\boldsymbol{v}$ with depths of one,

$$\langle\, \boldsymbol{u} + \boldsymbol{v}\rangle = \langle(\, \boldsymbol{u}_{v} + \boldsymbol{v}_{v},\, \boldsymbol{u}_{\mathsf{P}}, \emptyset)\rangle = |\, \boldsymbol{u}_{v} + \boldsymbol{v}_{v}|$$

. Since $\boldsymbol{u}_{v}$ and $\boldsymbol{v}_{v}$ are scalars in $\mathbb{F}$, we must have $|\, \boldsymbol{u}_{v} + \boldsymbol{v}_{v}| \leq |\, \boldsymbol{u}_{v}| + |\, \boldsymbol{v}_{v}|$, so

$$|\, \boldsymbol{u}_{v} + \boldsymbol{v}_{v}| \leq |\, \boldsymbol{u}_{v}| + |\, \boldsymbol{v}_{v}| = \langle\, \boldsymbol{u}\rangle + \langle\, \boldsymbol{v}\rangle.$$

We will now proceed by induction; let n be a positive integer for which the triangle inequality holds for all trees with a depth of k or less. Let's consider compatible trees, $\boldsymbol{u}$ and $\boldsymbol{v}$ whose depths are less than or equal to $n + 1$. Then

$$\begin{aligned}\langle\, \boldsymbol{u} + \boldsymbol{v}\rangle &= \langle(\, \boldsymbol{u}_{v} + \boldsymbol{v}_{v},\, \boldsymbol{u}_{\mathsf{P}},\, \boldsymbol{u}_{\boldsymbol{E}} \boxplus \boldsymbol{v}_{\boldsymbol{E}})\rangle \\ &= \Big[|\, \boldsymbol{u}_{v} + \boldsymbol{v}_{v}| + \sum_{\boldsymbol{e}\in\, \boldsymbol{u}_{\boldsymbol{E}} \boxplus \boldsymbol{v}_{\boldsymbol{E}}} \langle\, \boldsymbol{e}\rangle\Big].\end{aligned}$$

Observe that $|\, \boldsymbol{u}_{v} + \boldsymbol{v}_{v}| \leq |\, \boldsymbol{u}_{v}| + |\, \boldsymbol{v}_{v}|$, and that each of the addends in

$$\sum_{\boldsymbol{e}\in\, \boldsymbol{u}_{\boldsymbol{E}} \boxplus \boldsymbol{v}_{\boldsymbol{E}}} \langle\, \boldsymbol{e}\rangle$$

has a depth of n or less, so

$$\sum_{\boldsymbol{e}\in\, \boldsymbol{u}_{\boldsymbol{E}} \boxplus \boldsymbol{v}_{\boldsymbol{E}}} \langle\, \boldsymbol{e}\rangle \leq \sum_{\boldsymbol{e}\in\, \boldsymbol{u}_{\boldsymbol{E}}} \langle\, \boldsymbol{e}\rangle + \sum_{\boldsymbol{e}\in\, \boldsymbol{v}_{\boldsymbol{E}}} \langle\, \boldsymbol{e}\rangle.$$

This implies that

$$\langle \boldsymbol{u} + \boldsymbol{v} \rangle \leq \Big[|\, \boldsymbol{u}_v| + |\, \boldsymbol{v}_v| + \sum_{\boldsymbol{e} \in \boldsymbol{u}_{\boldsymbol{E}}} \langle \boldsymbol{e} \rangle + \sum_{\boldsymbol{e} \in \boldsymbol{v}_{\boldsymbol{E}}} \langle \boldsymbol{e} \rangle \Big];$$

rearranging we get

$$\langle \boldsymbol{u} + \boldsymbol{v} \rangle \leq \Big[|\, \boldsymbol{u}_v| + \sum_{\boldsymbol{e} \in \boldsymbol{u}_{\boldsymbol{E}}} \langle \boldsymbol{e} \rangle \Big] + \Big[|\, \boldsymbol{v}_v| + \sum_{\boldsymbol{e} \in \boldsymbol{v}_{\boldsymbol{E}}} \langle \boldsymbol{e} \rangle \Big]$$

and hence

$$\langle \boldsymbol{u} + \boldsymbol{v} \rangle \leq \langle \boldsymbol{u} \rangle + \langle \boldsymbol{v} \rangle.$$

□

Corollary 3.5.1. *The absolute value forms a seminorm on* $\boldsymbol{T}$.

Proof. Propositions, 3.5.1 and 3.5.2, are sufficient for the absolute value to be a seminorm on $\boldsymbol{T}$. □

At this point we should consider the elements $\boldsymbol{o} \in \boldsymbol{T}$ which are analogues of zero. We define the set $\mathfrak{O} = \{ \boldsymbol{o} \in \boldsymbol{T} : \langle \boldsymbol{o} \rangle = 0\}$, and observe that for any $\boldsymbol{e} \in \boldsymbol{T}$, and $\boldsymbol{o} \in \mathfrak{O}$ the equation $\langle \boldsymbol{e} + \boldsymbol{o} \rangle = \langle \boldsymbol{e} \rangle$ must hold.

Since $\boldsymbol{T}$ is a seminormed vector space, it is also a pseudometric space and we can induce a fully fledged metric space over the quotient space $\check{\boldsymbol{T}} = \boldsymbol{T}/\mathfrak{O}$. For simplicity, we identify the coset of $\mathfrak{O}$ with respect to O with $\check{\mathsf{O}}$, and we take the induced metric on the normed vector space $\check{\boldsymbol{T}}$, to be

$$\mathrm{d}(\check{\boldsymbol{u}}, \check{\boldsymbol{v}}) = \langle \check{\boldsymbol{u}} - \check{\boldsymbol{v}} \rangle \text{ for all } \check{\boldsymbol{u}}, \check{\boldsymbol{v}} \in \check{\boldsymbol{T}}$$

.

For the moment, we will satisfy ourselves with constructing a rng of these trees, and define a multiplicative operator without identifying a corresponding identity. First we define the pairwise multiplication of the elements in two sets of nodes which is analogous to the $\boxplus$ operator used in the addition of trees.

Definition 3.5.2. *For sets of nodes* $\boldsymbol{B}, \boldsymbol{C}$, *we define*

$$\boldsymbol{B} \boxtimes \boldsymbol{C} = \{ \check{\boldsymbol{f}} \cdot \check{\boldsymbol{g}} : \check{\boldsymbol{f}} \in \boldsymbol{B}, \check{\boldsymbol{g}} \in \boldsymbol{C}\}$$

where $\cdot$ *is the multiplicative operator for trees, and we also define*

$$\sigma(p, \boldsymbol{B}, \boldsymbol{C}) = \{\Sigma_{\check{\boldsymbol{f}}_{\mathsf{P}} = p} \check{\boldsymbol{f}} : \check{\boldsymbol{f}} \in \boldsymbol{B} \boxtimes \boldsymbol{C}\}$$

and

$$\boldsymbol{B} \boxdot \boldsymbol{C} = \{\sigma(p, \boldsymbol{B}, \boldsymbol{C}) : p \in \boldsymbol{L}(\boldsymbol{B} \boxtimes \boldsymbol{C})\}$$

Clearly, if either of the operands is the empty set, the result for both operators must also be the empty set. The purpose of $\boldsymbol{B} \boxdot \boldsymbol{C}$ *is to collect the compatible*

extension nodes within the product in much the same way as we collect terms in the product of two polynomials.

We can see that if tree multiplication is commutative, then $\boxtimes$ must also be commutative. The operator $\boxdot$ is also commutative, since the addition of trees is commutative.

Definition 3.5.3. *For nodes $\check{\boldsymbol{u}}, \check{\boldsymbol{v}} \in \check{\boldsymbol{T}}$, we define their product, $\check{\boldsymbol{u}} \cdot \check{\boldsymbol{v}}$, by*

$$\check{\boldsymbol{u}} \cdot \check{\boldsymbol{v}} = \begin{cases} \check{\mathbf{O}} & \text{if either of the nodes is } \check{\mathbf{O}} \\ (\check{\boldsymbol{u}}_{\nu}\, \check{\boldsymbol{v}}_{\nu},\ \check{\boldsymbol{u}}_{\mathsf{P}}\, \check{\boldsymbol{v}}_{\mathsf{P}},\ \check{\boldsymbol{u}}_{\boldsymbol{E}} \boxdot \check{\boldsymbol{v}}_{\boldsymbol{E}}) & \text{otherwise} \end{cases} \tag{3.1}$$

commutative,

Proposition 3.5.3. *Multiplication of trees in $\check{\boldsymbol{T}}$ is commutative.*

Proof. Suppose there is a number n such that multiplication is commutative for all trees $\check{\boldsymbol{u}}$, $\check{\boldsymbol{v}}$ such that $\text{depth}(\check{\boldsymbol{u}}), \text{depth}(\check{\boldsymbol{v}}) \leq n$.

Multiplication involving nodes with a depth of zero clearly commutes, so n may reasonably take the value 0.

In the case where both nodes are simple, it is evident that they must commute, since scalar multiplication commutes, and the multiplication of rational polynomials is commutative.

Suppose that one or both of the nodes has a depth of $n+1$. The extension sets of the nodes all have depths of n or less, so the elements of the extension set of the product must be independent of the order of the operators in the multiplication, and both scalar multiplication and polynomial multiplication commute. Hence the multiplication of nodes with a depth of $n+1$ must commute. By induction, we can say that trees of arbitrary depth commute with this definition of multiplication in $\check{\boldsymbol{T}}$. □

Proposition 3.5.4. *Multiplication of trees in $\check{\boldsymbol{T}}$ is associative.*

Proof. Suppose there is a number n such that multiplication is commutative for all trees $\check{\boldsymbol{u}}$, and $\check{\boldsymbol{v}}$, $\check{\boldsymbol{w}}$ such that $\text{depth}(\check{\boldsymbol{u}}), \text{depth}(\check{\boldsymbol{v}}), \text{and depth}(\check{\boldsymbol{w}}) \leq n$.

Multiplication involving nodes with a depth of zero is clearly associative, so n may reasonably take the value 0.

In the case where all of the nodes are simple, it is evident that they must be associative, since scalar multiplication is associative, and the multiplication of rational polynomials is associative.

Suppose that one or more of the nodes has a depth of $n+1$ or less. Then

$$\check{\boldsymbol{u}} \cdot (\check{\boldsymbol{v}} \cdot \check{\boldsymbol{w}}) = (\check{\boldsymbol{u}}_{\nu},\ \check{\boldsymbol{u}}_{\mathsf{P}},\ \check{\boldsymbol{u}}_{\boldsymbol{E}}) \cdot (\check{\boldsymbol{v}}_{\nu}\, \check{\boldsymbol{w}}_{\nu},\ \check{\boldsymbol{v}}_{\mathsf{P}}\, \check{\boldsymbol{w}}_{\mathsf{P}},\ \check{\boldsymbol{v}}_{\boldsymbol{E}} \boxdot \check{\boldsymbol{w}}_{\boldsymbol{E}})$$

Suppose that one or both of the nodes has a depth of $n+1$. The extension sets of the nodes all have depths of n or less, so the elements in the extension

set of their product must be independent of the order of the bracketting in the multiplication. Hence the multiplication of nodes with a depth of $n+1$ must be associative. By induction, we can say that multiplication of trees of arbitrary depth is associative. multiplication in $\check{\boldsymbol{T}}$. □

Proposition 3.5.5. *Tree-multiplication distributes over tree-addition in* $\check{\boldsymbol{T}}$.

Intuitively, the properties of operations on the scalar values and the rational polynomial labels of each node suggests that this must be true.

Proof. We want to show that for $\check{\boldsymbol{u}}$, $\check{\boldsymbol{v}}$, and $\check{\boldsymbol{w}} \in \check{\boldsymbol{T}}$, where nodes $\check{\boldsymbol{v}}$ and $\check{\boldsymbol{w}}$ are compatible, $\check{\boldsymbol{u}} \cdot (\check{\boldsymbol{v}} + \check{\boldsymbol{w}}) = \check{\boldsymbol{u}} \cdot \check{\boldsymbol{v}} + \check{\boldsymbol{u}} \cdot \check{\boldsymbol{w}}$ is true.

Multiplication by nodes which have a depth less than two clearly distributes over addition, since the multiplication of scalars and of polynomials distributes over their addition, and there are no extension sets to complicate matters. The case of multiplication where one of the trees has a depth of less than two and the other has an arbitrary depth also distributes, since the extension set

Let us consider the case of multiplication involving a node, $\check{\boldsymbol{u}}$, with a depth of two, that is to say that $\check{\boldsymbol{u}}$ has an extension set that contains only simple nodes. Then for compatible summands with depths of two or less,

$$\begin{aligned}
\check{\boldsymbol{u}} \cdot (\check{\boldsymbol{v}} + \check{\boldsymbol{w}}) &= (\check{\boldsymbol{u}}_\nu, \check{\boldsymbol{u}}_\mathsf{P}, \emptyset) \cdot ((\check{\boldsymbol{v}}_\nu, \check{\boldsymbol{v}}_\mathsf{P}, \check{\boldsymbol{v}}_{\boldsymbol{E}}) + (\check{\boldsymbol{w}}_\nu, \check{\boldsymbol{w}}_\mathsf{P}, \check{\boldsymbol{w}}_{\boldsymbol{E}})) \\
&= (\check{\boldsymbol{u}}_\nu, \check{\boldsymbol{u}}_\mathsf{P}, \emptyset) \cdot (\check{\boldsymbol{v}}_\nu + \check{\boldsymbol{w}}_\nu, \check{\boldsymbol{v}}_\mathsf{P}, \check{\boldsymbol{v}}_{\boldsymbol{E}} \boxplus \check{\boldsymbol{w}}_{\boldsymbol{E}}) \\
&= (\check{\boldsymbol{u}}_\nu(\check{\boldsymbol{v}}_\nu + \check{\boldsymbol{w}}_\nu), \check{\boldsymbol{u}}_\mathsf{P}\, \check{\boldsymbol{v}}_\mathsf{P}, \emptyset) \cdot (\check{\boldsymbol{v}}_\nu + \check{\boldsymbol{w}}_\nu, \check{\boldsymbol{v}}_\mathsf{P}, \check{\boldsymbol{v}}_{\boldsymbol{E}} \boxplus \check{\boldsymbol{w}}_{\boldsymbol{E}}) \\
&= ((\check{\boldsymbol{u}}_\nu\, \check{\boldsymbol{v}}_\nu + \check{\boldsymbol{u}}_\nu\, \check{\boldsymbol{w}}_\nu), \check{\boldsymbol{u}}_\mathsf{P}\, \check{\boldsymbol{v}}_\mathsf{P}, \check{\boldsymbol{v}}_{\boldsymbol{E}} \boxplus \check{\boldsymbol{w}}_{\boldsymbol{E}})
\end{aligned}$$

but $\check{\boldsymbol{u}}_\mathsf{P}$ must equal $\check{\boldsymbol{v}}_\mathsf{P}$, so

$$\begin{aligned}
&= (\check{\boldsymbol{u}}_\nu\, \check{\boldsymbol{v}}_\nu + \check{\boldsymbol{u}}_\nu\, \check{\boldsymbol{w}}_\nu), \check{\boldsymbol{u}}_\mathsf{P}\, \check{\boldsymbol{w}}_\mathsf{P}, \check{\boldsymbol{v}}_{\boldsymbol{E}} \boxplus \check{\boldsymbol{w}}_{\boldsymbol{E}}) \\
&= (\check{\boldsymbol{u}}_\nu\, \check{\boldsymbol{v}}_\nu, \check{\boldsymbol{u}}_\mathsf{P}\, \check{\boldsymbol{v}}_\mathsf{P}, \check{\boldsymbol{v}}_{\boldsymbol{E}}) + (\check{\boldsymbol{u}}_\nu\, \check{\boldsymbol{w}}_\nu, \check{\boldsymbol{u}}_\mathsf{P}\, \check{\boldsymbol{w}}_\mathsf{P}, \boldsymbol{w}_{\boldsymbol{E}}) \\
&= \check{\boldsymbol{u}} \cdot \check{\boldsymbol{v}} + \check{\boldsymbol{u}} \cdot \check{\boldsymbol{w}}
\end{aligned}$$

Note that this is independent of the depths of nodes $\check{\boldsymbol{v}}$ and $\check{\boldsymbol{w}}$.

Suppose then that there is an integer n such that multiplication of nodes with a depth of n or less distributes over addition, and we consider the case where our factors, $\check{\boldsymbol{u}}$, have depths of $n+1$ or less. Then

$$\begin{aligned}
\check{\boldsymbol{u}} \cdot (\check{\boldsymbol{v}} + \check{\boldsymbol{w}}) &= (\check{\boldsymbol{u}}_\nu, \check{\boldsymbol{u}}_\mathsf{P}, \check{\boldsymbol{u}}_{\boldsymbol{E}}) \cdot ((\check{\boldsymbol{v}}_\nu, \check{\boldsymbol{v}}_\mathsf{P}, \check{\boldsymbol{v}}_{\boldsymbol{E}}) + (\check{\boldsymbol{w}}_\nu, \check{\boldsymbol{w}}_\mathsf{P}, \check{\boldsymbol{w}}_{\boldsymbol{E}})) \\
&= (\check{\boldsymbol{u}}_\nu(\check{\boldsymbol{v}}_\nu + \check{\boldsymbol{w}}_\nu, \check{\boldsymbol{u}}_\mathsf{P}\, \check{\boldsymbol{v}}_\mathsf{P}, \check{\boldsymbol{u}}_{\boldsymbol{E}} \boxdot (\check{\boldsymbol{v}}_{\boldsymbol{E}} \boxplus \check{\boldsymbol{w}}_{\boldsymbol{E}}))
\end{aligned}$$

all of the nodes in the expression for the extension set are of depth less than or equal to n, so

$$= \check{\boldsymbol{u}} \cdot (\check{\boldsymbol{w}} + \check{\boldsymbol{v}})$$

□

Corollary 3.5.2. *$\check{\boldsymbol{T}}$ with tree addition and tree multiplication is a commutative rng.*

Proof. Propositions 3.5.3, 3.5.4, and 3.5.5 are sufficient to establish that it is a commutative rng. □

3.6 Discussion

This rng arose from attempts to capture the nuanced associations in survey questions of the form "Thinking about the weather forecast, how would you rate the chances of your favourite sporting team in the coming match?", and to be able to incorporate the sorts of possibly conflicting data into simulation models and to elicit in these models the dynamic interaction between simulated management strategies and the attitudes of modelled individuals and stakeholders as they respond to the the managed system. In this context, the difference between a conditional assessment and an unconditional assessment may be central to the behaviour we want from simulated entities. The labels of nodes in a tree could be associated with particular preconditions, and perturbations of a tree—perhaps by summation, multiplication, restriction, or extension—might provide appropriate ways of altering the perceptual and attitudinal orientation of the modelled entities. Alas this project did not continue, but the unfolding mathematics converged with the desire to control the mix of submodels as discussed in Chapter 1.

3.6.1 semantic coherence in the extensions

The way the extensions of a node influence the value of a node can be perverse if the data in these nodes aren't appropriate. The overriding rule is that extensions to a node *must be relevant to that node.* For extrapolation from surveys this is a matter associated with the coding of data, for simulation models (like MSE models or evaluating climate adaptation strategies) it is a matter of keeping track of influences and connections appropriately. I don't believe that this is entirely a trivial matter, but I also don't believe it is a complex one.

Example...

3.7 Example model

3.8 Possible domains

MSE models incorporating public opinion (surveys)

Relationship to Bayesian reasoning, Adaptive decision making (Baysian net?)

Adaptive behaviour (things can learn by adjusting weightings or adding new branches)

Constructing or assessing phylogenetic trees?

Chapter 4

Adaptive submodel selection in hybrid models

Randall Gray[1]

University of Tasmania

Simon Wotherspoon

University of Tasmania

Abstract

Hybrid modeling seeks to address problems associated with the representation of complex systems using "single-paradigm" models: where traditional models may represent an entire system as a cellular automaton, for example, the set of submodels within a hybrid model may mix representations as diverse as individual-based models of organisms, Markov chain models, fluid dynamics models of regional ocean currents, and coupled population dynamics models. In this context, hybrid modelers try to choose the best representations for each component of a model in order to maximize the utility of the model as a whole.

Even with the flexibility afforded by the hybrid approach, the set of models constituting the whole system and the dynamics associated with interacting models may be most efficient only in parts of the global state space of the system. The immediate consequence of this possibility is that we should consider adaptive hybrid models whose submodels may change their representation based on their own state and the states of the other submodels within the system.

[1]Published in *Frontiers in Environmental Science*, 20/8/2015
Corresponding author:`Randall.Gray@limnal.net` (Randall Gray),
`Simon.Wotherspoon@utas.edu.au` (Simon Wotherspoon)

This paper uses a simple example model of an artificial ecosystem to explore a hybrid model which may change the form of its component submodels in response to their local conditions and internal state relative to some putative optimization choices. The example demonstrates the assessment and actions of a "monitor" agent which adjusts the mix of submodels as the model run progresses. A simple mathematical structure is also described and used as the basis for a submodel selection strategy, and alternative approaches are briefly discussed.

4.1 Introduction

The case has been made for developing systems with submodels that change their representation according to their state. Vincenot et al. [2011b] identify reference cases describing the major ways system dynamics models (*SD*) and individual-based models (*IB*) can be coupled. Their final case, *SD-IB* model swapping, is exemplified in the models described by both Bobashev et al. [2007] and Gray and Wotherspoon [2012]. These papers argue that we can improve on conventional hybrid models, in terms of efficiency, fidelity, model clarity or execution speed by using an approach that allows the submodels themselves to change during a simulation. The last two papers implement simple models which demonstrate the approach, with correspondingly simple mechanisms to control transitions between different submodels.

Some authors argue that the explicit coupling of *SD* models and *IB* models may provide greater clarity and resolution in modeling [Vincenot et al., 2011b, Fulton, 2010]: parts of a model that are most clearly the result of aggregate processes are likely to be better suited to modeling with a *SD* approach. In contrast, the parts of a system where individuals have a significant influence on their neighbors [Botkin et al., 1972a] are better suited to an *IB* approach. This argument is closely tied to the notion of model fidelity. Following DelSole and Shukla [2010], we take *fidelity* to be the degree to which a model's trajectory is compatible with real trajectories. If our immediate goal is to maximize the utility of the set of submodels within a model as it runs, this must include the fidelity of the system in the decision process.

Measuring or estimating execution speed and numerical error are comparatively straight-forward, but determining model fidelity is not. Models with a high degree of fidelity should produce results which are consistent with observed data from real instances of the system they model across both a wide range of starting conditions and under the influence of ad hoc perturbations, such as fires through a forested domain. Model fidelity is addressed by DelSole and Shukla [2010] in the context of seasonal forecasting models. They explore the relationship between fidelity and skill using an information-theoretic approach. They describe *skill* loosely as the ability to reproduce actual trajectories, and they describe *fidelity* as measuring the difference between the distribution of model results and the distribution of real world results. They highlight the

attractiveness of mutual information and relative entropy as measures (or at least indices) of skill and fidelity, but they observe that in their domain, climate modeling, the necessary probability distributions are unknown.

The issues of fidelity and the attendant cost/benefit balance are central to the discussion in Bailey and Kemple [1992]. This paper assesses the costs and benefits of three different upgrades to an existing model designed to help determine the best mix of types of radios used in a military context; their objective is to prioritize implementation of the refinements of their model. The fundamental issues they address are substantially the same as issues that influence dynamic model selection.

The paper by Yip and Marlin [2004] compares three models used for real-time optimization of a boiler network: simple linear extrapolation from the system's current state, quadratic prediction with the coefficients based on historical data and updated at every step, and a detailed process model that corresponds closely with the physical elements of the modeled system. Their conclusion correlates the fidelity of the model with its ability to control the real-time optimization of the system. They explicitly note that there are real costs associated with the increased fidelity. These costs include model development and the need for more expensive sensors. They note that increasing fidelity in the model enabled the system to adapt to changing fuel more efficiently, and that when there were frequent changes in fuel characteristics the simpler models performed poorly.

The projects described in Little et al. [2006] and Fulton et al. [2011] both used hybrid models as a means of decreasing the run-time, and increasing the fidelity of the modeled contaminant uptake in simulated organisms. This was accomplished by mixing individual-based submodels and regional population-based systems models. Gray and Wotherspoon [2012] explicitly used changes in the representation of agents to improve the execution speed of a contamination tracking model, without losing the fidelity of the individual based uptake model. In this paper we will develop a more general strategy which may be appropriate for more complex systems.

4.2 Model organization

For clarity, we will take the term *niche* to refer to something in the model which could be modeled in several ways: a "porpoise" niche could be filled by many instances of an individual-based model, models of pods, or a regional *SD* model of the porpoises. This is essentially the same as the term *component* in Vincenot et al. [2011b]. The motivation for departing from this convention arose from confusion resulting from inadvertently using *component* both in a technical and non-technical sense. The close analogy between the nature of a niche in an ecosystem and the nature of a component as discussed in Vincenot et al. [2011b] suggested the choice of *niche*.

Each of the alternative ways of representing a niche can be viewed as a *submodel*, and the word *representation* will be used to reflect a particular choice of

submodel within a niche. An explicit instance of a submodel (such as a specific pod or an *SD* model) will be referred to as an *agent*. The *configuration* of the model at any moment consists of the particular set of submodels which fill the niches that comprise the model as a whole. For an adaptive hybrid model, there may be a large number of possible configurations and the selection of a "best" configuration is a complex matter.

Each agent running in a model must necessarily have data which can serve to characterize it for these assessments. This data would typically be some subset of its state variables, but the data alone may not be enough to base an assessment on: there may also be extrinsic data which play a role in a particular submodel's or agent's activity and impinges on its suitability. Then, the characterization of an agent — its *state vector* — is an amalgam of its own state and the state of other niches it interacts with, and it can be regarded as a point in the state space which the submodel is defined over.

A corresponding set of data characterizes a niche in the model; here, it is typically some appropriate aggregation of agent-level state variables (a biomass-by-size distribution, for example), relative rankings of the suitability of agents and alternative submodels, and indications of what extrinsic support all of the various alternatives require. This niche-level state vector provides the data needed for optimizing the configuration globally, and for managing the configuration when niche-wide effects become significant, for example, for an incipient epidemic.

Thus, there are three distinct levels of organisation which may influence the considerations regarding the current configuration, and inform any decision about what may need to change, namely

1. agent-level data need to be examined to determine how well suited each agent is to its current state and the context provided by the agents it interacts with,
2. a niche-level assessment which compares the utility of each of its current agents within a niche with their alternative submodels, and
3. a model-wide assessment which determines whether there are cross-agent conflicts or unmet needs arising from a particular configuration.

The state vectors which form the domains of submodels and niches are loci in appropriate state spaces and can be encoded as an elements in appropriate vector spaces. The mathematical tools to manipulate these state vectors can then be applied to calculate the distances between two states, the similarity of loci which represent models or niches, or to identify trends or clusters.

4.2.1 Implications of changing configurations

At a basic level, hybrid models are designed to represent entities or processes in the real world in a way which brings more clarity, efficiency, or fidelity that may be possible with more traditional approaches. Adaptive hybrid models,

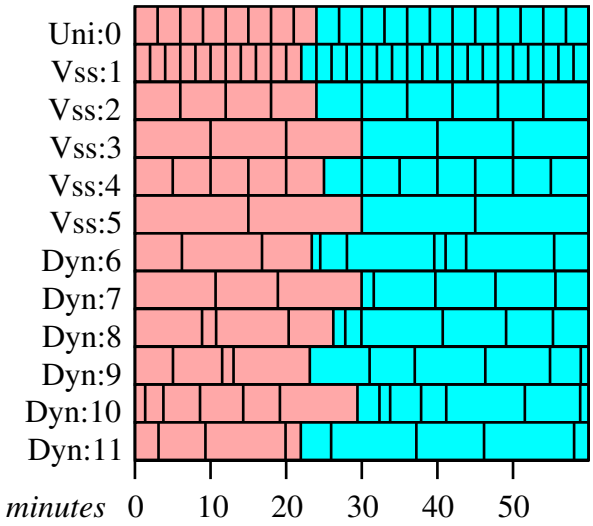


Figure 4.1: Time scheduling strategies. Red boxes represent time steps that have already passed, blue boxes represents scheduled time steps that have not yet been run. "Uni:" and "Vss:" submodels are members of a uniform or variable speed splitting submodels and require uniform time steps, and "Dyn:" submodels have adaptive time steps.

implicitly acknowledge that the appropriate representation may change through time. An important consequence is that when a submodel in a niche changes, it may trigger changes in representation elsewhere in the model.

We might consider an example where an *SD* submodel which represents the prey for an *SD* based predator changes to *IB* submodels. It seems reasonable to expect the representation of the predator might follow suit. This may change the spatial resolution, the fineness of the "quantities" represented, and possibly the time steps associated with the predators and prey. Disparities in either of the first two are simple enough to deal with: modelers routinely use interpolation as a means of removing inappropriate edges, or generating subscale data, for example. Changes in an agent's time step can have a dramatic causal influence on the subsequent simulation.

Chivers [2009] discusses how individual-based models are sensitive to when state variables are updated. In his discussion, the issue arises as a result of when the probability of a predator-prey interaction is calculated relative to when the prey are removed from the system, though similar effects are also likely to occur in other contexts. The temporal sensitivity of submodels' interactions needs careful examination in order to construct submodels that proceed through time coherently and interact correctly.

Multi-agent models must have strategies to manage the agents as they step from the start of the simulation to its end. The simplest method is to make everything within the model use the shortest time step required. This is computationally inefficient in a heterogeneous model.

A better approach is the technique of variable speed splitting, such as in Walters and Martell [2004] and many others. (Figure 4.1) This approach allows models to step through time in different intervals by dividing the largest interval required into smaller steps that are more appropriate for the submodels with naturally shorter time scales. While models with uniform time steps are a trivial example of this approach, variable speed splitting is almost as simple and much more efficient. This technique can keep the subjective times of a set of agents moderately consistent, but ad hoc stepping changes would still seem to be awkward or difficult.

Both of these strategies may be subject to artifacts arising from the sequence in which agents are given their time step. The general class of model errors of the sort described in Chivers [2009] arise as a consequence of structure of the processing across the set of agents in a simulation. *IB* models which process agents species-by-species will be particularly vulnerable to these sorts of artifacts, since there will be an implicit advantage or disadvantage to being early in the list. Similarly, advantage or disadvantage can arise when there is a change in representation, perhaps from an *SD* submodel to an *IB* submodel; a shorter time step in this situation may introduce a great many small time steps which agents may exploit. This kind of problem can be overcome by introducing a randomizing process within each time step. Early versions of the variable speed splitting model in Lyne et al. [1994a] suffered from predator-prey artifacts arising from a naïve introduction of predators and prey into the list of agents, and such randomizing was introduced to minimize the effects. In situations where the time steps of the interacting agents differ, implementing a randomization strategy may require a significant increase in the complexity of the system to accommodate irregular stepping through the lists of agents, or a significant change in the basic structure of the model.

Gray et al. [2006b] and Fulton et al. [2009a] describe models that have a well developed approach to coordinating agents using adaptive time steps. In these models agents may set their own time steps to intervals that are suitable for their current activity or role. This strategy can readily incorporate submodels with uniform time steps, or collections that employ a variable speed splitting strategy. When agents interact, they either explicitly become synchronous before interaction occurs by setting their time steps appropriately and waiting, or they implicitly acknowledge that there is a temporal mismatch. (Figure 4.1)

While some agents should be given execution priority (such as an agent which models ocean currents), most agents will have their execution order within a time step randomized, effectively preventing a large class of execution order dependent artifacts. The associated overhead in the most recent work, Gray et al. [2006b], Gray and Wotherspoon [2012], is marginally higher than one would expect from single-stepping or variable speed stepping systems, but the advantages arising from the ability to ensure synchrony and change time steps in response to environmental stimulus outweigh the small computational overhead. This last approach seems likely to be the most appropriate for a general hybrid model that supports swapping models.

General adaptive hybrid models must have a mechanism for scheduling each

agent's execution which keeps the cohort of agents roughly synchronous, and it should able to handle changes in an agent's time step when the agent changes its representation; where possible, agents should also be designed so that they may run at other time steps as well as their own preferred time step so they can become synchronous and interact at the appropriate temporal scale with other agents.

4.2.2 Systematically adjusting the model configuration

A model's configuration should only change when there is an overall benefit in the efficiency or fidelity of the system. A straightforward way of determining this is to have a monitoring routine that runs periodically, polling the agents, and ranking likely configurations according to their relative benefit or cost. This means that each submodel would need a way to provide, to the monitor, a measure of its current suitability, and to indicate what it needs from other niches.

Algorithm 1 Basic processing pass for the monitor

```
for all niches do
    for all submodels in the niche do
        for all agents in the submodel do
            generate agent state vector
            generate the submodel state vector
                note extrinsic requirements
        end for
    end for

    generate niche state vector
end for

Run niche-level assessment
Flag any whole of model issues
for all candidate configurations do
    Deprecate untenable configuration
    Adjust for unavoidable extrinsic
        requirements
end for

Select best indicated configuration
```

The last step in Algorithm 1 is deliberately vague.

Algorithm 1 illustrates a possible assessment pass for a monitor, though how appropriate it may be is an open question. Configuration ranking for the example model will be cast in terms of evaluating an objective function based on elements of the vector space of tree elements described in the Appendix.

A monitor may have large number of potential candidate configurations, but we

would like to keep the actual number quite low. The example model described below has a global domain associated with a particular representation, along with local domains (subregions of the global domain) which are associated with finer scale representations of the modeled entities. The set of potential candidate trees could be quite large; in practice we reduce the number by casting the candidate trees in a more general way – including trees representing particularly good representations and particularly poor representations: the first to steer the configuration toward good choices, and the second to drive it away from poor choices. We can use the hierarchical organisation (whole-model, niche, submodel, agent) to help limit our search space, as well as the geographic context of the agents (whole-domain, local cell, immediate-locus).

The sets of candidate trees which are associated with particular configurations will need to be crafted carefully as a part of the model design. These trees reflect the modelers understanding of the strengths and weaknesses of each of the submodels (or sets of different submodels) which may be employed.

Exactly how a monitoring routine is integrated into the model framework is a subjective choice best left to the team implementing the models, but one very attractive option is to implement the monitor as an agent in the system. This would allow the monitor to assess its own performance and the needs of other agents with respect to its own suitability with the option of swapping itself our for a montitor which implements some alternative strategy.

4.3 The example model

The purpose of the example model described below, is to provide a context for a discussion of the dynamics associated with a hypothetical simulation using this model. The ends of the spectrum between *SD* models and *IB* models are represented, and the environment is unrealistically simple in order to keep us from being swamped by detail.

The model consists of a spatially explicit environment that is partitioned into nine cells (Figure 4.2). The biotic elements consist of plants, fruit, seeds, herbivores, and carnivores. The herbivores feed on the plants and their fruit; and carnivores prey upon juvenile herbivores. The plants and herbivores are interdependent: fruit is the sole diet for juvenile herbivores and the plants need juvenile herbivores to make the seeds viable by eating the fruit.

The representations are equation-based *SD* models of the interactions between the plants and animals and *IB* models for plants and animals. The *SD* submodels model the biomass with respect to size, for plants and animals, or simply numeric quantities for fruit and seeds, and they can operate at either the global or cell-sized scale. Modeling biomass in this way makes it possible to minimize the loss of fidelity incurred by swapping from *IB* agents to *SD* agents and visaversa, since we preserve more of the essential nature of the populations. A more detailed description of the *SD* agents is presented in the *Supplementary Material*.

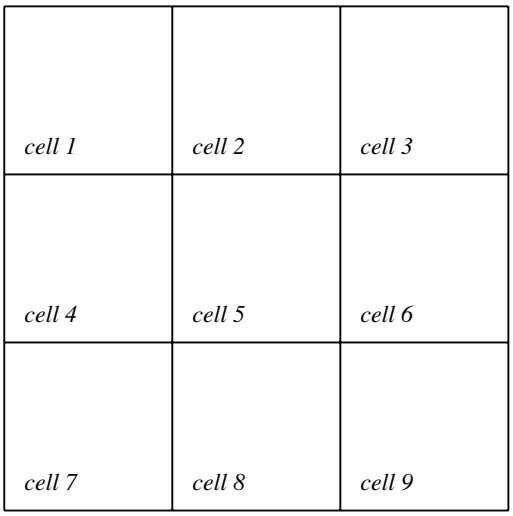


Figure 4.2: The model domain is divided into nine cells. An *SD* agent is associated with each of these cells and with the domain as a whole. Any *IB* agents which are created during the simulation will be associated with one cell at any given time.

Fruit and seeds

Fruit and seeds are treated somewhat differently to the rest of the niches. They exist principally as numbers of entities that are updated as a result of the activities of other, more explicit *SD* or *IB* models. There are explicit routines that deal with uniquely "fruit" and "seed" processing to handle spoilage and germination, respectively.

For fruit and seeds we have the following relationships

$$dN_{\mathtt{F}}(t) = \mathit{Production} - \mathit{Spoilage} - \mathit{FruitEaten}$$

and

$$dN_{\mathtt{S}}(t) = s * \mathit{FruitEaten} - \left(1 - \frac{N_{\mathtt{P}}(t)}{K_{\mathtt{P}}}\right) \mathit{Germ}.$$

where $N_{\mathtt{P}}(t)$ is the biomass of plants at time t, and $K_{\mathtt{P}}$ is the carrying capacity of the pertinent domain (either global or cell-based). The processing for fruit

Algorithm 2 Basic processing pass for fruit

$N_{\mathtt{F}} \leftarrow N_{\mathtt{F}} - (\mathit{Spoilage}_{\mathtt{F}} \cdot N_{\mathtt{F}})$

is quite simple and consists only of applying "spoilage"; no reference to other agents in the system is required, and only the number of fruit is adjusted as a result (Algorithm 2). Seed models will adjust their "seed count" as well as

Algorithm 3 Basic processing pass for seeds

$NewTreeCount \leftarrow Germination \cdot SeedCount$
$SeedCount \leftarrow SeedCount - (NewTreeCount$
$\quad + Spoilage_{\mathtt{S}} \cdot SeedCount)$
generate $NewTreeCount$ new plant agents and introduce them into the system

the biomass distribution for plants in their time step, according to the level of germination. Germination is probabilistic as is the size of the plant a germinated seed becomes in its pass, though the distribution of possibly sizes is quite restrained (Algorithm 3).

SD representations

Each of the niches has an integral equation expressing the change in biomass for a given size; an animal's equation is of the form[2]

$$\begin{aligned} dN_{\mathtt{A}}(t,x) = & Growth\&Starv + Repr \\ & - PredMort - NatMort. \end{aligned}$$

We do not include migration terms in the *SD* models, since that will be addressed by the *IB* forms. The assumption is that the *SD* representation is most appropriate when population levels are moderately high, and there is adequate food; under these conditions, we will assume that the net migration associated with a domain will be close to zero.

Plants are represented by similar equations, namely

$$\begin{aligned} dN_{\mathtt{P}}(t,x) = & \left(1 - \frac{N_{\mathtt{P}}(t)}{K_{\mathtt{P}}}\right)[Growth + Germ] \\ & - PredMort - NatMort \end{aligned}$$

where $N_{\mathtt{P}}(t,x)$ is the biomass of plants of size x at time t.

The important state variables for the *SD* are, for each domain, the biomass-by-size distributions for plants, herbivores and carnivores, and the raw numbers of fruit and viable seeds.

The system of equations described in the *Supplementary Material* is evaluated using a fourth order Runge-Kutta algorithm; the numbers of fruit and seeds, and both the global and cell-based biomass distributions for plants and animals are updated at the end of the calculation. The model will adjust the values in the global and cell-based models to allow data from models running with better resolution (usually more localized models) (Algorithm 4) to take precedence.

Most of the important parameters and many of the functions associated with the life history of the modeled entities are not specified. This way we may

[2]See the *Supplementary Material* for a more detailed set of equations.

Algorithm 4 Basic processing pass for the *SD* models

```
for all agents in this domain do
    Incorporate quantities that are
        controlled in other agents
    Run Runge-Kutta4
    Update only quantities that are
        controlled by this agent
end for
```

consider possible trajectories without being tied to a particular conception or parameterization of the system.

IB representations

Individual-based representations for plants, herbivores and carnivores follow the pattern in Little et al. [2006]; fruit and seeds are only modeled in the *SD* representation, though their numbers are modified by the activities of the herbivores irrespective of how those herbivores are represented.

4.3.1 *IB* Plants

Plants maintain a reference to their cell, their location, a mass and a peak mass. If a plant's mass drops below a certain proportion ($\mathtt{P}_{M\Omega}$) of its peak mass, it dies — this provides a means for the herbivores to drive the plant population to local extinction.

We will suppose that plants grow according to a sigmoidal function with some reasonable asymptote and intermediate sharpness; fruiting occurs probabilistically as in the *SD* representation.

Algorithm 5 Basic processing pass for plants

```
if (Mass ≥ P_Mature) ∧ (P_Fruits ≥ rnd_0,1) then
    AddFruit(P_ρ Mass^(2/3))
end if
if (Mass ≤ P_MΩ PkMass) ∨ (Ω_indP < rnd_0,1) then
    Die
else
    Mass ← Γ_P(δt, Mass)
    if Mass > PkMass then
        PkMass ← Mass
    end if
end if
```

The plant agent goes through the steps in Algorithm 5 in each of its time steps. In the algorithm, $\Gamma_{\mathtt{P}}(\delta t, mass)$ is an analogue of the probability of a plant growing from one size to another from the *SD* representation, $\mathtt{P}_{Mature}$ is

the parameter that indicates the mass a plant must be before it fruits, $\mathtt{P}_{Fruits}$ is the probability of a mature plant fruiting, and $\mathtt{P}_{\rho}$ is the amount of fruit relative to the fruiting area. The routine AddFruit updates the models representing fruit in the domain.

4.3.2 *IB* Animals

Like the plants, animals maintain a reference to their cell, their location, and a mass. They also maintain several variables that are associated with foraging or predation, namely the amount of time until they need to eat (*Sated*), and the amount of time they have been hungry (*Hungry*).

Animals will grow while they do not need to eat and will only forage when they are hungry. Reproduction happens in a purely probabilistic way once the animal is large enough, and the young are not cared for by the parents.

Animal movement is constrained so that they will tend to stay within their nominated home cell, only migrating (changing their home cell to an adjacent cell) when food becomes scarce or if the population exceeds some nominated value and causes crowding.

The analogues of the mechanisms for growth and starvation in the *SD* representation are quite different to those of the *IB* version. In the *SD* models, starvation and growth occur as a result of the relative population levels of the consumer and the consumed rather than the local availability of food.

There are no real programmatic differences between the *IB* representations of herbivores and carnivores; their differences lie in their choices of food and the way their "time-to-eat" variable is initially managed. InDiViduAl-based, newborn carnivores begin with a long time till they need to eat. This reflects a reliance on some unmodeled foodstuff until they are large enough to prey on the juvenile herbivores. In contrast, the juvenile herbivores must begin eating fruit immediately, and only switch to foraging on plants when they are larger (but before they can reproduce). For both species, if the amount of time they have been hungry exceeds a particular value, H_{Ω} or C_{Ω}, the individual dies.

So, if we take $\mathtt{A}$ to represent either carnivores ($\mathtt{C}$) or herbivores ($\mathtt{H}$) below, then the processing pass for an animal is shown in Algorithm 6, where $\mathtt{A}_{moveT}$ is the amount of time an animal can be hungry before it migrates, $\mathtt{A}_{\Omega}$ is the amount of time it takes for the animal to starve, $\mathtt{A}_{EatLimit}$ is the most the animal can eat as a proportion of its mass, $\mathtt{A}_{RepSize}$ is the minimum size an animal may breed at and $\mathtt{A}_{RepP}$ is the probability of reproducing. The routines $\textsc{PreyPresent}_{\mathtt{H}}$ and $\textsc{Eat}_{\mathtt{H}}$ have different cases for juvenile and adult herbivores, since juveniles prey upon fruit, and the seeds from the fruit they eat need to be accounted for in the appropriate places. There is a similar issue with juvenile carnivores. Their *preylist* will always be set to a value that indicates that they may eat as much as they like, and the corresponding call to $\textsc{Eat}_{\mathtt{C}}$ will handle this value appropriately.

Algorithm 6 Basic processing pass for herbivores and carnivores

if $(\Omega_{\mathrm{ind}_\mathtt{A}} > \mathrm{rnd}_{0,1}) \vee (\mathit{Hungry} \geq \mathtt{A}_\Omega)$ **then**
  DIE
end if
$\mathit{PreyList} \leftarrow \textsc{PreyPresent}_\mathtt{A}(\mathit{Locus}, \mathit{Mass})$
if $\mathit{Sated} \geq 0$ **then**
  $\mathit{Mass} \leftarrow \mathit{Mass} + \textsc{Growth}_\mathtt{A}(\mathit{mass}, \delta t)$
else if $(\mathit{Hungry} \geq 0) \wedge (\mathrm{len}(\mathit{PreyList}) > 0)$ **then**
  $\mathit{Sated} \leftarrow \textsc{Eat}(\mathit{PreyList}, \mathtt{A}_{\mathit{EatLimit}}, \mathit{mass})$
  $\mathit{Hungry} \leftarrow 0$
  $\mathit{ForageCt} \leftarrow 0$
else if $(\mathit{Hungry} \geq 0) \wedge \mathit{len}(\mathit{PreyList}) = 0)$ **then**
  FORAGE
  $\mathit{ForageCt} \leftarrow \mathit{ForageCt} + 1$
else if $(\mathit{Hungry} \geq \mathtt{A}_{\mathit{moveT}}) \vee \textsc{Crowded}_\mathtt{A}$ **then**
  $\textsc{Migrate}_\mathtt{A}(\mathit{Locus})$
else
  if $(\mathit{mass} \geq \mathtt{A}_{\mathit{RepSize}}) \wedge (\mathtt{A}_{\mathit{RepP}} \geq \mathrm{rnd}_{0,1})$ **then** $\textsc{Reproduce}_\mathtt{A}(\mathit{Locus})$
  end if
end if

4.3.3 The monitor and model dynamics

The following may be typical of the types of situations that could or should cause changes in the configuration:

- *Low population* —- If, in an *SD* representation, the number of individuals filling a niche (either explicitly taken from a distribution, or estimated using a mean and a biomass) drops below a nominated value, then the biomass in that niche should be converted to *IB* agents representing those individuals. This type of change is motivated by the observation that at low population levels the assumption that we can treat the population as having uniform access to resources (or be uniformly available to predators) breaks down;

- *High population* —- If a niche in a cell is represented by *IB* agents and the number of individuals exceeds a (higher) nominated value, the biomass those agents represent should be subsumed by the distribution in the local *SD* submodel. The change in representation is attractive here for two reasons: an equation-based representation will be much faster, and *SD* submodels are arguably simpler to calibrate;

- *Starvation risk* — If the mean amount of time an animal in a cell spends *hungry* in a cell exceeds half of $\mathtt{A}_\omega$ (or some other nominated time), the prey biomass must convert to *IB* agents if it isn't already so (bearing in mind that this isn't pertinent for fruit). This mean is calculated by averaging the means of each animal in the cell. If this is triggered, it indicates

that the biomass of the prey species is sparse enough that homogeneity assumption is unlikely to hold;

- *Relative biomass* — If the biomass available for predation is represented in a local *SD* agent and its density drops below some proportion of the minimum required to support the predators in the domain, the prey species should convert its biomass into *IB* agents and, if the predator is represented by a *SD* agent, it should also convert to an *IB* form. If the biomasses are such that the effective predation rate is unsustainable, the mixing assumption is unlikely to hold.

The pertinent data for conditions will be periodically reported to the monitor through a set of status trees. The trees are able to represent single entities, nested entities and aggregates equally well, and can preserve structural information which may also be used in the comparison of these trees. One of the basic elements we can easily incorporate into a submodel's status tree is the agent's own assessment of its competence relative to its state-vector and its local conditions. This measure of "self-confidence" can probably be maintained at little computational cost for most agents, and may be the most significant component in a monitor's assessment. The *high* and *low* population level conditions can clearly be determined by the agent in question; it can set its level of self-confidence upward or downward as appropriate. *Starvation* can also be encoded in the relevant node of an agent's status tree, but since starvation alone may not indicate a problem with the way the entity is represented, it probably wouldn't reduce the value for its confidence.

A starvation trigger may usually arise as a natural consequence of the population dynamics, but it may also occur when there is a mismatch in representations which has not been adequately addressed in the design stage. The final condition based on the relative biomasses is one which properly lies in the realm of the monitor – it would be quite inefficient for each of the candidate animals to be querying their prey for available biomass, summing the result, and then noting the need for change.

The monitor will primarily use the confidence values associated with agents and their niches, and the distance from trees which describe the state of the model or its set of submodels to trees which describe "known good" configurations. With data obtained directly from the agents in the system and from alternative representations it generates status trees,

- $\check{\boldsymbol{\tau}}^{\Sigma}_{sn}$, is a candidate status tree tied to a specific configuration. The serial number, sn, ties it to a configuration with that serial number,
- $\check{\boldsymbol{\tau}}^{\Sigma}_{d}$, is a candidate tree which represents the current state of a domain,

  $\boldsymbol{\tau}^{\Sigma}_{t}$, an aggregate tree for the whole domain at time t,

  $\boldsymbol{\tau}^{\Sigma}_{SD(n),t}$, aggregate trees for each cell, $n \in \{1, \ldots, 9\}$,

  $\boldsymbol{\tau}_{\mathsf{R}(i),t}$, specific status trees for each agent,

$\boldsymbol{\tau}_{R,t}$, specific status trees for a representation R for each representation associated with a niche,

and

$\hat{\boldsymbol{\tau}}_{\mathsf{R}(i),t}$, candidate trees for all possible representations of each agent i,

at the beginning of each of its steps. The model may have a mix of *SD* and *IB* representations, and some of the trees will have to incorporate data from many agents ($\boldsymbol{\tau}_t^{\Sigma}$, any of the $\hat{\boldsymbol{\tau}}_{\mathsf{R}(i),t}$, and $\boldsymbol{\tau}_{R,t}$, for example). A candidate tree is a status tree which represents an alternative submodel in a niche, and candidate trees are generated for specific agents and for each niche. When the monitor begins to generate status or candidate trees for a given agent, it first looks to see if it has generated an appropriate tree already. If it finds one, it incorporates or adjusts the tree appropriately; perhaps by incorporating the agent's biomass and size into the tree's data. We will also denote the configuration of a domain (global or local) with $\check{\boldsymbol{\tau}}_c^{\Sigma}$ where c identifies the domain in question.

The monitor assesses the trees by calculating aggregate values of particular attributes, comparing the trees' divergences from allegedly ideal configurations, and by looking how uniform groups are – groups of individuals that are all very similar are good candidates for simpler representations.

We can calculate the average confidence value from any of these trees by evaluating

$$\frac{\langle\overline{\mathrm{mask}}(\,\boldsymbol{\tau}, \mathit{confidence}, 0)\rangle}{\mathrm{supp}(\overline{\mathrm{mask}}(\,\boldsymbol{\tau}, \mathit{confidence}, 0))},$$

for example. The trees and functions to manipulate them are described in the Appendix.

Now let us consider what a simulation might look like. Figure 4.3 provides an overview of the configuration of the system as our hypothetical simulation runs. The model begins with eleven agents (not counting the monitor). The monitor runs its first step generating the status trees: $\boldsymbol{\tau}_0^{\Sigma}$, which characterizes the model in aggregate, $\boldsymbol{\tau}_{SD(0),0}^{\Sigma}, \ldots, \boldsymbol{\tau}_{SD(9),0}^{\Sigma}$, which record the aggregate state of the ten *SD* submodels, the aggregate status tree for the *IB* agent, $\boldsymbol{\tau}_{IB(0),[}^{\Sigma}9]$, status trees for the *SD* submodels: $\boldsymbol{\tau}_{\mathsf{SD}(0),0}{}^{-}\boldsymbol{\tau}_{\mathsf{SD}(10),0}$, the status tree for the lone carnivore, $\boldsymbol{\tau}_{IB(11),0}$, followed by the trees which represent alternative agents: $\hat{\boldsymbol{\tau}}_{SD(0),0}{}^{-}\hat{\boldsymbol{\tau}}_{SD(10),0}$ and $\hat{\boldsymbol{\tau}}_{IB(11),0}$. As mentioned earlier, there is only the single tree for agent 11 (the carnivore) since its alternative representation is embodied in $\hat{\boldsymbol{\tau}}_{SD(10),0}$. During the simulation a simulated fire will occur.

The first steps which must be taken before ranking of potential configurations is to find the sets of candidate trees which best approximate the current configuration at both the global and cell levels. We do this by calculating a similarity index or a distance which indicates how close each of the candidate trees are to the configuration of each of the domains. There are many ways we could do this: for an index which only considers structural similarity we might use something like the simple function

$$\mathrm{ssim}(\,\boldsymbol{c},\, \boldsymbol{\tau_d}) = \frac{\mathrm{overlap}(\,\boldsymbol{c},\, \boldsymbol{\tau_d})}{\max(\|c\|_{\mathsf{T}}, \|\tau_d\|_{\mathsf{T}})},$$

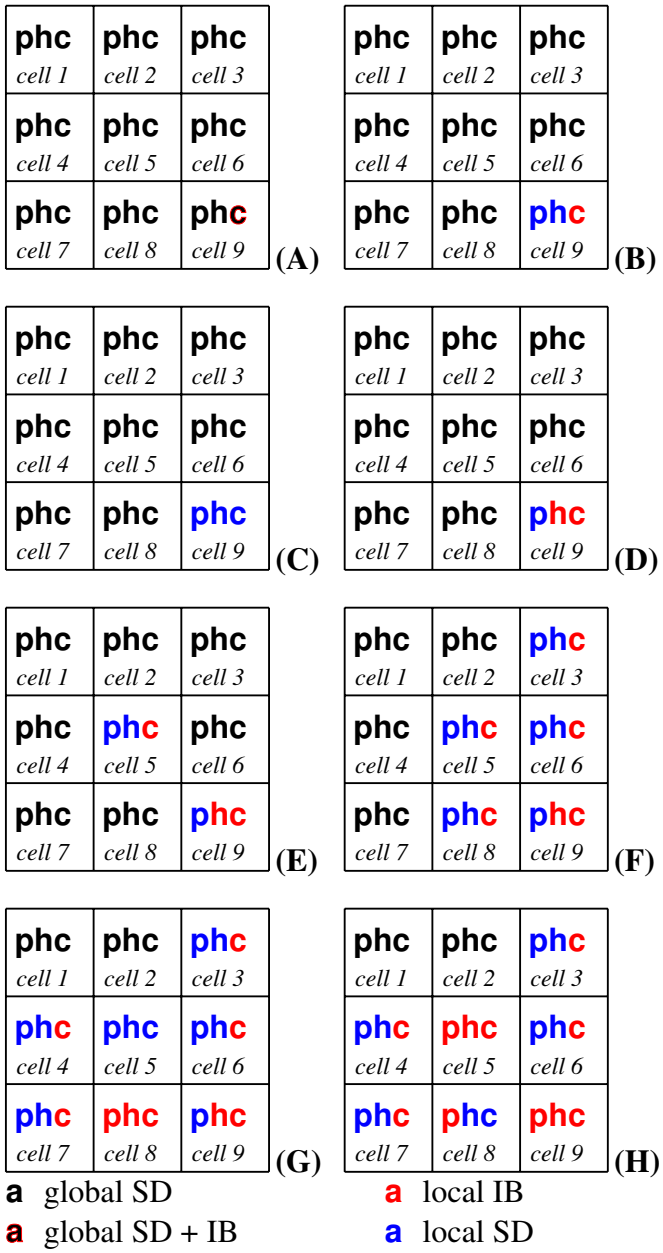


Figure 4.3: The color of the **p**,**h** and **c** indicate an agent's current representation within a cell at various points in the description of a simulation. In each, a black symbol indicates that the biomass of plants (**p**), herbivores (**h**) or carnivores (**c**) is modeled with the global *SD* agent, a blue symbol indicates that the biomass is modeled with a cell's *SD* agent, and red indicates that an *IB* model is being used. Symbols composed of two colors indicate that more than one representation is currently controlling portions of the relevant biomass.

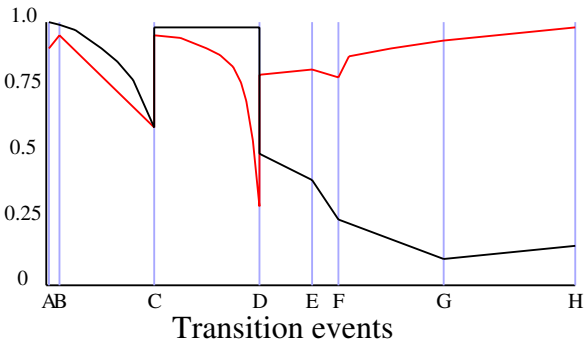


Figure 4.4: Normalized indexes of execution speed (black) and fidelity (red) the against configuration changes through time associated with Figure 4.3

but for a more comprehensive treatment which factors values which are incorporated into the candidate and status trees we might apply the $\Delta(,)$ or d functions described in the Appendix. The d function is a well-defined distance over the vector space of trees, while the Δ(,f)unction is an index of similarity that incorporates structural characteristics as well as the numerical distance between compatible subtrees. To refine such an analysis we could apply mask and $\overline{\text{mask}}$ to select only the relevant parts of the candidate and status trees.

So to assess the configuration of a domain, we would use our chosen measure to construct a set of the results of applying an optimisation function, *opt*, to each of the candidate trees and their similarity to the current configuration. So if $\boldsymbol{S}$ is the set of all serial numbers for candidates, $\check{\boldsymbol{\tau}}_d^{\Sigma}$ is the status tree fo the current domain, and is the , we calculate

$$(C) = \{(\delta(\check{\boldsymbol{\tau}}_d^{\Sigma}, \check{\boldsymbol{\tau}}_i^{\Sigma}), i) : \forall i \in \boldsymbol{S}\},$$

and this is used to generate

$$\boldsymbol{C}^* = \{(\text{opt}(\check{\boldsymbol{\tau}}_i^{\Sigma}), c, \check{\boldsymbol{\tau}}_i^{\Sigma}, i) : \forall (c, i) \in \boldsymbol{C}\}$$

where δ stands for our chosen measure of similarity.

The elements in $\boldsymbol{C}^*$ are then assessed by the monitor, and the best permissible candidate is selected. If there is only a small improvement on the current configuration, $\check{\boldsymbol{\tau}}_d^{\Sigma}$, the monitor will leave the configuration as it is; otherwise, the monitor would then manage the creation of new agents to replace less optimal representations and manage the exchange of state data.

So the early phase of our simulation might begin like so:

1. Both of the aggregate trees $\boldsymbol{\tau}_0^{\Sigma}$ and $\boldsymbol{\tau}_{SD(9),0}^{\Sigma}$ indicate that there is an *IB* agent in their domain and that their *SD* representation does not perform well for the indicated biomass. Both the status and candidate trees for agent 11, $\boldsymbol{\tau}_{11(0),}$, $\boldsymbol{\tau}_{IB(11),0}$ and $\hat{\boldsymbol{\tau}}_{IB(11),0}$, indicate that it is confident that it can represent the biomass, and that there are no immediate unmet requirements from other agents. Figure 4.3 (**A**)

2. The monitor assesses the trees against a prepared set of configurations: each of the alternative configurations (including the current configuration) is compared to a set of prepared, "efficient" configurations. The configuration of cell 9, $\check{\boldsymbol{\tau}}^{\Sigma}_{9}$, notes global *SD* representations for plants and herbivores. This configuration is ranked lower than the alternative which has an individual based model for carnivores and a local *SD* submodel for the other entities in the cell. The monitor makes this change in configuration, and informs the global *SD* agent that it is no longer controlling the biomasses in cell 9. Figure 4.3 (**B**)

3. The model may run for some time without any change in configuration. Both the herbivores and carnivores breed. The increased execution speed between **C** and **D** in Figure 4.4 is a result of a change in representation: the number of carnivores, recorded in $\boldsymbol{\tau}^{\Sigma}_{SD(C),t_4}$, reaches a point that prompts the monitor to convert them to an *SD* form. Figure 4.3 (**C**)

4. The biomass of carnivores has increased significantly by the time the model reaches **D** in Figure 4.4, and they are now eating all the young herbivores; as a result the carnivore population is now prey-limited, and the *Relative biomass* condition is triggered. Both the carnivore and herbivore populations are converted to *IB* representations. Notice that dynamics in the fidelity in Figure 4.4 around **D** arise from the collapse of the carnivore's prey, followed by the increase in fidelity after the representation change at **D**. Figure 4.3 (**D**)

5. A carnivore, agent 43, has been hungry ($(Hungry \geq \mathtt{A}_{moveT}$) and has migrated to the cell 5 (noted in $\boldsymbol{\tau}_{IB(43),t_5}$). As occurred in cell 9 at step 2, the monitor converts plants and herbivores in cell 5 to a local *SD* representation, with *IB* carnivores. Figure 4.3 (**E**)

6. A lot of activity has occurred in this monitor interval: a *Starvation risk* is triggered in cell 9 because too many of the carnivores are hungry (many of the $\boldsymbol{\tau}_{IB(n),t_7}$ trees indicate that the elapsed time without eating is greater than *Hungry*). There has been more migration to cells 5,6 and 8 from cell 9 (more of the $\boldsymbol{\tau}_{IB(n),t_7}$ trees indicate residence in new cells), and a chance migration has introduced a carnivore into cell 3 from cell 5. Cells 3,6 and 8 are converted to local *SD* and *IB* representations as happened in step 3. Figure 4.3 (**F**)

7. The population of carnivores in cell 9 crashes as a result of migration and the scarcity of prey, (reported in $\check{\boldsymbol{\tau}}^{\Sigma}_{9}$) The *IB* juvenile herbivores are patchy and harder to find, so only a few carnivores are getting enough to eat. There will be many $\boldsymbol{\tau}_{IB(n),t_{10}}$ which indicate hunger or death due to starvation. The monitor cleans up the dead agents. There are chance migrations from cell 5 into cells 4 and 7 (in $\boldsymbol{\tau}^{\Sigma}_{SD(4),0}$ and $\boldsymbol{\tau}^{\Sigma}_{SD(7),0}$). A fire begins in cell 8, moving through cell 5: biomass loss in all niches causes all niches to shift to *IB* representations. Figure 4.3 (**G**)

8. Juvenile herbivores are reappearing in cell 9, but the available plant biomass (recorded in $\boldsymbol{\tau}_{\mathsf{SD}(9),t_{11}}$) has dropped due to reduced germination rates, triggering the *Relative biomass* condition in cell 9 causing the

plants to convert to an *IB* representation. The fire in cell 8 has killed all animal biomass in the cell; they *do not* return to the global *SD* representation because their status trees diverge by too much. Instead, they convert to local *SD* representations (which represent zero biomass quite efficiently). Plants remain as *IB* agents The fire spreads to cell 5. Figure 4.4 shows a modest increase in execution speed between **G** and **H** due to the population losses associate with the fire. Figure 4.3 (**H**)

- . . . the simulation continues

4.4 Discussion

Adaptive hybrid models *can* be constructed so that each submodel is aware of its other representations and is able to change form as appropriate [Gray and Wotherspoon, 2012]. This approach requires each model to have a reasonably close coupling with its alternative representations, and the burden of instrumenting (and maintaining) the necessary code quickly becomes untenable in complex models. Worse, it removes the possibility of more subtle configuration management that can accept poor performance in one part of a system in exchange for much better performance elsewhere. It seems that a guiding principle should be that in an adaptive hybrid model, each representation should know only as much about the rest of the model as it *must* know, and no more. The facility for a submodel to delve into the workings of other submodels, or the workings of the model as a whole, decreases the clarity that hybrid modeling makes possible, and opens avenues for unwanted, unanticipated behavior.

The major argument in favor of closely integrated representations for submodels is that it makes common (or at least similar) state variables easy to maintain across representations, even in the face of many representation changes. It is an attractive arguement, but the long term consequence is an ever growing burden of code maintenance.

Constructing hybrid models isn't significantly more complex than constructing traditional models. Adaptive hybrid models of the sort described in this paper will require a more significant investment in the design of a monitoring routine, and in the crafting of appropriate sets of candidate configurations. The transition dynamics such a model will exhibit depend on the sets of candidate configurations, and it seems likely that a combination of analysis and experimentation may be the most effective way to develop a set of useful configurations. The hybrid models associated with Lyne et al. [1994a], Little et al. [2006], Fulton et al. [2009a] were built by extending the repertoire of ways of representing elements of the ecosystem or the anthropic components rather than wholescale redesign and replacement.

We can imagine an ideal adaptive hybrid model, where any state information which must be passed on is accompanied by an appropriate, opaque parcel of code to perform the maintenance. As long as the monitor knows what information each of these maintenance interfaces needs, they can be updated each time the monitor interrogates the agent which has control of the state data.

This is a readily attainable ideal: many programming languages support first class functions with closures, and these features are precisely what we need to address this problem. *Scheme, Python, ML, Common Lisp, Lua, Haskell*, and *Scala* all have first order functions with closures and, hence, the capacity to build model systems with this capability.

The state vectors and their supporting maintenance procedures can be treated as data and passed in lists associated with the status trees. If a monitor decides to swap representations, the accumulated lists of maintenance functions may be passed on to the new representation. A new representation inherits a maintenance list with variables that are part of its native state, it can claim them as its own and continue almost as though it had been running the whole time. In this way, a new representation doesn't need to know anything about its near kin, only that it must be able to run these black-box functions that come from other submodels, and to pass them on when required.

It may seem that this concentrates the global domain knowledge in the monitor, but this is not really the case. The monitor knows how to blindly query agents for state data and to the data in maintenance procedures. The monitor also knows how to recognize and rank characterizations of the states of the submodels or niches and to use those data to select a configuration.

The domain knowledge is encapsulated in the sets of targets the monitor matches the current configuration against, and in the heuristic triggers (such as *Starvation risk*) associated with a submodel or niche.

The essential problems any monitor is likely to deal with are problems of set selection (recognition, pattern matching...) and optimisation. These are common tasks: web searches, voice recognition, and route planning have become ingrained parts of modern society. Like route planning, the monitor needs to be able to reassess the "optimal" strategy as an ongoing process.

There are many options to choose from to rank configurations. A few of the likely candidates include

- using an objective function to evaluate each of the possible configurations,
- selecting a configuration based on decision trees,
- using neural nets to match model states and direct us to an appropriate configuration,
- using Bayesian networks to determine the most likely candidate,

and

- using support vector machines to select the target/configuration pairs.

In writing this paper, one of the vexing difficulties has been finding a suitable mathematical representation which would allow comparisons between configurations, submodel states and the states of niches. We need proxies that describe models and configurations of models in a way that we may readily understand,

manipulate and reason about, and being able to deal with submodels which are, in themselves, adaptive hybrid models, seems to be a naturally desirable trait. The vector space of trees described in the Appendix has some nice properties, and may be directly useful with many of the options above: it forms a commutative ring (without necessarily having a unit), and would naturally inherit the body of techniques which only require the properties of such a ring.

4.5 Conclusion

There are still some major obstacles to developing a fully fledged adaptive hybrid model which is generic enough to tackle instances as varied as marine ecosystem modeling and urban planning. Foremost is a relative lack of real examples. The simulation of the hypothetical model[3] has tried to expose the character of an adaptive hybrid model which uses a monitor to manage the configuration of the system. There are parts of the description of the example system which are conspicuous by their absence; this is largely because they lie in almost wholly uncharted water. As a modeling community, we need to develop a wide range of approaches to how a model may assess the relative merits of a set of configurations. Many of the mechanisms we need for adaptive hybrid models already exist, but are found in domain specific models, and in wholly different domains, such as search engines and GPS navigation.

Establishing a suitable mathematical representation for model configurations which gives us access to well developed techniques for set selection, pattern recognition and component analysis would seem to be almost as urgent as adaptive hybrid examples of real systems.

Acknowledgments
The authors would like to thank two anonymous reviewers whose comments have improved the paper immeasurably. Thanks also go to a patient and understanding editor at Frontiers, and to Dr Tony Smith, who gave up a weekend to work a scientifically and grammatically fine toothed comb through the paper. The responsibility for any mistakes, awkward sentences, or places where it just does not make sense now rests completely with the lead author.

[3]The model described in this paper is currently under development and will be made freely available when it has been completed.

Appendix

Mathematical definitions

The trees we use are members of a normed vector space: we can add them, find out how far apart they are and interpolate between them. In principle, we can run clustering algorithms to find configurations that are similar, and identify when a model has left one cluster and entered another.

Definition .0.1. *Let $\mathcal{S}$ be a set of labels, and let $\mathbb{F}$ be a field like the real or complex numbers. Then we define a node $\boldsymbol{n}$ as a triplet of the form $(s, v, \boldsymbol{E})$, with $v \in \mathbb{F}$, $s \subset \mathcal{S}$, and the set $\boldsymbol{E}$ is a (possibly empty) set of nodes of the same form with the restriction that no two nodes in $\boldsymbol{E}$ may have the same label in their first ordinate. We also define the triple $\mathbf{O} = (\emptyset, 0, \emptyset)$, which we will call the null tree, and define $\boldsymbol{T}$ to be the union of the set of all trees composed of a finite number of these nodes.*

The ordinates of $\boldsymbol{u} = (\boldsymbol{u}_{\mathsf{P}}, \boldsymbol{u}_{v}, \boldsymbol{u}_{\boldsymbol{E}})$ in $\boldsymbol{T}$ correspond to its label*,* value*, and* extension set*. An element of $\boldsymbol{u}_{\boldsymbol{E}}$ will be called an* extension*.*

In our discussion, it will help to have a few more descriptive terms.

A node with an empty extension set is called a *simple* node, if this node happens to be a root node, then it is a simple tree.

Two trees, $\boldsymbol{u}, \boldsymbol{v} \in \boldsymbol{T}$ are called *compatible* if either $\boldsymbol{u}_{\mathsf{P}} = \boldsymbol{v}_{\mathsf{P}}$ or at least one of $\boldsymbol{u}$ and $\boldsymbol{v}$ is $\mathbf{O}$.

We define the *depth* of a tree with:

$$\text{depth}(\boldsymbol{u}) = \begin{cases} 0 & \text{if } \boldsymbol{u} = (\emptyset, 0, \emptyset) = \mathbf{O} \\ 1 & \text{if } \boldsymbol{u} \text{ is a simple node} \\ 1 + \max(\{\text{depth}(\boldsymbol{v}) : \forall\, \boldsymbol{v} \in \boldsymbol{u}_{\boldsymbol{E}}\}) & \text{otherwise.} \end{cases}$$

We will also define for $\boldsymbol{u} \in \boldsymbol{T}$,

$$\text{trim}(\boldsymbol{u}) = \begin{cases} \mathbf{O} & \text{if } \boldsymbol{u} = \mathbf{O} \\ \mathbf{O} & \text{if } \boldsymbol{u} \text{ is simple} \\ (\boldsymbol{u}_{\mathsf{P}}, \boldsymbol{u}_{v}, \{\text{trim}(\boldsymbol{e}) : & \\ \qquad \forall\, \boldsymbol{e} \in \boldsymbol{u}_{\boldsymbol{E}}\} \setminus \{\mathbf{O}\}) & \text{otherwise.} \end{cases}$$

The cardinality of a tree is the number of nodes it contains. Specifically,

$$\|\boldsymbol{u}\|_{\mathsf{T}} = \begin{cases} 0 & \text{if } \boldsymbol{u} = \mathbf{O} \\ 1 + \sum_{\boldsymbol{e} \in \boldsymbol{u}_{\boldsymbol{E}}} \|\boldsymbol{e}\|_{\mathsf{T}} & \text{otherwise.} \end{cases}$$

Simple nodes are the only nodes that have a cardinality of one, and $\mathbf{O}$ is the only node or tree with a cardinality of zero.

The support of a tree is the number of nodes which have a non-zero value.

$$\operatorname{supp}(\boldsymbol{u}) = \begin{cases} 0 + \sum_{\boldsymbol{e}\in \boldsymbol{u}_{\boldsymbol{E}}} \operatorname{supp}(\boldsymbol{e}) & \text{if } \boldsymbol{u}_{v} = 0 \\ 1 + \sum_{\boldsymbol{e}\in \boldsymbol{u}_{\boldsymbol{E}}} \operatorname{supp}(\boldsymbol{e}) & \text{otherwise}. \end{cases}$$

Related is the fundamental support of a tree, which only counts nodes with no zero valued nodes in their connection to the root node

$$\operatorname{fund}(\boldsymbol{u}) = \begin{cases} 0 & \text{if } \boldsymbol{u}_{v} = 0 \\ 1 + \sum_{\boldsymbol{e}\in \boldsymbol{u}_{\boldsymbol{E}}} \operatorname{fund}(\boldsymbol{e}) & \text{otherwise}. \end{cases}$$

Clearly the support of a tree, $\operatorname{supp} \boldsymbol{u}$, must lie in the domain $[0, \|\boldsymbol{u}\|_{\mathsf{T}}]$ and $\operatorname{fund} \boldsymbol{u} \leq \operatorname{supp}(\boldsymbol{u})$.

The *overlap* between two trees is defined

$$\operatorname{overlap}(\boldsymbol{u}, \boldsymbol{v}) = \begin{cases} 0 & \text{if } \boldsymbol{u} = \mathbf{O} \text{ or } \boldsymbol{v} = \mathbf{O} \text{ or } \boldsymbol{u}_{\mathsf{P}} \neq \boldsymbol{v}_{\mathsf{P}} \\ 1 + \sum\limits_{\substack{\boldsymbol{e}\in \boldsymbol{u}_{\boldsymbol{E}} \\ \boldsymbol{f}\in \boldsymbol{v}_{\boldsymbol{E}}}} \operatorname{overlap}(\boldsymbol{e}, \boldsymbol{f}) & \text{otherwise} \end{cases}$$

Clearly two trees, $\boldsymbol{u}$ and $\boldsymbol{v}$, are compatible if and only if $\operatorname{overlap}(\boldsymbol{u}, \boldsymbol{v}) \neq 0$; they will be said to *completely overlap* if $\|\boldsymbol{u}\|_{\mathsf{T}} = \|\boldsymbol{v}\|_{\mathsf{T}} = \operatorname{overlap}(\boldsymbol{u}, \boldsymbol{v})$.

We can now define scalar multiplication, and tree addition.

Definition .0.2. *Take $a \in \mathbb{F}$ and $\boldsymbol{u} \in \boldsymbol{T}$, then*

$$a\,\boldsymbol{u} = \begin{cases} \mathbf{O} & \text{if } \boldsymbol{u} = \mathbf{O} \\ (\boldsymbol{u}_{\mathsf{P}}, a\,\boldsymbol{u}_{v}, \{a\,\boldsymbol{f} : \boldsymbol{f} \in \boldsymbol{u}_{\boldsymbol{E}}\} \setminus \{\mathbf{O}\}) & \text{otherwise.} \end{cases} \tag{1}$$

Definition .0.3. *Let $\boldsymbol{u}$ and $\boldsymbol{v}$ be compatible elements of $\boldsymbol{T}$. Then taking the symbol $+$ to be addition in the field $\mathbb{F}$, we extend it to addition in $\boldsymbol{T}$ so that for nodes $\boldsymbol{u}$ and $\boldsymbol{v}$,*

$$\boldsymbol{u} + \boldsymbol{v} = \begin{cases} \mathbf{O} & \text{if } \boldsymbol{u} = \boldsymbol{v} = \mathbf{O} \\ \boldsymbol{u} & \text{if } \boldsymbol{u} \neq \mathbf{O} \text{ and } \boldsymbol{v} = \mathbf{O} \\ \boldsymbol{v} & \text{if } \boldsymbol{u} = \mathbf{O} \text{ and } \boldsymbol{v} \neq \mathbf{O} \\ (\boldsymbol{u}_{\mathsf{P}}, \boldsymbol{u}_{v} + \boldsymbol{v}_{v}, \emptyset) & \text{if } \boldsymbol{u}_{\boldsymbol{E}}, \boldsymbol{v}_{\boldsymbol{E}} = \emptyset \\ \Big(\boldsymbol{u}_{\mathsf{P}}, \boldsymbol{u}_{v} + \boldsymbol{v}_{v}, (\{\boldsymbol{f} + \boldsymbol{g} : \boldsymbol{f} \in \boldsymbol{u}_{\boldsymbol{E}} \text{ and } \boldsymbol{g} \in \boldsymbol{v}_{\boldsymbol{E}} \text{ and } \boldsymbol{f}_{\mathsf{P}} = \boldsymbol{g}_{\mathsf{P}}\} & \\ \quad \cup \{\boldsymbol{f} : \boldsymbol{f} \in \boldsymbol{u}_{\boldsymbol{E}} \text{ and } \boldsymbol{f}_{\mathsf{P}} \neq \boldsymbol{g}_{\mathsf{P}} \forall\, \boldsymbol{g} \in \boldsymbol{v}\} & \\ \quad \cup \{\boldsymbol{g} : \boldsymbol{g} \in \boldsymbol{v}_{\boldsymbol{E}} \text{ and } \boldsymbol{g}_{\mathsf{P}} \neq \boldsymbol{f}_{\mathsf{P}} \forall\, \boldsymbol{f} \in \boldsymbol{u}\}) \setminus \{\mathbf{O}\}\Big) & \text{otherwise.} \end{cases} \tag{2}$$

Definition .0.4. *Let $\boldsymbol{u}$ and $\boldsymbol{v}$ be compatible elements of $\boldsymbol{T}$. Then we define inner-multiplication between the two nodes*

$$\boldsymbol{u} \cdot \boldsymbol{v} = (\boldsymbol{u}_{\mathsf{P}}, \boldsymbol{u}_{v}\, \boldsymbol{v}_{v}, \{\boldsymbol{f} \cdot \boldsymbol{g} : \boldsymbol{f} \in \boldsymbol{u}_{\boldsymbol{E}}, \boldsymbol{g} \in \boldsymbol{v}_{\boldsymbol{E}} \text{ and } \boldsymbol{f}_{\mathsf{P}} = \boldsymbol{g}_{\mathsf{P}}\} \setminus \{\mathbf{O}\}) \tag{3}$$

It can be shown that $\boldsymbol{T}$ with scalar multiplication and tree addition forms a vector space. This isn't quite enough to give us distances between trees, however, so we define a semi-norm

Definition .0.5. *Let $\boldsymbol{u}$ be an element of $\boldsymbol{T}$. Then we can define a semi-norm over $\boldsymbol{T}$*

$$\langle \boldsymbol{u} \rangle = \begin{cases} 0 & \text{if } \boldsymbol{u} = \mathbf{O} \\ |\, \boldsymbol{u}_v| & \text{if } \boldsymbol{u}_{\boldsymbol{E}} = \emptyset \\ |\, \boldsymbol{u}_v| + \sum_{\boldsymbol{e} \in \boldsymbol{u}_{\boldsymbol{E}}} \langle \boldsymbol{e} \rangle & \text{otherwise.} \end{cases}$$

It is clear that $\langle \boldsymbol{u} \rangle$ will always be non-negative, and the only shortcoming is that we can have a node $\boldsymbol{u}$ with $\langle \boldsymbol{u} \rangle = 0$, but $\boldsymbol{u} \neq \mathbf{O}$. In order to turn this into a normed vector space we take the set $\mathfrak{O} = \{ \boldsymbol{u} : u \in \boldsymbol{T} \text{ and } \langle \boldsymbol{u} \rangle = 0 \}$ and we construct an equivalence relation on $\boldsymbol{T}$ by the rule $[\boldsymbol{u}] \equiv [\boldsymbol{v}]$ if and only if there exist $\boldsymbol{z_u}, \boldsymbol{z_v} \in \mathfrak{O}$ such that $\boldsymbol{u} + \boldsymbol{z_u} = \boldsymbol{v} + \boldsymbol{z_v}$. It can be shown that scalar multiplication, tree addition, and the semi-norm behave appropriately with respect to the equivalence classes. This means that if we identify elements of $\boldsymbol{T}$ with their equivalence class, then we can take $\langle \boldsymbol{u} \rangle$ to be a norm and that it induces a distance function

$$\mathrm{d}(\boldsymbol{u}, \boldsymbol{v}) = \langle \boldsymbol{u} - \boldsymbol{v} \rangle$$

.

Definition .0.6. *We define the functions* mask *and* $\overline{\text{mask}}$ *that set the values associated with particular nodes in a tree to $v \in \mathbb{F}$. Specifically, if $\mathcal{L} \subset \boldsymbol{T}$, then*

$$\text{mask}(\boldsymbol{u}, \mathcal{L}, v) = \begin{cases} (\boldsymbol{u}_{\mathsf{P}}, v, \{\text{mask}(\boldsymbol{f}, \mathcal{L}, v) : \boldsymbol{f} \in \boldsymbol{u}_{\boldsymbol{E}}\}) & \text{if } \boldsymbol{u}_{\mathsf{P}} \in \mathcal{L} \\ (\boldsymbol{u}_{\mathsf{P}}, \boldsymbol{u}_v, \{\text{mask}(\boldsymbol{f}, \mathcal{L}, v) : \boldsymbol{f} \in \boldsymbol{u}_{\boldsymbol{E}}\}) & \text{otherwise} \end{cases} \tag{4}$$

and

$$\overline{\text{mask}}(\boldsymbol{u}, \mathcal{L}, v) = \begin{cases} (\boldsymbol{u}_{\mathsf{P}}, v, \{\overline{\text{mask}}(\boldsymbol{f}, \mathcal{L}, v) : \boldsymbol{f} \in \boldsymbol{u}_{\boldsymbol{E}}\}) & \text{if } \boldsymbol{u}_{\mathsf{P}} \notin \mathcal{L} \\ (\boldsymbol{u}_{\mathsf{P}}, \boldsymbol{u}_v, \{\overline{\text{mask}}(\boldsymbol{f}, \mathcal{L}, v) : \boldsymbol{f} \in \boldsymbol{u}_{\boldsymbol{E}}\}) & \text{otherwise.} \end{cases} \tag{5}$$

$\text{mask}(\boldsymbol{u}, \mathcal{L}, 0)$ *would return a tree similar to $\boldsymbol{u}$, but all its nodes that have labels in $\mathcal{L}$ would have values of zero.*

We also define several functions which prune or select a child from a tree's extension set. This function returns only the part of $\boldsymbol{u}$ which overlaps $\boldsymbol{p}$,

$$\text{excise}(\boldsymbol{u}, \boldsymbol{p}) = \begin{cases} (\boldsymbol{u}_{\mathsf{P}}, \boldsymbol{u}_v, \{\text{excise}(\boldsymbol{f}, \boldsymbol{g}) : \boldsymbol{f} \in \boldsymbol{u}_{\boldsymbol{E}} \text{ and } \boldsymbol{g} \in \boldsymbol{p} \text{ and } \boldsymbol{f}_{\mathsf{P}} = \boldsymbol{g}_{\mathsf{P}}\} \setminus \{\mathbf{O}\}) & \text{if } \boldsymbol{u}_{\mathsf{P}} = \mathcal{L} \\ \mathbf{O} & \text{if } \boldsymbol{u}_{\mathsf{P}} \neq \boldsymbol{p}_{\mathsf{P}} \\ (\boldsymbol{u}_{\mathsf{P}}, \boldsymbol{u}_v, \emptyset) & \text{otherwise,} \end{cases} \tag{6}$$

and this one either returns an appropriately labelled child from the extension set (a branch), or $\mathbf{O}$.

$$\text{child}(\boldsymbol{u}, \ell) = \begin{cases} \boldsymbol{f} & \text{if } \boldsymbol{f}_{\mathsf{P}} = \ell \text{ and } \boldsymbol{f} \in \boldsymbol{u}_{\boldsymbol{E}} \\ \mathbf{O} & \text{otherwise.} \end{cases} \tag{7}$$

We now finish with a definition of a function, $\Delta(,)$, that gives us a measure of the degree of divergence between two trees.

Definition .0.7. *The degree of deviation between two trees,* $\boldsymbol{u}$ *and* $\boldsymbol{v}$ *is given by the expression*

$$\Delta(\boldsymbol{u}, \boldsymbol{v}) = \begin{cases} \|u\|_{\mathsf{T}} + \|v\|_{\mathsf{T}} - 2\,\text{overlap}(\boldsymbol{u}, \boldsymbol{v}) + \langle \boldsymbol{u} - \boldsymbol{v} \rangle & \textit{if } \boldsymbol{u} \textit{ and } \boldsymbol{v} \textit{ are compatible} \\ \|u\|_{\mathsf{T}} + \|v\|_{\mathsf{T}} & \textit{otherwise.} \end{cases} \tag{8}$$

The rationale behind this definition is that if trees $\boldsymbol{u}$ *and* $\boldsymbol{v}$ *are identical, then* $\Delta(\boldsymbol{u}, \boldsymbol{v})$ *will be zero. We also want nodes that aren't common to both trees to count as differences.*

A more nuanced rule for the example presented in Chapter **??** might be constructed by relating the number of individual-based agents needed to the area of a population's potential overlap with the contaminant plume.

Appendix A

Explicit Model

A.1 New Section

Explicit model based on the model of previous chapter.

Appendix B

Conclusion

B.1 New Section

Appendix A

An implementation of a model with adaptive configuration

A.1 Code environment and requirements

gambit-c slib modified tiny-clos (sclos) supporting libraries

A.2 Tiny-CLOS – an implementation of a restricted set of CLOS-like facilities

Tiny CLOS was developed at Xerox Parc by Gregor Kiczales in 1992 [Kiczales] and it has served as the starting point for many other object systems for the scheme programming language, such as Swindle, STklos, GOOPS and SOS, for example [Barzilay, STk, GNU, a,b]. This work is based on version 1.7 of the canonical tiny-clos; its major difference from the canonical release is that a number of files have been consolidated into one file, "`sclos.scm`".

A.3 Source code

file-by-file ... prune out everything that isn't called!

A.3.1 framework

```
;-  Identification and Changes

;--
;  framework-preamble%.scm -- Written by Randall Gray
;  Initial coding:
;     Date: 2012.11.12
;
;  History:
;


;;; #|
;;; The code in this file is supposed to make things a little easier.

;;;
;;;  The model framework uses SCLOS by Gregor Kiczales, and released
;;;  by Xerox Corp. in 1992. None of the code from sclos.scm has been
;;;  included in the other files which form the modelling framework.

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;; This file *must* be included, not loaded. ;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;



;;; |#
```

```
;-  Code

;; See also Kiczales, Gregor; Jim des Rivieres; Daniel G. Bobrow (July
;; 30, 1991). The Art of the Metaobject Protocol. The MIT Press. ISBN
;; 978-0262610742.


(include "sclos.scm")


(define-macro (with-exit name . body)
  `(call-with-current-continuation
    (lambda (,name) ,@body)))

(define-macro (state-variables . lst) `'(,@lst))
;; This makes it clear when classes are defining state vars

(define-macro (no-state-variables) `'())
;; This makes it clear when classes are defining NO state vars

(define-macro (inherits-from . iflst) `(list ,@iflst))

;; Support for the "isa?" calls and for a mapping from the class
;; object to a name

(define-macro (register-class cname)
  `(class-register 'add ,cname ',cname))


;; Remember, if you try to unquote classtype without splicing, the
;; interpreter tries to evaluate it.
```

```
;; cpl... class precedent list
(define-macro (isa? self . classtype)
  `(let ((ancestors (class-cpl (class-of ,self))))
     (if (apply orf
                (map (lambda (x) (member x ancestors))
                     (list ,@classtype)))
         #t #f)
     )
  )


(define-macro (declare-method name descr . otherstuff)
  `(define ,name
     (let ((g (make-generic)))
       (generic-register 'add g ',name)
       g)))

(define-macro (default-object-initialization name . otherstuff)
  (if (null? otherstuff)
      `(add-method
        initialize
        (make-method (list ,name)
                     (lambda (initialize-parent self args)
                       (initialise self '())
                       ;; call "parents" last to make the
                       ;; initialisation list work
                       (initialize-parent)
                       )))
      `(add-method
        initialize
```

```
        (make-method (list ,name)
                     (lambda (initialize-parent self args)
                       (initialise self (list ,@otherstuff))
                       ;; call "parents" last to make the
                       ;; initialisation list work
                       (initialize-parent)
                       )))

  ))


(define-macro (default-initialization name . otherstuff)
  (if (null? otherstuff)
      `(add-method
        initialize
        (make-method (list ,name)
                     (lambda (initialize-parent self args)
                       (initialise self '())
                       ;; call "parents" last to make the
                       ;; initialisation list work
                       (initialize-parent)
                       )))
      `(add-method
        initialize
        (make-method (list ,name)
                     (lambda (initialize-parent self args)
                       (initialise self (list ,@otherstuff))
                       ;; call "parents" last to make the
                       ;; initialisation list work
                       (initialize-parent)
                       )))
```

```
  ))


;; This works like so:
;;(sclos-method <parent1> (methname) body)
;;(sclos-method <parent1> (methname arg1 . rest) body)
;;(sclos-method <parent1> (methname . rest) body)
;;(sclos-method (<parent1>) (methname) body)
;;(sclos-method (<parent1>) (methname arg1 arg2) body)
;;(sclos-method (<parent1> <parent2>) (methname) body)
;;(sclos-method (<parent1> <parent2>) (methname arg1 arg2..) body)
;;(sclos-method (<parent1>) (methname arg1 arg2 . rest) body)
;;(sclos-method (<parent1> <parent2>) (methname arg1 arg2 . rest) body)

;; "(????-parent)" can be used to invoke the parent methods in the
;; body, where ???? is the value of methname so for a method called
;; "grok" you would use "grok-parent" to invoke the parent method
;; "self" is the name of the agent within the body of the method the
;; state variables are not available but you can use "my" and
;; "set-my!" or "slot-ref" and "slot-set!"

(define-macro (object-method . madness)
  ;; Call like (object-method classname (mname * margs) . body) where
  ;;   the margs doesn't include the call-parent-method and 'self' is
  ;;   the placeholder
  (if (< (length madness) 3)
      (error "object-method: bad arguments to method call " madness))

  (if (not (and (pair? (cadr madness)) (symbol? (caadr madness)) ))
      (error "method: bad (mname args...) in (method ...) "
```

```
           (caadr madness)))

(if (not (eq? (cadadr madness) 'self))
    (error "object-method: missing 'self' placeholder in definition "
           (caadr madness)))

(let ((classlst (if (list? (car madness))
                    (car madness)
                    (list (car madness))))
      (mname (caadr madness))
      (star (cadadr madness))
      (margs (cddadr madness))
      (body (cddr madness))
      )
  `(add-method
    ,mname
    (make-method
     (list ,@classlst)
     (lambda (,(string->symbol
                (string-append
                 (symbol->string mname) "-parent")) self ,@margs)
       (let ((my (lambda (x) (slot-ref self x)))
             (set-my! (lambda (x y) (slot-set! self x y)))
             )
         ,@body)
       )
     )
    )
  )
)
```

```
;; "(parent-body)" can be used to invoke the parent's body --- sclos
;; handles any arguments that might be needed: "self" is the name of
;; the agent within the body code "t" is the current model time dt is
;; the requested dt the state variables are not directly available --
;; use "my" or "set-my!", "slot-ref" or "slot-set!"



(define-macro (sclos-method . madness)
  ;; Call like (sclos-method classname (mname * margs) . body) where
  ;; the margs doesn't include the call-parent-method and 'self' is
  ;; the placeholder
  (if (< (length madness) 3)
      (error "sclos-method: bad arguments to method call " madness))
  (if (not (and (pair? (cadr madness)) (symbol? (caadr madness)) ))
      (error "method: bad (mname args...) in (method ...) "
             (caadr madness)))
  (if (not (eq? (cadadr madness) 'self))
      (error "sclos-method: missing 'self' placeholder in definition "
             (caadr madness)))

  (let ((classlst (if (list? (car madness))
                      (car madness)
                      (list (car madness))))
        (mname (caadr madness))
        (star (cadadr madness))
        (margs (cddadr madness))
        (body (cddr madness))
```

```
        )
    `(add-method
      ,mname
      (make-method (list ,@classlst)
                   (lambda (,(string->symbol
                              (string-append (symbol->string mname)
                                             "-parent"))
                            self ,@margs)
                     (let ((my (lambda (x) (slot-ref self x)))
                           (set-my! (lambda (x y)
                                      (slot-set! self x y)))
                           (kernel (slot-ref self 'kernel))
                           )
                       ,@body)
                     )
                   )
      )
    )
  )


;; "(parent-body)" can be used to invoke the parent's body --- sclos
;; handles any arguments that might be needed: "self" is the name of
;; the agent within the body code "t" is the current model time dt is
;; the requested dt the state variables are not directly available --
;; use "my" or "set-my!", "slot-ref" or "slot-set!"


;;(sclos-model-body <animal>
;;    (let ((fear (search environment 'predators))
;;          (feast (search environment 'prey))
```

```
;;          )
;;      (display (slot-ref 'name) "is running at " t)
;;      ...
;;      )
;;)


(define-macro (sclos-model-body myclass . body)
  ;; Call like (method classname (mname margs) . body) where the margs
  ;; doesn't include the call-next-method
  (if (not (symbol? myclass))
      (aborts "model-body: the class " myclass " should be an atom"))
  `(add-method
    run-model-body
    (make-method (list ,myclass)
                 (lambda (parent-body self t dt)
                   (let ((my (lambda (x) (slot-ref self x)))
                         (set-my! (lambda (x y) (slot-set! self x y)))
                         (kernel (slot-ref self 'kernel))
                         (skip-parent-body
                          (lambda ()
                            (if (not (slot-ref self 'agent-body-ran))
                                (begin
                                  (slot-set! self 'counter
                                             (1+ (slot-ref self 'counter)))
                                  ;;(set-my! 'subjective-time
                                  ;;    (+ (my'subjective-time) (my 'dt)))
                                  (slot-set! self 'agent-body-ran #t)))  )
                                    )
                                ,@body
                                )
```

```
                              )
                            )
                )
  )

;; like sclos-model-body, but chains straight to parent
(define-macro (sclos-use-parent-body myclass)
  `(sclos-model-body ,myclass (parent-body)))



(define-macro (sclos-setter myclass variable)
  (if (or (not (symbol? variable))
          (not (symbol? myclass)))
      (aborts "model-body: the class " myclass " and variable "
              variable " must be atoms"))
  `(add-method ,(string->symbol
                 (string-append "set-" (symbol->string variable) "!"))
               (make-method (list ,myclass)
                            (lambda (self arg)
                              (slot-set! self ',variable arg)))))

(define-macro (sclos-getter myclass variable)
  (if (or (not (symbol? variable)) (not (symbol? myclass)))
      (aborts "model-body: the class " myclass " and variable "
              variable " must be atoms"))
  `(add-method ,variable
               (make-method (list ,myclass)
                            (lambda (self)
                              (slot-ref self ',variable)))))
```

```
(include "framework.scm")

;(define-macro (migration-test .....))
;; returns an indication of how the agent is situated
;; w.r.t. representation

;(define-macro (migrate .....))
;; returns a new representation of the agent

;-  The End


;;; Local Variables:
;;; mode: scheme
;;; outline-regexp: ";-+"
;;; comment-column:0
;;; comment-start: ";;; "
;;; comment-end:""
;;; End:
```

A.3.2 framework.scm

```
; -*- mode: scheme; -*-
;-  Identification and Changes

;--
;  framework.scm -- Written by Randall Gray
;  Initial coding:
;     Date: 2016.07.26
;     Location: zero:/home/randall/Thesis/Example-Model/model/framework.scm
```

```
;
;  History:
;
;  This has the "defined" bits, not just the macros.

;-  Code


(if #t
    ;; If you are *not* running with the code

           ;;; ;;; (define load-level #f)
           ;;; ;;; (let* ((bl load)
           ;;; ;;;        (ll 0)
           ;;; ;;;        (nl (lambda x
           ;;; ;;;              (set! ll (+ ll 1))
           ;;; ;;;              (apply bl x)
           ;;; ;;;              (set! ll (- ll 1))))
           ;;; ;;;        (ll? (lambda () ll)))
           ;;; ;;;   (set! load nl)
           ;;; ;;;   (set! load-level ll?))

    ;; this block should not be run; it only exists to remind me
    ;; to (include ...) the framework file rather than (load ...) it.

    (if (not (zero? (load-level)))
        (error (string-append "You should be including the framework file,"
                              " rather than loading it.\n")))
)

;; These are not necessarily controls, but they are used in a similar
```

```
;; fashion and need to preceed framework-classes.

(define (class-name-of x)
  (cond
   ((isa? x <agent>)
    (let ((n (class-register 'name? (class-of x))))
      (and n
           ;;(string->symbol (string-append "instance-of-"
           ;;   (symbol->string n)))
           n
          )
      ))
   ((and (sclos-object? x) (assoc x (class-register)))
    (let ((n (class-register 'name? x)))
      (and n
           (string->symbol (string-append "class:" (symbol->string n)))
           )))
   (else #f)))


(define class-register
  (let ((register '())
        )
    (lambda args
      (if (null? args)
          register
          (let ((cmd (car args))
                (opts (if (null? (cdr args)) #f (cdr args))))
            (cond
             ((and (eq? cmd 'dump))
              (for-each dnl register)
```

```
 )

((and (eq? cmd 'add)
      opts
      (assq (car opts) register))
 (error "Attempting to re-register a class!" opts args)
 )

((and (eq? cmd 'add) opts (not (assq (car opts) register)))
 (set! register ;; save things as lists
       (acons (car opts) (cdr opts) register))
 )

((and (eq? cmd 'name?) opts)
 (let ((a (assq (car opts) register)))
   (and a (cadr a))))

((and (eq? cmd 'rec-by-class) opts)
 (let ((a (assq (car opts) register)))
   a))

((and (eq? cmd 'class?) opts)
 (let ((a (filter (lambda (x)
                    (eq? (car opts) (cadr x)))
                  register)))
   (and a (car a) (caar a))))

((and (eq? cmd 'rec-by-name) opts)
 (let ((a (filter (lambda (x)
                    (eq? (car opts) (cadr x)))
                  register)))
```

```
              (and a (car a))))

             (else (error "failed call to class-register"
                          args))))))))


(define generic-register
  (let ((register '())
        )
    (lambda args
      (if (null? args)
          register
          (let ((cmd (car args))
                (opts (if (null? (cdr args)) #f (cdr args))))
            (cond
             ((and (eq? cmd 'dump))
              (for-each dnl register)
              )

             ((and (eq? cmd 'add)
                   opts
                   (not (assq (car opts) register)))
              (set! register ;; save things as lists
                    (acons (car opts) (cdr opts) register))
              )

             ((and (eq? cmd 'name?) opts)
              (let ((a (assq (car opts) register)))
                (and a (cadr a))))

             ((and (eq? cmd 'rec-by-generic) opts)
```

```
             (let ((a (assq (car opts) register)))
               a))

            ((and (eq? cmd 'generic?) opts)
             (let ((a (filter (lambda (x)
                                (eq? (car opts) (cadr x)))
                              register)))
               (and a (car a) (caar a))))

            ((and (eq? cmd 'rec-by-name) opts)
             (let ((a (filter (lambda (x)
                                (eq? (car opts) (cadr x)))
                              register)))
               (and a (car a))))

            (else (error "failed call to generic-register"
                         args)))))))


(define (sclos-object? a)
  (and (%instance? a) #t))

(define (agent? a)
  (and (%instance? a) (isa? a <agent>) #t))

(define (has-slot? a k)
  (member k (map car (class-slots (class-of a)))))

(define (dnl* . args)
  (if (not (null? args))
      (let ((h (car args))
```

```
            (t (cdr args)))
        (display (car args))
        (for-each (lambda (x) (display " ") (display x)) t)))
  (newline))


(define (dnl . args)
  (if (not (null? args))
      (let ((h (car args))
            (t (cdr args)))
        (display (car args))
        (for-each (lambda (x) (display x)) t)))
  (newline))


(define files-included-immediately
  '("support-lib.scm"
    "framework-controls.scm"
    "framework-classes.scm"
    "framework-declarations.scm"
    "framework-methods.scm"))

(define framework-load
  (let* ((oload load)
         (always-print-loading #f)
         (always-print-loaded #f)
         (always-load #f)
         (warn-loaded #f)
         (loaded '())
         (loading '())
         (prload (lambda (x y)
```

```
                (display (string-append "\n[" x " "))
                (display y)(display "]\n")))
    )
(lambda (fn . args)

  (if (not (null? fn))
      (if (member fn loaded)
          (if warn-loaded
              (dnl "[" fn " is already loaded"))
          (cond
           ((member fn (list "loaded?"))
            (for-each dnl (reverse loaded))
            )
           ((and (string? fn) (not (null? args)))
            (dnl "forced loading of " fn)
            (oload fn) ;; forces a load, doesn't record it
            )
           ((eq? fn 'warn-loaded)
            (set! warn-loaded #t))

           ((eq? fn '!warn-loaded)
            (set! warn-loaded #f))

           ((eq? fn 'always-load)
            (set! always-load #t))

           ((eq? fn '!always-load)
            (set! always-load #f))

           ((eq? fn '!print-loaded)
            (set! always-print-loaded #f))
```

```
            ((eq? fn '!print-loading!)
             (set! always-print-loading #f))

            ((eq? fn 'print-loaded)
             (set! always-print-loaded #t))

            ((eq? fn 'print-loading!)
             (set! always-print-loading #t))

            ((member fn '(loaded loaded-list loaded? loaded-list?))
             (prload "loaded" loaded))

            ((member fn '(loading loading-list
                                  loading? loading-list? load-list?))
             (prload "loading" loading))

            (#t
             (set! loading (cons fn loading))
             (if always-print-loading (prload "loading" loading))
             (if (or always-load (not (member fn loaded)))
                 (begin
                   (oload fn)
                   (set! loading (cdr loading))
                   (set! loaded (cons fn loaded))
                   (if always-print-loaded (prload "loaded" loaded))))
             ))
           )
         )))
)
```

```
(define load framework-load)

(define (aborts . args) (error (apply string-append args)))

(define (load-model . extra-files)
  (if (and (pair? extra-files) (pair? (car extra-files)))
      (set! extra-files (car extra-files)))

  (for-each load files-included-immediately)

  ;; submodel code -- all the classes get loaded here
  (load "submodels.scm")

  ;; The kernel -- last because it *feels* right.
  (load "kernel.scm")
  (load "model-configuration.scm")
  (if (pair? extra-files)
      (for-each load extra-files))
  )


(display "Now run: (load-model) or (load-model filename ...)\n")

;-  The End


;;; Local Variables:
;;; comment-end: "-;" -;
;;; comment-start: ";;; " -;
;;; mode: scheme -;
```

```
;;; outline-regexp: ";-+" -;
;;; comment-column: 0 -;
;;; End:


"This serves as a template for other population agents, as well as the
population machinery for the dynamic patch class."



(load "utils.scm")
(load "units.scm")
(load "sort.scm")
(load "maths.scm")
(load "maths.scm")
(load "integrate.scm")
(load "units.scm")
(load "postscript.scm")

(load "integrate.scm")
(load "matrix.scm")

;;=============================================================================;;
;;
;;                        Constants and such
;;
;;=============================================================================;;
```

```
(define unbounded +inf.0)

;;=========================================================================;;
;;
;;                        TERMS FOR THE dP/dt expression
;;
;;=========================================================================;;


;; Multiplicative cap on the growth of the species. k = "unbounded" makes it go away
;;(define (logistic-- a k) (if (eq? k unbounded) 1.0 (- 1 (pow (/ a k) 2.4))))

(define (logistic-- a k) (if (eq? k unbounded) 1.0 (- 1 (/ a k))))

(define (logistic-growth-- a k) (if (> a k) 0.0 (- 1.0 (/ a k))))
(define (logistic-mort-- a k) (if (< a k) 0.0 (- (/ a k) 1.0)))

;; sums the contributors to population growth
(define growth-terms-- +)

;; mortality-terms should be included as a modifier within a growth-terms s-exp
(define (mortality-terms-- . args) (- 0 (apply + args)))


;;=========================================================================;;


"
The usual pattern for a std-d/dt would be

(make-basic-population 'grass '(grass rabbit fox) ...)
```

```
"


(define (make-basic-population name population-names . rest)
  (let* ((name name)
         (self #f)
         (nrest (length rest))
         (popnames population-names)
         (populations #f)
         (growth (if (>= nrest 1) 0 (list-ref rest 0)))
         (nmort (if (>= nrest 2) 0 (list-ref rest 1)))
         (cap (if (>= nrest 3) unbounded (list-ref rest 2)))
         (val (if (>= nrest 4) 0 (list-ref rest 3)))
         (logistic #f)
         (logistic-growth #f)
         (logistic-mort #f)
         (prey-rates #f) ;; attack-rate and efficiency (in that order)
         (pred-att-rate #f) ;; predator's attack-rate is obtained from the predator
         (helper-rate #f)
         (competitor-rate #f)

         (fascist-init #f)
         )
    (let* ((preys-on (lambda (vals att-rate eff)
                       (apply + (map * vals att-rate eff))))

           (preyed-on-by (lambda (vals att-rate)
                           (apply + (map * vals att-rate))))

           ;; These are like the predation terms, but without any symmetry
```

```
(helped-by (lambda (vals helper-rate)
             (apply + (map * vals helper-rate))))

(competes-with (lambda (vals competitor-rate)
                 (apply + (map * vals competitor-rate))))


(d/dt
 (if (zero? nrest)

     (lambda args (abort
                   (string-append "d/dt is uninitialised:"
                                  " use the 'set-d/dt! call to set it")))
     (lambda (t . pvals)
       ;; t is the first element in the list, other
       ;; elements must correspond to the populations
       ;; indicated in the "populations" list when the
       ;; population is created
       ;;
       ;; vals ought to be in order
       (let ((val (list-ref pvals (- (length popnames)
                                     (length (member name popnames))))))
         (if #f
             (begin
               (dnl name "-----------------------")
               (dnl "value ====> " val)
               (dnl "growth = " growth)
               (dnl "preys-on = " (preys-on pvals (map car prey-rates)
                                            (map cadr prey-rates)))
               (dnl "total mortality =  "
                    (+ nmort (if pred-att-rate
```

```
                    (preyed-on-by pvals
                                  (map car pred-att-rate))
                    0)))
      (dnl "natural mortality = " nmort)
      (dnl "preyed-on-by = "
           (preyed-on-by pvals
                         (if pred-att-rate
                             (map car pred-att-rate)
                             0)))

      (if logistic
          (dnl "logistic " logistic "*" (logistic-- val cap)))
      (if logistic-growth
          (dnl "logistic-growth "
               logistic-growth "*"
               (logistic-growth-- val cap)))
      (if logistic-mort
          (dnl "logistic-mort " logistic-mort "*"
               (logistic-mort-- val cap)))

      (dnl)))


(* val
   (if logistic (* logistic (logistic-- val cap)) 1.0)
   (growth-terms-- growth
                   (if prey-rates
                       (preys-on pvals (map car prey-rates)
                                 (map cadr prey-rates))
                       0)
```

```
                                (if helper-rate
                                    (helped-by pvals helper-rate)
                                    0)

                                (mortality-terms--
                                 nmort
                                 (if pred-att-rate
                                     (preyed-on-by pvals
                                                   (map car pred-att-rate))
                                     0)
                                 (if competitor-rate
                                     (competes-with pvals competitor-rate)
                                     0)
                                 )))
         ))
     )
 )
(accessor
 (letrec ((population
           (lambda args
             (cond
              ((null? args)
               val)

              ((eq? (car args) 'dump)
               (pp (list
                    'name name
                    'self self
                    'pops popnames populations
                    'params val cap growth nmort
                    'prey-rates prey-rates
```

```
      'pred-att-rate pred-att-rate
      'logistic/-growth/-mort
        logistic logistic-growth logistic-mort
      'd/dt d/dt
      'func population)))

((eq? (car args) 'set!)
 (set! val  (cadr args)))

((eq? (car args) 'register-populations)
 ;; expects (animal 'register-populations
 ;;    (list plant animal toothy-animal....))
 (if (not (apply andf (map procedure? (cadr args))))
     (Abort))
 (if (null? (cdr args))
     (Abort "No populations?  What's the point?")
     (set! populations (copy-list (cadr args)))
     ))

((eq? (car args) 'register-prey)
 (if (not populations)
     (Abort "You must register the populations before registering the prey")
     (begin
       (set! prey-rates (make-list (length populations) '(0 0)))
       (if (not (null? (cdr args)))
           (begin
             (for-each
              (lambda (x y)
                (list2-assoc-set! populations prey-rates x y))
              (map car (cdr args)) (map cdr (cdr args)))
             )
```

```
            )
          )
        ))

((eq? (car args) 'register-predators)
 (if (not prey-rates)
     (abort (string-append
             (symbol->string name)
             " has been called before both the prey list has been
                registered")))
 (set! pred-att-rate (map (lambda (y) (y 'attack-rate self)) populations))
 (if (zero? (apply + (map abs (map car pred-att-rate))))
     (set! pred-att-rate #f))
 ;;(dnl name pred-att-rate)
 )

((eq? (car args) 'logistic)
 (if (not (or (null? (cdr args)) (boolean? (cadr args)) (number? (cadr
    args))))
     (abort "argument to (entity 'logistic) must be null, #f #t or a
        number"))
 (if (null? (cdr args))
     (set! logistic 1.0)
     (set! logistic (cadr args)))

 (if logistic
     (begin
       (set! logistic-growth #f)
       (set! logistic-mort #f)))

 )
```

```
((eq? (car args) 'register-helpers)
 (if (not populations)
     (Abort "You must register the populations before registering
        interactions")
     (begin
       (set! helper-rate (make-list (length populations) 0))
       (if (not (null? (cdr args)))
           (begin
             (for-each
              (lambda (x y)
                (list2-assoc-set! populations helper-rate x y))
              (map car (cdr args)) (map cadr (cdr args)))
             )
           )
       )
     ))

((eq? (car args) 'register-competitors)
 (if (not populations)
     (Abort "You must register the populations before registering
        interactions")
     (begin
       (set! competitor-rate
             (make-list (length populations) 0))
       (if (not (null? (cdr args)))
           (begin
             (for-each
              (lambda (x y)
                (list2-assoc-set! populations competitor-rate x y))
              (map car (cdr args)) (map cadr (cdr args)))
```

```
            )
          )
        )
      ))

((eq? (car args) 'logistic-growth)
 (if (not (or (null? (cdr args))
              (boolean? (cadr args))
              (number? (cadr args))))
     (abort "argument to (entity 'logistic-growth) must be null, #f #t or a
        number"))
 (if (null? (cdr args))
     (set! logistic-growth 1.0)
     (set! logistic-growth (cadr args)))
 (if logistic-growth (set! logistic #f))

 )

((eq? (car args) 'logistic-mort)
 (if (not (or (null? (cdr args))
              (boolean? (cadr args))
              (number? (cadr args))))
     (abort
      "argument to (entity 'logistic-mort) must be null, #f #t or a
         number"))
 (if (null? (cdr args))
     (set! logistic-mort 1.0)
     (set! logistic-mort (cadr args)))
 (if logistic-mort (set! logistic #f))
 )
```

```
((and (eq? (car args) 'attack-rate) (procedure? (cadr args)))
 (if (not prey-rates)
     (if fascist-init
         (abort
          (string-append
           (symbol->string name)
           " has been called before both the prey list has been
              registered"))
         0)
     (let ((p (list2-assoc populations prey-rates (cadr args))))
       (if p (car p) 0))))

((eq? (car args) 'update)
 (let ((dP (apply self (cdr args))))
   (set! value (+ value (* dP (cadr args))))
   ))

((eq? (car args) 'set-d/dt!)
 (set! d/dt (cadr args)))

((eq? (car args) 'd/dt)
 d/dt)

((number? (car args))
 (apply d/dt args))

(else (abort
       (string-append "The argument to "
                      (symbol->string name) ", "
                      (object->string (car args))
                      ", is not recognised")))
```

```
                        ) ;; cond
                       ) ;; lambda
                     ) ;; population defn
                   ) ;; letrec closure
              population) ;; letrec
            ) ;; accessor definition
          ) ;; let* closure
      (set! self accessor) ;; so we can identify ourselves to others
      accessor) ;; return accessor function for the population
    ) ;; outer let*
  )


;-  The End


;;; Local Variables:
;;; mode: scheme
;;; outline-regexp: ";-+"
;;; comment-column:0
;;; comment-start: ";;; "
;;; comment-end:""
;;; End:
```

A.3.3 Framework controls

```
;-  Identification and Changes

;--
;  framework-controls.scm -- Written by Randall Gray
```

```
;-  Discussion

;-  Configuration stuff

;-  Included files

;-  Variables/constants both public and static

;--    Static data


;; This is a list of symbols (or strings, I guess) -- kdnl* calls
;; which have a member of this list, get printed during the run.

(define kernel-messages '())
(define adjust-grey #t)
(define nested-agents '()) ;; needs this for the moment



;; If this is set to true, the kernel becomes inaccessible to the
;; agent when it is not running. Thus, it is unable to access
;; non-local information when answering queries and performing updates

(define blue-meanie #f)

(define logger-tags '())
(define submodel-register '())
(define (register-submodel tag . filelist)
  (if (assoc tag submodel-register)
      (set-cdr! (assoc tag submodel-register)
```

```
                (append (assoc tag submodel-register) filelist))
      (set! submodel-register
            (cons (cons tag filelist) submodel-register)))
  )



;-  The End


;;; Local Variables:
;;; mode: scheme
;;; outline-regexp: ";-+"
;;; comment-column:0
;;; comment-start: ";;; "
;;; comment-end:""
;;; End:
```

A.3.4 Framework classes

```
;- Identification and Changes

;- Load initial libraries

;; sclos.scm *must* be loaded before this file


;;-------------------------------------------;;
;; This is the order things must happen in   ;;
;; any file defining methods or model bodies ;;
```

```
;;-------------------------------------------
;; (include "framework-preamble%.scm")
;; (load-model-framework)
;;-------------------------------------------


;-- Define/allocate new classes

;--- substrate

;; we will add a "list" primitive ... different from "pair"
;(define <pair>        (make-primitive-class))
(define <list>        (make-primitive-class))
(define <integer>     (make-primitive-class))
(define <rational>    (make-primitive-class))
(define <real>        (make-primitive-class))
(define <complex>     (make-primitive-class))

(define general-class-of
  (let* ((primitive-class-of class-of)
         (co (lambda (x)
               (cond ;; these are carefully ordered!
                ((list? x)        <list>)
                ((integer? x)     <integer>)
                ((rational? x)    <rational>)
                ((real? x)        <real>)
                ((complex? x)     <complex>)
                (#t (primitive-class-of x))))))
    (set! class-of co)))
```

```
;(define <null>         (make-primitive-class))
;(define <symbol>       (make-primitive-class))
;(define <boolean>      (make-primitive-class))
;(define <procedure>    (make-primitive-class <procedure-class>))
;(define <number>       (make-primitive-class))
;(define <vector>       (make-primitive-class))
;(define <char>         (make-primitive-class))
;(define <string>       (make-primitive-class))
;(define  <input-port>  (make-primitive-class))
;(define <output-port>  (make-primitive-class))


;--- helpers/warts



;; sclos classes
(register-class <pair>)
(register-class <null>)
(register-class <boolean>)
(register-class <integer>)
(register-class <rational>)
(register-class <real>)
(register-class <complex>)
(register-class <symbol>)
(register-class <procedure>)
(register-class <number>)
(register-class <vector>)
(register-class <char>)
```

```
(register-class <string>)
(register-class <input-port>)
(register-class <output-port>)
(register-class <class>)
(register-class <top>)
(register-class <object>)
(register-class <procedure-class>)
(register-class <entity-class>)
(register-class <generic>)
(register-class <method>)


;--- agent based classes


(define <agent>
  (make-class (inherits-from <object>)
              (state-variables name type representation agent-state
                               note
                               kernel
                               subjective-time priority jiggle
                               dt
                               schedule
                               migration-test timestep-schedule counter
                               map-projection
                               state-flags
                               agent-epsilon
                               agent-schedule
                               dont-log
                               agent-body-ran
```

```
                              ;; as a parent of other agents
                              subsidiary-agents active-subsidiary-agents
                              parent-nesting-state

                              ;; as a child of other agents
                              nest-parent child-nesting-state
                              )
               ))

;; subsidiary-agents are agents which may be embedded in a larger
;; dynamic agent. Agents know what their parent agent is (if they have
;; one) and may indicate to the parent that they should be added to
;; the active list. The parent agent is the one that actually decides
;; if a agent is to move into the active queue or out of the active
;; queue.  Whe things get moved, "value" from the parent is moved into
;; the relevant sub-agents.  The set of ecoservices of the parent
;; contains all of the types represented in its sub-agents.
;;
;; priority is an integer, the higher the integer the greater the
;; priority.  The default value is zero jiggle is a real number in (0,
;; 1).  Setting jiggle to values outside that domain suppresses the
;; the randomisation of the jiggle.  If an agent has a schedule, then
;; the schedule WILL be used in determining the next dt.



(register-class <agent>)

;; name is a string
;; representation is a symbol
```

```
;; subjective-time, and dt are numbers representing a time and an interval,
;; body is a function (lambda (self t df . args) ...)
;; migration-test is a function (lambda (self t df . args) ...)
;; timestep-schedule is a list of times at which the agent needs to run
;;    (a list of monotonically increasing numbers)
;; kernel is a function that can be used to interact with
;;    the kernel of the simulation


(load "log-classes.scm") ;; These are used to generate output.


(define <tracked-agent>
  (make-class (inherits-from
               <agent>) (state-variables track tracked-paths track-schedule
                                         track-epsilon)))
;; "track" will either be a list like (... (t_k x_k y_k) ...) or false
;; "tracked-paths" is a list of non-false traces or false

(register-class <tracked-agent>)

(define <thing>
  (make-class (inherits-from <tracked-agent>)
              (state-variables mass dim location direction speed)))
(register-class <thing>)
;;(class-register 'add <thing> "<thing>")

(define <environment>
  (make-class (inherits-from <agent>)
              (state-variables default-value minv maxv))) ;; bounding volume
```

```
(register-class <environment>)

;;; Local Variables:
;;; mode: scheme
;;; outline-regexp: ";-+"
;;; comment-column:0
;;; comment-start: ";;; "
;;; comment-end:""
;;; End:
```

A.3.5 Framework declararations

```
;-  Identification and Changes

;--
;  framework-declarations.scm -- Written by Randall Gray


;--- basic wrangling -- ALL classes
(declare-method
 initialise ;; *** Note spelling ***
 "handles the initialisation of state variables")
(declare-method
 dump
 "dumps a 'readable' view of the state of an object")

(declare-method
 agent-prep
 "runs any 'prep' code, like opening files")
(declare-method
 agent-shutdown
```

```
 "runs any 'shutdown' code, like closing files")

(declare-method
 set-state-flag!
 "set a flag in the ad hoc state-variable list")
(declare-method
 add-state-flag
 "add a flag to the ad hoc state-variable list")
(declare-method
 state-flag
 "return the value of the flag in the ad hoc state-variable list")

(declare-method
 insert-agent!
 "add an agent in the introspection list")
(declare-method
 append-agent!
 "add an agent in the introspection list")

;--- Misc.

(declare-method total-value "description needed")

;--- state variables
(declare-method
 name
 "A unique identifier for the agent")
(declare-method
 set-name!
 "Set the unique name")
(declare-method
```

```
 type
 "the categorical type of agent (analogous to species or genus)")
(declare-method
 set-type!
 "set/change the type of an agent")
(declare-method
 representation
 "individual, aggregate, analytic, circle, polygon...")
(declare-method
 set-representation!
 "record the agent's representation")
(declare-method
 comment
 "reference information for agent")
(declare-method
 set-comment!
 "reference information for agent")
(declare-method
 subjective-time
 "the agent's current time")
(declare-method
 set-subjective-time!
 "adjust the agents view of 'now'")
(declare-method
 priority
 "run priority")
(declare-method
 set-priority!
 "change the run priority")
(declare-method
 jiggle
```

```
 "the current random adjustment when being inserted in the runqueue")
(declare-method
 set-jiggle!
 "reset the jiggle")
(declare-method
 dt
 "the amount of time the agent is running for or will run")
(declare-method
 set-dt!
 "change the dt")
(declare-method
 migration-test
 (string-append "a function which indicates if an entity"
                " should change representation: "
                "(f self t ldt return)"))
(declare-method
 set-migration-test!
 "set/change the migration test function")
(declare-method
 timestep-schedule
 "a list of scheduled timesteps")
(declare-method
 set-timestep-schedule!
 "change the timestep list")
(declare-method
 kernel
 "function which handles kernel queries")
(declare-method
 set-kernel!
 "change the kernel query function")
```

```
;--- supporting the modelling framework
(declare-method
 snapshot
 "used by logging mechanism")
(declare-method
 i-am
 "returns the representation of the entity")
(declare-method
 is-a
 "representation predicate")
(declare-method
 parameter-names
 "names of all parameters (slots) of the agent's class")
(declare-method
 parameters
 "values of the parameters (slots) of the agent;s class")
(declare-method
 set-parameters!
 (string-append "set parameters using a list of the form "
                "'(pval...) where vals are in the right order"))
(declare-method
 query
 "Send a query to the kernel")
(declare-method
 run-at
 "schedule a tick at a particular time")
(declare-method
 run
 "run an agent")
(declare-method
 run-model-body
```

```
 "run an agent's model-body")
(declare-method
 run-nested-model-body
 "run a nested model body")
(declare-method
 run-agents
 "This routine runs a list of agents using the passed 'run' procedure")
(declare-method
 run-nested-agents
 "This routine hands a list of nested agents to run-agents")


;;(declare-method model "description needed")
;;(declare-method set-model! "description needed")

(declare-method
 extra-variable
 "refer used in the logging to get non-slot data out of a class")
(declare-method
 extra-variable-list
 "refer used in the logging to get non-slot data out of a class")

(declare-method migrate "description needed")

;--- <thing>
(declare-method
 mass
 "Return the mass of an entity")
(declare-method
 set-mass!
 "Set the mass of an entity")
```

```
(declare-method
 dim
 "Return the number of dimension an entity works in")
(declare-method
 set-dim!
 "Set the dimensionality of an entity -- (x y z) would be 3'")
(declare-method
 location
 "Return the location of an entity")
(declare-method
 set-location!
 "Set the location of an entity")
(declare-method
 direction
 "Return the direction of an entity")
(declare-method
 set-direction!
 "Set the direction of an entity")
(declare-method
 speed
 "Return the speed of an entity")
(declare-method
 set-speed!
 "Set the speed of an entity")

(declare-method
 track-locus!
 "Insert the time and location in the track list")
(declare-method
 track
 "return the entity's track")
```

```
(declare-method
 set-track!
 "set the track to the supplied 'track' list")
(declare-method
 new-track!
 "indicates that a new segment of tracking is to start")
(declare-method
 tracks
 "returns a list of track segments (or false)")

;--- <environment> generics
(declare-method
 contains?
 "predicate which indicates containment")
(declare-method
 value
 "typically the value of an ecoservice or patch")
(declare-method
 set-value!
 "set a value -- in the case of the environment, its default value")
(declare-method
 min-bound
 "list of minima in the dimensions")
(declare-method
 max-bound
 "list of maxima for the dimensions")


(load "log-declarations.scm")

;-  The End
```

```
;;; Local Variables:
;;; mode: scheme
;;; outline-regexp: ";-+"
;;; comment-column:0
;;; comment-start: ";;; "
;;; comment-end:""
;;; End:
```

A.3.6 Framework methods

```
;- Identification and Changes

;- Load initial libraries

;; these must be loaded before this is
;;(include
;;(load "maths.scm")
;;(load "integrate.scm")


;;-------------------------------------------;;
;; This is the order things must happen in   ;;
;; any file defining methods or model bodies ;;
;;-------------------------------------------
;; (include "%framework.scm")
;; (load-model-framework)
;;-------------------------------------------
```

```
;;-------------------------------------------------
;; Important routines which I Really Ought to Know ;;
;;-------------------------------------------------



;- Utility functions

(define show-warnings #f)
(define kernel-messages '())
(define temporal-fascist #f) ;; This can make things quite picky.

(define (symbol<? s t)
  (if (not (and (symbol? s) (symbol? t)))
      (error "Passed a non symbol to symbol<?" s t)
      (string<? (symbol->string s) (symbol->string t))))

(define set-kernel-messages! #f)
(define add-kernel-message! #f)
(define remove-kernel-message! #f)
(define kdnl* #f)

(let* ((kernel-messages '())
       (set (lambda (msg)
              (if (not (list? lst))
                  (error "set-kernel-messages! requires a list of symbols" lst))
              (set! kernel-messages (copy-list lst))))
       (add (lambda (msg)
              (if (not (symbol? msg))
                  (error "add-kernel-message! requires a symbol" msg))
```

```
            (set! kernel-messages (uniq (sort (cons msg kernel-messages)) symbol<?))))
      (remove (lambda ()
                (if (not (symbol? msg))
                    (error "remove-kernel-messages! requires a symbol" msg))
                (set! kernel-messages (filter (lambda (x) (not (eq? msg x))) kernel-messages))))
      (kdnl (lambda (msg . args)
              (if (or (member msg kernel-messages) (member '* kernel-messages)
                      (and (list? msg)
                           (not (null? (intersection msg kernel-messages)))))
                  (begin
                    (display msg)(display " ==> ")
                    (apply dnl* args)))))

      )
  (set! set-kernel-messages! set)
  (set! add-kernel-message! add)
  (set! remove-kernel-message! remove)
  (set! kdnl* kdnl)
)

(define (warning . args)
  (if show-warnings
      (begin
        (display "*** warning: ")
        (apply dnl* args)
        )))

(define FirstJiggle 1.0)
(define LastJiggle 0.0)
(define DefaultPriority 0)
```

```
(load "log-methods.scm")



;-- Define/allocate new classes

;--- substrate

;--- helpers/warts

;--- agent based classes

;-- handy macros that need class definitions

;-- Define/allocate new generic methods

;; This is defined all in one place so we don't get rogue
;; redefinitions of a generic method clobbering the definitions which
;; occur earlier in the file. I don't know if tiny-clos can trap such
;; things, but it seems simple enough to split things so that it
;; doesn't happen, though I prefer to keep a class all in one place.
;; Ditto for the "make-class" calls.


;--- Helper classes (wart classes)
; none yet

;--- Agent classes
```

```
;---- <agent>

;--- Helper classes (wart classes)

;----- (initialise) ;; fundamental component in the init routine

(object-method <object> (initialise self args)
               ;; args should be null or a list of the form ('tag value ...),
               ;; slotlist is a list of valid slotnames
              (if (and (pair? args) (pair? (car args)) (= (length args) 1))
                  (set! args (car args)))
              (let ((slots  (map car (class-slots (class-of self)))))
                (for-each
                 (lambda (slotname argval)
                   (if (member slotname slots)
                       (set-my! slotname argval)
                       (begin
                         (display
                          (string-append
                           "Use of undeclared class variable: "
                           (if (string? slotname)
                               slotname
                               (object->string slotname))
                           " in " (symbol->string (class-name-of self)) "\n"))
                         (error "+++Redo from Start+++" '--Hex:TP!=NTP))
                       )
                   )
                 (evens args) (odds args)))
              )
```

```
(sclos-method <class> (run self pt pstop pkernel)
              (if (not (isa? self <agent>))
                  (begin (display "Attempt to (run ...) a non-agent\n")
                         (error "+++Curcurbit Error+++"
                                (slot-ref self 'name)))))



;--- Agent classes
;---- <agent> methods


;----- (initialize)

(sclos-method <agent> (initialize self args)
              (initialise
               self (list 'state-flags '()
                          'subjective-time 0.0
                          'dt 1.0
                          'jiggle LastJiggle
                          'priority DefaultPriority
                          'migration-test uninitialised
                          'counter 0 'map-projection (lambda (x) x)
                          'agent-schedule '() 'agent-epsilon 1e-6
                          'agent-state 'ready-for-prep
                          'agent-body-ran #f

                          'dont-log '(ready-for-prep
                                      ;; agent things
                                      agent-body-ran agent-schedule
```

```
 agent-epsilon map-projection counter
 migration-test state-flags
 dont-log timestep-schedule kernel

 ;; log agent things
 introspection-list introspection-schedule
 timestep-epsilon

 dims ;; thing things

 ;; environment things
 default-value minv maxv

 ;; ecoservice things
 plateau-interval growth-rate

 ;; landscape things
 service-list service-update-map
 update-equations terrain-function
 dump-times scale
 log-services-from-patch
 log-patches-from-habitat

 ;; animal things
 domain-attraction food-attraction
 near-food-attraction searchspeed
 wanderspeed foragespeed
 movementspeed foodlist homelist
 breedlist habitat
 )
))
```

```
              (initialize-parent)
              ;; call "parents" last to make the initialisation list work
              (initialise self args))


(sclos-method <agent> (agent-prep self . args)
              (slot-set! self 'timestep-schedule
                         (unique (sort (slot-ref self 'timestep-schedule) <)))
              ;; ensures no duplicate entries
              (if (eq? (slot-ref self 'agent-state) 'ready-for-prep)
                  (slot-set! self 'agent-state 'ready-to-run)
                  (error (string-append
                          (name self)
                          " has been instructed to prep but it's state is "
                          (my 'agent-state)))
))


;; Termination can happen from any state
(sclos-method <agent> (agent-shutdown self . args)
              (slot-set! self 'agent-state 'terminated))


;----- (dump) ;; This dumps all the slots from agent up.

(sclos-method <agent> (dump self . count)
              (set! count (if (null? count) 0 (car count)))
              (let* ((slots (map car (class-slots (class-of self))))
                     (vals  (map (lambda (x) (slot-ref self x)) slots)))
                (for-each (lambda (x y)
                            (display (make-string count #\space))
```

```
                          (display x) (display ": ")
                          (display y)(newline))
                        slots vals)))

;; This is the default log method in the absence of more specific code.

(sclos-method (<agent>) (log-data self logger format caller targets)
              (kdnl* '(log-* log-data)
                     (name self)
                     "[" (my 'name) ":"
                     (class-name-of self) "]" "in <agent>:log-data")
              (let ((file (slot-ref logger 'file))
                    (show-field-name (slot-ref logger 'show-field-name))
                    (missing-val (slot-ref logger 'missing-val))
                    (spaced-out #f)
                    (cntr 0)
                    )

                (for-each ;; field in the variable list
                 (lambda (field)
                   (if show-field-name
                       (begin
                         (if (not spaced-out)
                             (set! spaced-out #t)
                             (display " " file))
                         (display field file)))

                   (cond
                    ((has-slot? self field)
                     (kdnl* '(log-* log-data logging-debug)
                            "  " (name self) (class-name-of self)
```

```
          "Dumping " field "=" (if (has-slot? self field)
                                   (slot-ref self field)
                                   "missing!"))
    (if (not spaced-out)
        (set! spaced-out #t)
        (display " " file))
    (display (slot-ref self field) file)
    )
   ((member field (extra-variable-list self))
    (kdnl* '(log-* log-data logging-debug)
           "  " (name self) (class-name-of self)
           "Dumping extra " field "="
           (extra-variable self field))
    (if (not spaced-out)
        (set! spaced-out #t)
        (display " " file))
    (display (extra-variable self field) file)
    )
   (missing-val
    (if (not spaced-out)
        (set! spaced-out #t)
        (display " " file))
    (display missing-val file))
   (else
    (if (not spaced-out)
        (set! spaced-out #t)
        (display " " file))
    )
   )
  )
(unique (if #t targets
```

```
                            (filter (not-member (my 'dont-log)) targets))))
                (newline file)
                )
              )

(sclos-method (<agent>) (run-agents self t dt agentlist run)
                      (for-each (lambda (x)
                                  (if (< (subjective-time x) (+ t dt))
                                          (run x t (+ t dt) (my 'kernel))
                                          (dnl* "Skipping" (name x)))
                                  )
                                agentlist
                                )
                      )

(sclos-method (<agent>) (run-nested-agents self t dt run)
              (let ((al (my 'subsidiary-agents)))
                (if (not (null? al))
                    (run-agents self t dt al run))
                ))



;----- (name)

(sclos-method <agent> (name self)
              (if (not (or (string? (my 'name)) (eq? (my 'name) #f)))
                  (error "agent:name -- not a string")
                  (my 'name)))

;(sclos-method <agent> (name self)
```

```
;             (my 'name))

;; (add-method name
;;              (make-method (list <agent>)
;;                           (lambda (name-parent self)
;;                             (slot-ref self 'name))))

;----- (set-name!)

(sclos-method (<agent> <string>) (set-name! self n)
              (if (string? n)
                  (set-my! self 'name n)
                  (error "agent:set-name! -- arg is not a string")))



(define undefined (lambda x 'undefined))
(define undefined-state-flag (lambda x 'undefined-state-flag))

(sclos-method <agent> (type self)
              (my 'type))

;;

(sclos-method <agent> (set-type! self newtype)
              (set-my! 'type newtype))

;;

(sclos-method (<agent> <symbol>) (set-state-flag! self sym val)
              (let ((v (assoc sym (my 'state-flags))))
```

```
                (if v
                  (set-cdr! v val)
                  (set-my! 'state-flags
                           (cons (cons sym val)  (my 'state-flags)))
                  )
                ))

(sclos-method (<agent> <symbol>) (add-state-flag self sym val)
              (if (assoc sym (my 'state-flags))
                  (set-state-flag! self sym val)
                  (set-my! 'state-flags
                           (cons (cons sym val)  (my 'state-flags)))
                  ))


(sclos-method (<agent> <symbol>) (state-flag self sym)
              (let ((r (assoc sym (my 'state-flags))))
              (if r
                  (cdr r)
                  undefined-state-flag)))


;----- (representation)
(sclos-method <agent> (representation self)
              (let ((rep (my 'representation)))
                (if (symbol? rep)
                    rep
                    (error "agent:representation --  not a symbol"))))
```

```
;----- (set-representation!)
(sclos-method (<agent> <string>) (set-representation! self n)
              (if (symbol? n)
                  (set-my! self 'representation n)
                  (error "agent:set-representation! -- arg is not a symbol"))
              )


;----- (subjective-time)
(sclos-method <agent> (subjective-time self)
              (my 'subjective-time))


;----- (set-subjective-time!)
(sclos-method (<agent>) (set-subjective-time! self n)
              (if (number? n)
                  (slot-set! self 'subjective-time n)
                  (error "agent:set-subjective-time! -- arg is not a number")))


;----- (priority)
(sclos-method (<agent>) (priority self)
              (my 'priority))


;----- (set-priority!)
(sclos-method (<agent> <number>) (set-priority! self n)
              (if (number? n)
                  (slot-set! self 'priority n)
                  (error "agent:set-priority! -- arg is not a number")))
```

```
;----- (jiggle)
(sclos-method <agent> (jiggle self)
              (my 'jiggle))

;----- (set-jiggle!)
(sclos-method (<agent> <number>) (set-jiggle! self n)
              (if (number? n)
                  (slot-set! self 'jiggle n)
                  (error "agent:set-jiggle! -- arg is not a number")))

;----- (migration-test)
(add-method migration-test
            (make-method (list <agent>)
                         (lambda (migration-test self)
                           (slot-ref self 'migration-test))))

;----- (set-migration-test!)
(add-method set-migration-test!
            (make-method (list <agent> <number>)
                         (lambda (set-migration-test!-parent self ntest)
                           (if (procedure? ntest)
                               (slot-set! self 'migration-test ntest)
                               (error (string-append
                                       "agent:set-migration-test! -- "
                                       "arg is not a procedure")))
                           )))

;----- (timestep-schedule)
(add-method timestep-schedule
            (make-method (list <agent>)
```

```
                        (lambda (timestep-schedule self)
                          (slot-ref self 'timestep-schedule))))


;----- (set-timestep-schedule!)
(add-method set-timestep-schedule!
            (make-method (list <agent> <number>)
                         (lambda (set-timestep-schedule!-parent self nbody)
                           (if (list? nbody)
                               (slot-set! self 'timestep-schedule nbody)
                               (error (string-append
                                       "agent:set-timestep-schedule! -- "
                                       "arg is not a procedure")))
                           (slot-set! self 'timestep-schedule nbody)
                           )))

;----- (kernel)
(add-method kernel
            (make-method (list <agent>)
                         (lambda (kernel self)
                           (slot-ref self 'kernel))))

;----- (set-kernel!)
(add-method set-kernel!
            (make-method (list <agent> <number>)
                         (lambda (set-kernel! self n)
                           (if (number? n)
                               (slot-set! self 'kernel n)
                               (error (string-append
                                       "agent:set-kernel! -- "
                                       "arg is not a number"))
```

```
                        )))
            )


(sclos-method (<agent>) (snapshot self)
              (map (lambda (x) (list x (slot-ref self x)))
                   (class-slots (class-of self))))


(sclos-method <agent> (i-am self) (my 'representation))

(sclos-method (<agent>) (is-a self list-of-kinds)
              (member (my 'representation) list-of-kinds))

(sclos-method <agent> (parameter-names self)
              (map car (class-slots (class-of self))))
(sclos-method <agent> (parameters self)
              (map (lambda (x) (slot-ref self x))
                   (map car (class-slots (class-of self)))))

(sclos-method (<agent> <pair>) (set-parameters! self newparams)
              (for-each (lambda (x y) (slot-set! self x y))
                        (parameter-names self) newparams))

(sclos-method (<agent> <symbol>) (extra-variable self field) #!void)
(sclos-method (<agent> <symbol>) (extra-variable-list self) '())


(sclos-method <agent> (query self kernel . args)
              (apply (my 'kernel) (append (list 'query) args)))
(sclos-method <agent> (run-at self x)
```

```
              (let ((tq (cons x (my 'timestep-schedule))))
                (set-my! 'timestep-schedule (sort tq <=))))

(add-method
 run
 (make-method
  (list <agent>)
  (lambda (run-parent self T pstop pkernel)
    (let ((my (lambda (x) (slot-ref self x)))
          (set-my! (lambda (x y) (slot-set! self x y)))
          (kernel (slot-ref self 'kernel)))
      (kdnl* 'run-model-body "<agent>" (class-name-of self))
      (let ((t T))
        (let ((stop pstop))
          (let ((kernel pkernel))
            (let ((ttr (begin
                         (set-my! 'timestep-schedule
                                  (prune-local-run-queue
                                   (my 'subjective-time)
                                   (my 'timestep-schedule)))
                         (my 'timestep-schedule))))
              (let ((dt (interval t (slot-ref self 'dt) stop ttr)))
                (let ((subj-time (my 'subjective-time)))
                  (let ((DT 0))
                    (set-my! 'kernel kernel)
                    (cond
                     ((< subj-time t)
                      (if temporal-fascist
                          (begin
                            (kdnl* "[" (my 'name) ":"(class-name-of self) "]"
                                   "a/an" (my 'representation)
```

```
          "is lost in time at" subj-time "or" t)
        'missing-time)
((letrec
     ((loop-through-time
       (lambda (st ddt)
         (kdnl* 'passing-control-to-model
                "["(my 'name)":"(class-name-of self)"]"
                "Passing control to the model at" t
                "for" (if (< st t)
                          ddt
                          (- (+ t dt) subj-time)))
         (let ((m (run-model-body
                   self
                   subj-time
                   (if (< st t)
                       ddt
                       (- (+ t dt)
                          subj-time)))))
           (cond
            ((eq? m #!void)
             (error (string-append
                     "The model body for "
                     (class-name-of
                      self)
                     " returned an error: #!void")))
            ((eq? dt #!void)
             (error (string-append
                     "dt for "
                     (class-name-of
                      self)
                     " is somehow #!void (error)")))
```

```
            ((eq? DT #!void)
             (error (string-append
                     "DT for "
                     (class-name-of
                      self)
                     " is somehow #!void (error)")))
            ((number? m)
             (set! DT (+ DT m))
             (set! st (+ st ddt))
             (set! subj_time st)
             (cond
              ((< st t)
               (loop-through-time st (min (- t
                                             subj-time)
                                          (my 'dt))))
              ((< st (+ t dt))
                    (loop-through-time
                     st
                     (min (- t subj-time)
                          (my 'dt)
                          dt)))
                   (else #!void))
             ((or (symbol? m) (list? m))
              (kdnl* "BORK!!!" m))
             (else (kdnl* "BORK!!!" m)))
            (else #!void))

          m))))
  loop-through-time)
subj-time
(min (- t subj-time) (my 'dt)))))
```

```
((and (> dt 0.) (>= subj-time (+ t dt)))
 (kdnl* "["
        (my 'name)
        ":"
        (class-name-of self)
        "]"
        "a/an"
        (my 'representation)
        "is driving a DeLorian."
        "  Expected subjective-time to be"
        t
        "but it was"
        subj-time
        "and dt ="
        dt)
 'back-to-the-future)
(#t
 (kdnl* 'passing-control-to-model
        "["
        (my 'name)
        ":"
        (class-name-of self)
        "]"
        "Passing control to the model at"
        t
        "for"
        dt)
 (let ((m (if (isa? self <agent>)
              (run-model-body self t dt)
              dt)))
   (set! DT (+ DT m))
```

```
                    m))
                 (else #!void))
                (if (zero? DT)
                    (and (dnl "******************************")
                         (error "BAD TICK")))
                (if (isa? self <agent>)
                    (set-subjective-time!
                     self
                     (+ DT (my 'subjective-time))))
                (kdnl* '(nesting run-model-body)
                       (class-name-of self)
                       (name self)
                       "Leaving run with "
                       DT
                       " @ "
                       (my 'subjective-time)
                       "["
                       (my 'dt)
                       "]")
                'ok))))))))))


(define blue-meanie #f)


;; This routine does the running since "run" has fixed up the ticks
;; It looks like (run-model-body me t dt) in code
(sclos-method <agent> (run-model-body self t ldt)
              ;;  The model returns the amount of time it actually ran for
              (kdnl* '(nesting run-model-body) (class-name-of self)
                     "Running at " t "+" ldt "[" (my 'dt) "]")
```

```
(if (< dt 0.0) (error 'bad-dt))

(let* ((return (model-body self t ldt)))

   ;; The model's tick is done now, adjust the
   ;; subjective_time to reflect how much time it took
   ;; for the tick
  (if (number? return)
      ;;(set-my! 'subjective-time (+ t return))
      ;; Any non-numeric return value indicate a "condition"
      ;; which by definition means that no time should
      ;; have been used else ... Huston, we have a
      ;; problem....

      (begin
        (dnl (my 'name)
             " returned from its model body with " return)
        return
        ;; Just deal with it.
        ))

  ;; deal with any changes in the entity's
  ;; representation, or the general configuration of the
  ;; model as a whole

  ;; prefix a symbol ('migrate, for
  ;; example) to the return value if it needs to change,
  ;; last bit should be "return"

  (let ((mtrb ((my 'migration-test) self t ldt return)))
    ;; we don't want to make calls to a closure that has vanished
```

```
                (if blue-meanie (set-my! 'kernel #f))

                (if mtrb
                    (if (pair? return)
                        (cons mtrb return)
                        (cons mtrb (list return)))
                    return))
              )
            )  ;; returns the amount of time it ran in the last
               ;; tick, some request to the scheduler or an error
               ;; condition


;; model-body knows "self" "t" "dt" and all its state variables.
;; This particular version of the routine should not call parent-body.
(sclos-model-body <agent>
                (if #t
                    (begin
                      (kdnl* 'track-subjective-times
                             "[" (my 'name) ":" (class-name-of self) "]"
                             " running at " (my 'subjective-time) ":" t)
                      (skip-parent-body)
                      ))
                dt)


;---- <tracked-agent> methods

;----- (initialise)
```

```
(add-method initialize
            (make-method (list <tracked-agent>)
                         (lambda (initialize-parent self args)
                           ;;(dnl "<thing> init")
                           (initialise self
                                       '(track #f tracked-paths #f
                                               track-schedule '()
                                               track-epsilon 1e-6))
                           ;; call "parents" last to make the
                           ;; initialisation list work
                           (initialize-parent)
                           )))


(sclos-method (<tracked-agent> <number> <pair>) (track-locus! self t loc)
              (let ((tr (my 'track)))
                (set-my! 'track
                         (if tr
                             (append (my 'track) (list (cons t loc)))
                             (list (cons t loc))
                         ))
              )
)

(sclos-method (<tracked-agent>) (track self)
              (my 'track))


(sclos-method (<tracked-agent> <number> <pair>) (set-track! self t)
              (set-my! 'track (deep-copy t))) ;; we copy it so that we
                                              ;; aren't subject to the
```

```
                                          ;; track changing under
                                          ;; our feet


(sclos-method (<tracked-agent>) (new-track! self)
              (let ((p (my 'tracked-paths))
                    (t (my 'track)))
                (cond
                 ((and p t) (set-my! 'tracked-paths (cons p t)))
                 (t (set-my! 'tracked-paths (list t))))
                (set-my! 'track #f)))

(sclos-method (<tracked-agent>) (tracks self)
              (my 'tracked-paths))


(sclos-model-body <tracked-agent>
                  (track-locus! self t (my 'location)) ;; even if they
                                                       ;; aren't
                                                       ;; moving
                  (parent-body)
                  dt
                  )



;---- <thing> methods

;----- (initialise)
(add-method initialize
```

```
            (make-method (list <thing>)
                         (lambda (initialize-parent self args)
                           ;;(dnl "<thing> init")
                           (initialise self '(dim #f location #f
                                                  direction #f
                                                  speed #f mass #f
                                                  track #f
                                                  tracked-paths #f))
                           (initialize-parent) ;; call "parents" last
                                               ;; to make the
                                               ;; initialisation list
                                               ;; work
                           )))

;----- (mass)
(add-method mass
            (make-method
             (list <thing>)
             (lambda (call-parent-method self)
               (slot-ref self 'mass))))


;----- (set-mass!)
(add-method set-mass!
            (make-method
             (list <thing> <number>)
             (lambda (call-parent-method self n)
               (if (not (number? n))
                   (error "thing:set-mass! -- bad number")
                   (slot-set! self 'mass n)))))
```

```
;----- (dim)
(add-method dim
            (make-method
             (list <thing>)
             (lambda (call-parent-method self)
               (slot-ref self 'dim))))


;----- (set-dim!)
(add-method set-dim!
            (make-method
             (list <thing> <number>)
             (lambda (call-parent-method self n)
               (if (not (integer? n))
                   (error "thing:set-dim! -- bad integer")
                   (slot-set! self 'dim n)))))

;----- (speed)
(add-method speed
            (make-method
             (list <thing>)
             (lambda (call-parent-method self)
               (slot-ref self 'speed))))


;----- (set-speed!)
(add-method set-speed!
            (make-method
             (list <thing> <number>)
             (lambda (call-parent-method self n)
               (if (not (number? n))
```

```
                (error "thing:set-speed! -- bad number")
                (slot-set! self 'speed n)))))


;----- (location)
(add-method location
            (make-method
             (list <thing>)
             (lambda (call-parent-method self)
               (slot-ref self 'location))))


;----- (set-location!)
(add-method set-location!
            (make-method
             (list <thing>)
             (lambda (call-parent-method self vec)
               (if (not (= (length vec) (slot-ref self 'dim)))
                   (error "thing:set-location! -- bad list length")
                   (slot-set! self 'location vec)))))


;----- (direction)
(add-method direction
            (make-method
             (list <pair>)
             (lambda (call-parent-method self)
               (slot-ref self 'location))))


;----- (set-direction!)
```

```
(add-method set-direction!
            (make-method
             (list <thing> <pair>)
             (lambda (call-parent-method self vec)
               (if (not (= (length vec) (slot-ref self 'dim)))
                   (error "thing:set-direction! -- bad list length")
                   (slot-set! self 'direction vec)))))
```

```
;---- environment methods

(sclos-method <environment> (min-bound self)
              (copy-list (my 'minv)))

(sclos-method <environment> (max-bound self)
              (copy-list (my 'maxv)))

(sclos-method (<environment> <pair>) (contains? self loc)
              (let ((mbounds (min-bound self))
                    (Mbounds (max-bound self))
                    )
                (apply andf (append (map < mbounds eloc)
                                    (map < eloc Mbounds)))))


;; Default environment only has the default value, oddly enough
(sclos-method (<environment> <pair>) (value self loc)
              (my 'default-value))
```

```
(sclos-method (<environment> <pair>) (set-value! self loc val)
              (set-my! 'default-value val))

(sclos-method (<environment> <thing>) (contains? self entity)
              (contains? (location entity))
              )

(sclos-method (<environment> <symbol> <pair>) (value self tag loc . args)
              (my 'default-value))

(sclos-method (<environment> <symbol> <pair>) (set-value! self tag loc val)
              (set-my! 'default-value val))

(sclos-method (<environment> <thing>) (contains? self entity)
              (contains? (location entity))
              )


;;; Local Variables:
;;; mode: scheme
;;; outline-regexp: ";-+"
;;; comment-column:0
;;; comment-start: ";;; "
;;; comment-end:""
;;; End:

;-   Identification and Changes

;--
```

```
;  kernel.scm -- Written by Randall Gray
;  Initial coding:
;     Date: 2008.04.07
;     Location: localhost:/usr/home/gray/Study/playpen/kernel.scm.scm
;
;-  Code


;;(require 'sort)
;;(require 'pretty-print)
;;(require 'common-list-functions)

;;(load "stats-bins.scm")

;;*** The kernel must be loaded *after* the agents are all created --
;;this is so it has access to the "run" "migrate" ... generic
;;methods.


;;-------------------------------------------------
;; Important routines which I Really Ought to Know ;;
;;-------------------------------------------------


;; (queue t stop runqueue . N)
;; t is model time, stop is when the whole model ought to stop
;; runqueue is a list of run-records

;; (q-insert Q rec reccmp) inserts the record "rec" into the queue "Q"
;; sorting records with "reccmp"
;;; Call: (set! rq (q-insert rq (make <whatever> ...) Qcmp))
```

```
(define lookit-running-names #f) ;; emits the name of the running
                                 ;; agent if true

(load "postscript.scm")
;;; This is loaded here since it may be used to produce all sorts of
;;; snapshot data spanning agents


(define developing #t)
(define current-page #f)
(define current-page-number 0)


(define terminating-condition-test
         (lambda rq #f))


;-------------------------------------------------
;               Kernel support
;-------------------------------------------------

;; Compares the subjective time of two agents
(define (Qcmp r1 r2)
  (let ((st1 (subjective-time r1))
        (st2 (subjective-time r2)))
    (cond
     ((< st1 st2) #t)
     ((> st1 st2) #f)
     (#t (let ((p1 (priority r1))
               (p2 (priority r2)))
```

```
           (cond
            ((> p1 p2) #t)
            ((< p1 p2) #f)
            (#t (< (jiggle r1) (jiggle r2)))
            ))))))

(define (map-q arg q)
  (map (lambda (s) (arg s)) q))


(define (running-queue rq stop)
  (cond
   ((null? rq) #f)
   ((not (pair? rq)) #f)
   ((not (list? rq)) #f)
   ((not (agent? (car rq))) #f)
   (#t (let ((f (car rq)))
         (< (subjective-time f) stop)))))

;; returns the "time" of the agent at the head of the queue (unused)
(define (model-time rq stop)
  (if (running-queue rq stop)
      (if (or (null? rq) (not (list? rq)))
          rq
          (let ((f (car rq)))
            (subjective-time f))
          )
      'end-of-run
      )
  )
```

```
;; returns an interval (tick-length) based on the current time, the
;; desired tick length, the nominated end of the run and a list of
;; target times
(define (interval t ddt stopat tlist)
  ;; tlist is a sorted queue of times to run
  (if (< (- stopat t) ddt)
      (set! ddt (- stopat t)))

  (cond
   ((null? tlist)
    ddt)
   ((and (list? tlist)
         (number? (car tlist))
         (eq? (car tlist) t)
         )
    ddt)
   ((and (list? tlist)
         (number? (car tlist))
         )
    (- (car tlist) t))
   (else 'bad-time-to-run)))

;; remove stale times in the time-to-run queue
(define (prune-local-run-queue tm ttr)
  (let (
;       (call-starts (cpu-time))
        (r '())
        )
    (set! r (let loop ((l ttr))
              (if (or (null? l)
                      (> tm (car l))
```

```
                    )
                l
                (loop (cdr l)))))
;   (set! kernel-time (+ kernel-time (- (cpu-time) call-starts)))
    r )
  )

;; insert a run record in the queue Q
(define (q-insert Q rec reccmp)
  (let ((j (jiggle rec))
;       (call-starts (cpu-time))
       )
    (set! Q (remove rec Q))

    (if (and (positive? j) (< j 1.0))
        (set-jiggle! rec (abs (random-real))))

    (let* ((f (append Q (list rec)))
           (sf (sort f reccmp))
           )
;     (set! kernel-time (+ kernel-time (- (cpu-time) call-starts)))
      sf)
    )
  )


;-------------------------------------------------
;           Kernel -- the main loop
;-------------------------------------------------

;; this is so we catch runaway growth
```

```
(define (test-queue-size rq N)
  (if (and (number? N) (> (length rq) N))
      (begin
        (dnl "There are " (length rq)
             " entries in the runqueue when we expect at most " N);
        (dnl "These entities total "
             (apply + (map (lambda (x) (members x)) rq)) " members");

        (map (lambda (me)
               (dnl " " (representation me) ":" (name me) " (" me ") @ "
                    (subjective-time me) " with " (members me))) rq)
        (abort "Failed queue size test")
        ))
  )


;; This is the main loop which runs agents and reinserts them after
;; they've executed.
;;
;; There is extra support for inter-agent communication and migration.
;;
;;     The queue doesn't (and *shouldn't*) care at all about how much
;;     time the agents used.

;;

(define (queue t stop runqueue . N)
  (set! N (if (null? N) #f (car N)))

  (let loop ((rq runqueue))
    (cond
```

```
     ((terminating-condition-test rq)
      (list 'terminated rq))
     ((null? rq)
      'empty-queue)
     ((and (list? rq) (symbol? (car rq)))
      rq)
     ((file-exists? "halt")
      (delete-file "halt")
      (shutdown-agents rq)
      rq)
     ((and (< t stop) (running-queue rq stop))
      (set! t (apply min (map-q subjective-time rq)))
      (test-queue-size rq N)
      (set! rq (run-agent t stop rq))
      (test-queue-size rq N)
          (loop rq))
     )
    )
  )

;; converts the parameter vector "params" to reflect a new representation
(define (convert-params params rep)
  (let* ((newparams params)
         )
    (list-set! newparams 3 rep)
    (list-set! newparams 5 (if (eq? rep 'individual) 1 0))
    newparams
    ))


(define (distances-to what agentlist loc)
```

```
  (map (lambda (agent)
         (if (and (procedure? agent)
                  (eq? (representation agent) what))
             (distance loc (location agent ))
             2e308)
         )
       agentlist)
  )

(define (distances-to-agents agentlist loc)
  (map (lambda (agent)
         (if (procedure? agent)
             (distance loc (location agent))
             1e308)
         )
       agentlist)
  )

(define (distances-to-populations agentlist loc)
  (distances-to 'population agentlist loc))


;; returns the index of the left-most value
(define (min-index n-list)
  (cond
   ((not (list? n-list)) 'not-a-list)
   ((null? n-list) #f)
   ((not (apply andf (map number? n-list)))
    'Non-number-entry)
   (#t
    (let* ((k (length n-list))
```

```
          (result (let loop ((ix 0)
                             (best #f)
                             )
                    (if (>= ix k)
                        best
                        (let ((n (list-ref n-list ix)))

                          (cond
                           ((infinite? n) ;; skip invalid entries
                            (loop (1+ ix) best))
                           ((and (number? n)
                                 (or
                                  (not best)
                                  (let ((b (list-ref n-list best)))
                                    (or (infinite? b)
                                        (and (number? best) (<= n b)))))
                                 )
                            (loop (1+ ix) ix))
                           (#t (loop (1+ ix) best)))
                          ))))
      (if (or (not result) (infinite? result) (>= result 1e308))
          #f
          result)
      ))
   )
  )

(define (kernel-call Q client query #!optional args)
  (cond
   ((procedure? query) ;; Do not return the client
    (filter (lambda (x) (and (query x) (not (eq? x client)))) Q))
```

```
   ((symbol? query)
    (cond
     ((eq? query 'time) (model-time Q +inf.0))
     ((eq? query 'agent-count) (length Q))
     ((eq? query 'next-agent) (if (null? Q) Q (car Q)))
     ((eq? query 'min-time)
      (if (null? Q) 0 (apply min (map subjective-time Q))))
     ((eq? query 'max-time)
      (if (null? Q) 0 (apply max (map subjective-time Q))))
     ((eq? query 'mean-time)
      (if (null? Q)
          0
          (/ (apply + (map subjective-time Q)) (length Q))))
     (#t (abort 'kernel-call:not-defined)))
    )
   (#t (abort 'kernel-call:bad-argument))
   )
  )


;; The agent function, "process",  must respond to the following things
;;    (snapshot agent)

;;    (i-am agent)
;;    (is-a agent)
;;    (representation agent)
;;    (name agent)
;;    (subjective-time agent)
;;    (dt agent)
```

```
;;    (parameters agent)

;;    (run-at agent t2)
;;    (run agent currenttime stoptime kernel)


;; The return values from agents fall into the following categories:

;;    a symbol
;;       is automatically inserted at the head of the queue and
;;       execution is terminated (for debugging)

;;    dt
;;       normal execution

;;    (list 'introduce-new-agents dt list-of-new-agents)
;;       indicates that the agents in list-of-new-agents should be
;;       added to the simulation

;;    (list 'remove-me dt)
;;       indicates that an agent should be removed from the simulation

;;    (list 'migrate dt list-of-suggestions)
;;       the list-of-suggestions is so that an external assessment routine

;;    (list 'domain dt message-concering-domain-problem)
;;       usually something like requests for greater resolution...

;;    (list 'migrated dt)
;;      indicates that a model has changed its representation for some reason
```

```
(define (prep-agents Q start end . args)
  (kdnl* 'prep "Prepping from" start "to" end "    with" Q)
  (for-each
   (lambda (A)
     (kdnl* 'prep "Prepping in lambda" (name A))
     (let ((kernel (lambda x (apply kernel-call (append (list rq A) x ))))
           )
       (kdnl* 'prep "Prepping in apply" (name A))
       (apply agent-prep (append (list A start end kernel) args))
       )
     )
   Q)
  )

(define (shutdown-agents Q . args)
  (for-each
   (lambda (A)
     (let* ((kernel (lambda x (apply kernel-call (append (list rq process) x))))
           )
       (apply agent-shutdown (append (list A kernel) args))
       )
     )
   Q)
   )


;; Dispatches a call to the agent through the "run" routine. It also
;; handles special requests from the agent like mutation and spawning.
;; subjective time is set in  (run ...)
```

```
(define run-agent
  (let ((populist '())) ;; Remember the population list across
                        ;; invocations ... (equiv to a "static" in C,
                        ;; folks.)


    (lambda (t stop rq . N) ;; This is the function "run-agent"
      (set! N (if (null? N) #f (car N)))

      (let* ((rq rq)
             (process (if (and (not (null? rq)) (list? rq)) (car rq) #f))
             ;; ... either false or the lambda to run
             (agent-state (slot-ref process 'agent-state))
             )

        (or (eq? agent-state 'running)
            (eq? agent-state 'ready-to-run)
            (and (eq? agent-state 'ready-for-prep)
                 (abort (string-append
                         "Attempted to run "
                         (symbol->string (class-name-of process)) ":"
                         (name process) " before it has been prepped")))
            (abort (string-append
                    "Attempted to run "
                    (symbol->string (class-name-of process)) ":"(name process)
                    " when it is in the state " (object->string agent-state))))

        (if process (kdnl* 'running "running" (name process) "at" t))

        (test-queue-size rq N)
        ;; remove the agent's run request from the top of the queue
```

```
(set! rq (remove process rq))
(test-queue-size rq N)

(kdnl* 'run-agent "In run-agent")

(slot-set! process 'agent-body-ran #f) ;; Mark things as not
;; having run through
;; the body list

;; Here result should be a complex return value, not the
;; number of ticks used.
(let* ((kernel
        (lambda x (apply kernel-call (append (list rq process) x ))))

       (result (if (symbol? process)
                   'bad-runqueue
                   (if (eq? agent-state 'suspended)
                       ;; A suspended agent "consumes" its
                       ;; time without doing anything, except
                       ;; update its subj. time.
                       (let ((dt (interval t
                                           (slot-ref process 'dt) stop
                                           (slot-ref process 'timestep-schedule))
                                 )
                             (st (slot-ref process 'subjective-time)))
                         'ok)
                       ;; If the thing queued is actually an
                       ;; agent, run the agent
                       (if (member <agent>
                                   (class-cpl (class-of process)))
                           (let ((r (run process t stop kernel)))
```

```
                        (if lookit-running-names
                            (dnl (slot-ref process 'name)
                                 " " (subjective-time process)
                                 " " r))
                        (kdnl* 'run-agent "finished with run"
                               (name process) "@"
                               (subjective-time process) "+" r)
                        r
                        )
                       (begin
                        (dnl "Found a non-agent, dropping it.")
                        'not-an-agent)
                       )
                   )
                ))
     )

(if (not (slot-ref process 'agent-body-ran))
    (begin
      (error
       (string-append
        "The agent " (class-name-of process) ":" (name process)
        " failed to chain back to the base <agent> "
        "sclos-model-body.\n"
        "This suggest that things have gone very wrong; "
        "either call (parent-body) or (skip-parent-body).")
       )
      'missed-model-body))
    (let ()
      (cond
       ((eq? result 'ok)
```

```
 (set! rq (q-insert rq process Qcmp))
 rq)

((number? result)
 ;; The result (in this case) is the amount of time used
 ;; measured from the subjective-time of the agent.
 ;; q-insert knows how to find out "when" the agent is,
 ;; and will re-insert it correctly.  subjective-time is
 ;; updated
 (abort "(run ...) returned a number rather than a state")
 (set! rq (q-insert rq process Qcmp)))

((symbol? result) ;;-----------------------------------------------
 (let ()
   (dnl "Got " result)

   (cond
    ((eq? result 'remove)
     rq)
    (else
     (cons result rq)))
   ))
((eq? result #!void)
 (let ((s
        (string-append "A " (symbol->string
                             (class-name-of process))
                       " tried to return a void from its "
                       "model-body.  This is an error")))
   ;;(Abort s)
   (abort s)
   ))
```

```
   (#t
    (set! rq (q-insert rq process Qcmp)))
   ((list? result)
    (case (car result)

      ;; Remove ================================================================
      ('remove
       rq
       ) ;; end of the migration clause

      ;; Migrate to a different model representation ===========================
      ('migrate
       #f
       ) ;; end of the migration clause

      ;; insert spawned offspring into the
      ;;system ** not implemented in make-entity
      ('spawnlist ;;-------------------------------------------------------------
       #f
       )
      (else 'boink)
      ) ; case
    ) ; cond clause
   )
  )
(test-queue-size rq N)
(kdnl* 'run-agent "Finished with run-agent" (name process)
       "@" (subjective-time process))
;; ********** THIS IS NOT THE ONLY WAY TO DO THIS **************
;; One might need to have a method that will take a kernelcall procedure from
;; another agent if there is out-of-band activity that requires a kernelcall.
```

```
              ;; This is the conservative option.
              rq)
          )
        )
      )
    )

(define nested-agents '())

;; Q is the preloaded run-queue
;; Start and End are numbers s.t. Start < End
(define (run-simulation Q Start End . close-up-shop)
  (prep-agents Q Start End)

  (set! Q (queue Start End Q))

  ;; We don't shut down just now, we are still developing
  (if (not developing) (shutdown-agents Q))

  (if (and (not (null? close-up-shop)) (procedure? (car close-up-shop)))
      ((car close-up-shop)))
  )

(define (continue-simulation Q End . close-up-shop)
  (set! Q (queue (subjective-time (car Q)) End Q))
  (if (and (not (null? close-up-shop)) (procedure? (car close-up-shop)))
      ((car close-up-shop)))
  )
```

```
;-  The End

;;; Local Variables: ***
;;; mode: scheme ***
;;; outline-regexp: ";-+" ***
;;; comment-column:0 ***
;;; comment-start: ";;; "  ***
;;; comment-end:"***" ***
;;; End: ***

; -*- mode: scheme; -*-
;-  Identification and Changes

;--
;  log.scm -- Written by Randall Gray
;  Initial coding:
;     Date: 2016.07.13
;     Location: zero:/home/randall/Thesis/Example-Model/model/log.scm
;
;  History:

;-  Discussion

;;; This pulls in the files for the log class

(load "log-classes.scm")
(load "log-declarations.scm")
(load "log-methods.scm")

(set! logger-tags '(logfile snapshot log-map
                            log-data log-table
```

```
                          log-agent-table
                          log-table* log-agent-table*))

(for-each register-submodel logger-tags)

(register-submodel 'logfile)
(register-submodel 'snapshot)
(register-submodel 'log-map)
(register-submodel 'log-data)
(register-submodel 'log-table)
(register-submodel 'log-agent-table*)
(register-submodel 'log-table*)
(register-submodel 'log-agent-table)



;-  The End


;;; Local Variables:
;;; comment-end: "-;" -;
;;; comment-start: ";;; " -;
;;; mode: scheme -;
;;; outline-regexp: ";-+" -;
;;; comment-column: 0 -;
;;; End:

;-  Identification and Changes

;;; #|
```

```
;;; The way this works is like so:

;;;   An agent is created which has a list of entities which are
;;;   polled when it runs (it calls "log-data") to cause the entities
;;;   to generate appropriate output which is sent to a port.  Before
;;;   it processes the list of entities, it calls "prepare-to-dump",
;;;   and after it has finished it calls "finished-dumping" -- these
;;;   two routines handle pagination page wrapping.

;;; |#

(define logger-tags '())


(define <introspection>
  (make-class (inherits-from <agent>)
              (state-variables file filename filetype format variables
                               variables-may-be-set missing-val show-field-name
                               preamble-state introspection-list
                               timestep-epsilon)))
;;- file is the output handle
;;  if filename is not a string, things go to stdout
;-  if append-time is not false it must either be true or the size of
;;- the number (digit count)
;;- if filetype is defined it must be a string and it will be appended
;;  to the filename
;;- the introspection list is the list of agent to be looked at
;;- the timestep schedule is the set of times to run at

(register-class <introspection>)
```

```
(define <logfile> (make-class (inherits-from <introspection>)
                              (no-state-variables)))
;;- file is the output handle
;;- if filename is not a string, things go to stdout
;;- if append-time is not false it must either be true or the size of
;;  the number (digit count)
;;- if filetype is defined it must be a string and it will be appended
;;  to the filename

(register-class <logfile>)

(define <snapshot> (make-class (inherits-from <introspection>)
                               (state-variables lastfile currentfile)))
;;- file is the output handle
;;- if filename is not a string, things go to stdout
;;- if append-time is not false it must either be true or the size of
;;  the number (digit count)
;;- if filetype is defined it must be a string and it will be appended
;;  to the filename

(register-class <snapshot>)


;; This is the basis for all the logging things like the log-map agent
;; and the log-data


(define <log-map> (make-class (inherits-from <snapshot>)
                              (state-variables ps png)))
;; ps and png indicate whether the files should be left on disk at the
;; end of the step
```

```
(register-class <log-map>)

(define <log-data> (make-class (inherits-from <logfile>)
                               (no-state-variables)))

;;- projections is an association list of slot-names and the projection
;;  functions to apply before output
;;- Data is generated using

(register-class <log-data>)

(define <log-agent-table> (make-class (inherits-from <log-data>)
                                      (state-variables target-agent)))

;;- projections is an association list of slot-names and the projection
;;  functions to apply before output
;;- Data is generated using

(register-class <log-agent-table>)

(define <log-table> (make-class (inherits-from <log-data>)
                                (no-state-variables)))
;;- projections is an association list of slot-names and the
;;  projection functions to apply before output
;;- Data is generated using

(register-class <log-table>)

(define <log-agent-table*> (make-class (inherits-from <log-agent-table>)
                                       (no-state-variables)))
```

```
;;- <log-agent-table*> hands off most of the output generation to the
;;  class entity being output
;;- projections is an association list of slot-names and the
;;  projection functions to apply before output
;;- Data is generated using

(register-class <log-agent-table*>)

(define <log-table*> (make-class (inherits-from <log-table>)
                                 (no-state-variables)))

;;- <log-table*> hands off most of the output generation to the class
;;  entity being output
;;- projections is an association list of slot-names and the
;;  projection functions to apply before output
;;- Data is generated using

(register-class <log-table*>)


;-  The End


;- Local Variables:
;- mode: scheme
;- outline-regexp: ";-+"
;- comment-column:0
;- comment-start: ";;;- "
;- comment-end:""
;- End:
```

```
;-  Identification and Changes

(declare-method introspection-list
                "return the introspection list")
(declare-method set-introspection-list!
                "set the list of agents to be examined")
(declare-method introspection-times
                "return the introspection list")
(declare-method set-introspection-times!
                "set the list of agents to be examined")
(declare-method page-preamble
                "open files & such")
(declare-method log-this-agent
                "log an agent's data")
(declare-method page-epilogue
                "make things tidy")
(declare-method set-variables!
                "set the list of variables")
(declare-method extend-variables!
                "extend the list of variables")


(declare-method schedule-times
                "return a schedule")
(declare-method schedule-epsilon
                "return a schedule's epsilon")
(declare-method set-schedule-times!
                "set the times the agents is scheduled to do something")
(declare-method set-schedule-epsilon!
                "set the epsilon for the schedule")
(declare-method insert-schedule-time!
```

```
                "insert a time into a schedule")
;;(declare-method flush-stale-schedule-entries!
;;   "name says it all, really...")
;;(declare-method scheduled-now?
;;   "name says it all, really...")

(declare-method log-data
                "err, ...log data to an open output")
(declare-method emit-page
                "set the list of agents to be examined")
(declare-method open-p/n
                "open a port for logging")
(declare-method close-p/n
                "close a logging port")


(declare-method map-log-track-segment
                "description needed")
(declare-method map-projection
                "description needed")
(declare-method set-map-projection!
                "description needed")
(declare-method map-log-data
                "specific for postscript output")
(declare-method map-emit-page
                "specific for postscript output")

(declare-method data-log-track-segment
                "description needed")
(declare-method data-log-data
                "specific for data output")
```

```
(declare-method data-emit-page
                "specific for data output")

;-  The End


;;; Local Variables:
;;; mode: scheme
;;; outline-regexp: ";-+"
;;; comment-column:0
;;; comment-start: ";;; "
;;; comment-end:""
;;; End:

;-  Identification and Changes



"In the logging code, a 'page' is notionally a pass through the target
agent list.  The page-preamble may open a file and write some
preliminary stuff, or it may just assume that the file is open already
and do nothing at all.  Similarly the page-epilogue does things like
close pages and emit 'showpage' for postscript stuff.
"



(define introspection-priority 10000)

(define (schedule-times self class-sym)
```

```
  (if (eq? class-sym 'introspection) (schedule-times self T 'timestep)
      (slot-ref self
                (string->symbol
                 (string-append (symbol->string class-sym) "schedule")))))

(define (schedule-epsilon self class-sym)
  (if (eq? class-sym 'introspection) (schedule-epsilon self T 'timestep)
      (slot-ref self
                (string->symbol
                 (string-append
                  (symbol->string class-sym)
                  "epsilon")))))

(define (set-schedule-times! self class-sym lst)
  (if (eq? class-sym 'introspection) (set-schedule-times! self T 'timestep)
      (slot-set! self (string->symbol (string-append (symbol->string class-sym)
                                                     "schedule")) lst)))

(define (insert-schedule-time! self class-sym t)
  (if (eq? class-sym 'introspection) (insert-schedule-time! self T 'timestep)
      (slot-set! self (string->symbol (string-append (symbol->string class-sym)
                                                     "schedule"))
                 (uniq (sort (cons t (schedule-times self class-sym)) <=)))))

(define (set-schedule-epsilon! self class-sym val)
  (if (eq? class-sym 'introspection) (set-schedule-epsilon! self T 'timestep)
      (slot-set! self (string->symbol (string-append (symbol->string class-sym)
                                                     "epsilon"))
                 val)))
```

```
(define (flush-stale-schedule-entries! self class-sym)
  (if (eq? class-sym 'introspection)
      (flush-stale-schedule-entries! self T 'timestep)
      (let ((cs-sym (string->symbol (string-append
                                     (symbol->string class-sym) "-schedule")))
            (ce-sym (string->symbol (string-append
                                     (symbol->string class-sym) "-epsilon"))))
        (slot-set! self
                   cs-sym
                   (filter (lambda (x) (> (+ x (slot-ref self ce-sym))
                                          (slot-ref self 'subjective-time))
                                       )
                           (slot-ref self cs-sym))))) )


(define (scheduled-now? self T class-sym)
  (if (eq? class-sym 'introspection)
      (scheduled-now? self T 'timestep)
      (let ((cs-sym (string->symbol (string-append
                                     (symbol->string class-sym)
                                     "-schedule")))
            (ce-sym (string->symbol (string-append
                                     (symbol->string class-sym)
                                     "-epsilon"))))
        (kdnl* '(log-sched-error) "[Getting" cs-sym " = " (slot-ref self cs-sym)
               " and " ce-sym " = " (slot-ref self ce-sym))
        (let ((sched (slot-ref self cs-sym)))
          (let ((v (not (or
                         (null? sched)
                         (zero?
                          (length
```

```
                          (filter
                           (lambda (x)
                             (<= (abs (- T x)) (slot-ref self ce-sym)))
                           sched)))))))
            (flush-stale-schedule-entries! self class-sym)
            v)))))

(define (introspection-filename filename filetype #!optional t)
  (if (string? filename)
      (if t
          (string-append filename "-" (pno t 6) "0" filetype)
          (string-append filename filetype))
      #f))




(sclos-method <agent> (map-projection self)
              (my 'map-projection))

(sclos-method (<agent> <procedure>) (set-map-projection! self p)
              (set-my! 'map-projection p))

;; Logger agents (things that inherit from introspection, really) have
;; a high priority; as a consequence they get sorted to the front of a
;; timestep
(add-method
 initialize
 (make-method (list <introspection>)
              (lambda (initialize-parent self args)
```

```
                ;;(dnl "<thing> init")
                (initialise self (list 'type 'introspection
                                       'priority introspection-priority
                                       'jiggle 0 'introspection-list '()
                                       'timestep-epsilon 1e-6 'file #f
                                       'filename #f 'filetype #f
                                       'format 'text 'missing-val "NoData"
                                       'show-field-name #f 'preamble-state '()
                                       'variables-may-be-set #t
                                       ))
                (initialize-parent) ;; call "parents" last to make the
                                    ;; initialisation list work
                )))

(sclos-method <introspection> (agent-prep self start end kernel . args)
              (agent-prep-parent)
              )

(sclos-method <introspection> (agent-shutdown self . args)
              (let ((file (my 'file)))
                (if (and (my 'file)
                         (output-port? (my 'file))
                         (not (memq (my 'file)
                                    (list (current-output-port)
                                          (current-error-port)))))
                    (close-output-port file))
                (set-my! 'file #f)
                (agent-shutdown-parent)
                ))

(sclos-model-body <introspection>
```

```
(kdnl* '(log-* introspection-trace)
       "[" (my 'name) ":" (class-name-of self) "]"
       "Introspection: model-body")

(let ((sched (my 'timestep-schedule))
      )

  (set! dt (if (and (pair? sched) (< (car sched) (+ t dt)))
               (- (car sched) t)
               dt))

  (kdnl* '(log-* introspection-trace)
         "      list:     " (my 'introspection-list))
  (kdnl* '(log-* introspection-trace)
         "      schedule: "
         (list-head (my 'timestep-schedule) 3)
         (if (> (length (my 'timestep-schedule)) 3)
             '... ""))

  (set-my! 'variables-may-be-set #f)
  (emit-page self)

  ;(skip-parent-body)
  (parent-body)
  ;;(max dt (* 2.0 dt))
  dt
))
```

```
(sclos-method (<introspection> <agent>) (insert-agent! self target)
              (set-my! 'introspection-list (cons target
                                                 (my 'introspection-list))))

(sclos-method (<introspection> <agent>) (append-agent! self target)
              (set-my! 'introspection-list (append (my 'introspection-list)
                                                   (list target))))

(sclos-method <introspection> (introspection-list self)
              (my 'introspection-list))
(sclos-method <introspection> (introspection-times self)
              (my 'timestep-schedule))

(sclos-method (<introspection> <list>) (set-introspection-list! self lst)
              (set-my! 'introspection-list lst))
(sclos-method (<introspection> <list>) (set-introspection-times! self lst)
              (set-my! 'timestep-schedule lst))

(sclos-method (<introspection> <list>) (set-variables! self lst)
              (if (and (my 'variables-may-be-set) (list? lst))
                  (set-my! 'variables lst)
                  (abort "cannot extend variables after it starts running")
                  ))

(sclos-method (<introspection> <list>) (extend-variables! self lst)
              (if (and (my 'variables-may-be-set) (list? lst))
                  (set-my! 'variables (unique* (append (my 'variables) lst)))
                  (abort "cannot extend variables after it starts running")
                  ))
(define (cnc a) (class-name-of (class-of a)))
(define (nm? a) (if (isa? a <agent>) (slot-ref a 'name) a))
```

```
(sclos-method (<introspection>) (emit-page self)
              (kdnl* '(log-* introspection-trace)
                     "[" (my 'name) ":" (class-name-of self) "]"
                     "Introspection: emit-page")
              (let ((format (my 'format)))
                (page-preamble self self format) ;; for snapshots,
                                                 ;; this will be
                                                 ;; "opening", for
                                                 ;; logfiles, it will
                                                 ;; only open the
                                                 ;; first time

                (for-each
                 (lambda (ila)
                   (kdnl* '(log-* introspection-trace) "   processing "
                          (cnc ila) " "  (procedure? ila))
                   (log-data ila self format self (my 'variables))
                   #f
                   )
                 (my 'introspection-list))
                )
              (page-epilogue self self (slot-ref self 'format))
              )


;---- snapshot methods

(sclos-method <snapshot> (initialize self args)
              (initialize-parent) ;; call "parents" last to make the
                                  ;; initialisation list work
```

```
              (initialise self (list 'type snapshot 'lastfile #f
                                     'currentfile #f))
              )

(sclos-use-parent-body <logfile>)


(sclos-method <snapshot> (page-preamble self logger format)
              (kdnl* '(introspection snapshot)"[" (my 'name) ":"
                     (class-name-of self) "]" "is preparing to dump")
              (let ((filename (my 'filename))
                    (filetype (my 'filetype))
                    (file (my 'file))
                    (t (my 'subjective-time))
                    )

                (cond
                 ((not (or (not filename) (string? filename)))
                  (error (string-append (my 'name)" has a filename which "
                                        "is neither false, nor a string.")))

                 ((not (or (not filetype) (string? filetype)))
                  (error (string-append (my 'name) " has a filetype which "
                                        "is neither false, nor a string.")))

                 ((not (number? t))
                  (error (string-append (my 'name) " has a subjective time "
                                        "which is not a number.")))
                 )

                ;; Open a new file
```

```
(cond
 ((not file)
  (let ((fn (introspection-filename (my 'filename)
                                    (my 'filetype) t)))
    (kdnl* '(introspection snapshot) "[" (my 'name) ":"
           (class-name-of self) "]" "opening" fn)
    (set-my! 'lastfile (my 'currentfile))
    (set-my! 'currentfile fn)
    (if (zero? (string-length fn))
        (set! file (current-output-port))
        (set! file (open-output-file fn)))
    ))
 ((memq file (list (current-output-port) (current-error-port)))
  ;; do nothing really
  (kdnl* '(introspection snapshot) "[" (my 'name) ":"
         (class-name-of self) "]"
         "is writing to stdout or stderr")
  #!void
  )
 (else
  (kdnl* '(introspection  snapshot) "[" (my 'name) ":"
         (class-name-of self) "]" " "
         "has hit page-preamble with a file that is still open."
         "\nThis is an error.\nClosing the file ("
         (my 'lastfile) ") and continuing.")
  (close-output-port file)
  (set-my! 'file #f)
  (let ((fn (introspection-filename (my 'filename)
                                    (my 'filetype) t)))
    (set-my! 'lastfile (my 'currentfile))
    (set-my! 'currentfile fn)
```

```
                    (if (zero? (string-length fn))
                        (set! file (current-output-port))
                        (set! file (open-output-file fn)))
                    )
                   )
                  )
                 (set-my! 'file file)))

(sclos-method <snapshot> (page-epilogue self logger format)
              (let ((file (my 'file)))
                (if (and file (not (memq file (list (current-output-port)
                                                    (current-error-port)))))
                    (begin
                      (kdnl* '(introspection snapshot) "[" (my 'name) ":"
                             (class-name-of self) "]"
                             "is closing the output port")
                      (close-output-port file)
                      (set-my! 'file #f)))))


(sclos-use-parent-body <snapshot>)

;---- logfile methods

(sclos-method <logfile> (page-preamble self logger format)
              (kdnl* '(introspection logfile) "[" (my 'name) ":"
                     (class-name-of self) "]" "is preparing to dump")
              (let ((filename (my 'filename))
                    (file (my 'file))
                    )
```

```
                (if (not (or (not filename) (string? filename)))
                    (error (string-append (my 'name) " has a filename which is "
                                          "neither false, nor a string.")))

                ;; Open a new file
                (if (not file)
                    (begin
                      (kdnl* '(introspection logfile) "[" (my 'name) ":"
                             (class-name-of self) "]" "is opening a log file")
                      (if (zero? (string-length filename))
                          (set! file (current-output-port))
                          (set! file (open-output-file filename)))
                      )

                    (set-my! 'file file)))
              )

(sclos-method <logfile> (page-epilogue self logger format)
              (kdnl* '(introspection logfile) "[" (my 'name) ":"
                     (class-name-of self) "]" "has finished a dump")
              #!void)


;---- log-map methods


;----- (initialize)
(add-method initialize
            (make-method (list <log-map>)
                         (lambda (initialize-parent self args)
                           ;;(dnl "<thing> init")
```

```
                          (initialize-parent) ;; call "parents" last
                                              ;; to make the
                                              ;; initialisation list
                                              ;; work
                          (initialise self '(type log-map format ps))
                          ;; keep all files
                          )))

(sclos-use-parent-body <log-map>)



(sclos-method <log-map> (page-preamble self logger format)
              ;; This *must* replace it's parent from <snapshot> since
              ;; it doesn't work with a traditional port
              (kdnl* '(log-* log-map) (name self) "[" (my 'name) ":"
                     (class-name-of self) "]" "in page-preamble")
              (let ((filename (my 'filename))
                    (filetype (my 'filetype))
                    (file (my 'file))
                    (t (my 'subjective-time))
                    )

                (cond
                 ((not (or (not filename) (string? filename)))
                  (error (string-append (my 'name) " has a filename which is "
                                        "neither false, nor a string.")))

                 ((not (or (not filetype) (string? filetype)))
                  (error (string-append (my 'name) " has a filetype which is "
                                        "neither false, nor a string.")))
```

```
 ((not (number? t))
  (error (string-append (my 'name) " has a subjective time "
                        "which is not a number.")))
 )

;; Open a new file
(cond
 ((not file)
  (kdnl* '(introspection log-map) "[" (my 'name) ":"
         (class-name-of self) "]" "is preparing to dump")

  (let ((fn (introspection-filename (my 'filename)
                                    (my 'filetype) t)))
    (set-my! 'lastfile (my 'currentfile))
    (set-my! 'currentfile fn)
    (if (zero? (string-length fn))
        (abort "Oh. Bother.")
        (set! file (make-ps fn '(Helvetica))))
    ))
 ((memq file (list (current-output-port) (current-error-port)))
  ;; do nothing really
  (kdnl* '(introspection log-map) "[" (my 'name) ":"
         (class-name-of self) "]" "has nothing to do")
  #!void
  )
 (else
  (kdnl* '(introspection log-map) "[" (my 'name) ":"
         (class-name-of self) "]"
         " Good, we've hit page-preamble with a file "
         "that is still open.\nClosing the file ("
```

```
                    (my 'lastfile) ") and opening a new one.")
                (close-output-port file)
                (set-my! 'file #f)
                (let ((fn (introspection-filename (my 'filename)
                                                  (my 'filetype) t)))
                  (set-my! 'lastfile (my 'currentfile))
                  (set-my! 'currentfile fn)
                  (if (zero? (string-length fn))
                      (abort "Oh. Bother.")
                      (set! file (make-ps fn '(Helvetica))))
                  )
                 )
                )
               (set-my! 'file file)))

(sclos-method <log-map> (page-epilogue self logger format)
              ;; This *must* replace it's parent from <snapshot> since
              ;; it doesn't work with a traditional port
              (kdnl* '(log-* log-map) (name self) "[" (my 'name) ":"
                     (class-name-of self) "]" "has page-epilogue")
              (let ((file (my 'file))
                    (name (my 'currentfile)))
                (if file
                    (begin
                      (file 'close)
                      (set-my! 'file #f)))
                )
              )


;; This logs to an open file
```

```
(sclos-method (<log-map> <procedure> <procedure> <symbol> <list>)
              (log-data self logger format caller targets)
              (lambda (target)
                (kdnl* '(log-* log-map) (name self) "[" (my 'name)
                       ":" (class-name-of self) "]" "in log-data"
                       (class-name-of target) (slot-ref target 'name))
                (let* ((name (slot-ref target 'name))
                       (p (slot-ref self 'map-projection))
                       (ps (slot-ref self 'file))
                       )
                  (ps 'comment "logging data for " name "****************")
                  (ps 'moveto (list (p '(20 20))))
                  (ps 'setgray 0.0)
                  (ps 'Helvetica 14)
                  (ps 'show (string-append (slot-ref self 'name)))
                  (ps 'comment "finished logging data for " name)
                  )))


;---- log-data methods
;----- (initialize)
(add-method initialize
            (make-method (list <log-data>)
                         (lambda (initialize-parent self args)
                           ;;(dnl "<thing> init")
                           (initialise self '(type log-data)) ;; keep all files
                           (initialize-parent) ;; call "parents" last
                                               ;; to make the
                                               ;; initialisation list
                                               ;; work
                           )))
```

```
(sclos-use-parent-body <log-data>)

(sclos-method <log-data> (agent-prep self start end kernel . args)
              ;; This opens the output file on initialisation.
              (agent-prep-parent) ;; parents should prep first
              (kdnl* '(log-* log-data) (name self) "[" (my 'name) ":"
                     (class-name-of self) "]" "in agent-prep")

              (let ((filename (my 'filename))
                    (filetype (my 'filetype)))
                (if (string? (my 'filename))
                    (begin
                      (kdnl* '(log-* log-data) (name self) "[" (my 'name)
                             ":" (class-name-of self) "]" "opening "
                             (introspection-filename filename
                                                     (if filetype filetype "")))
                      (set-my! 'file
                               (open-output-file
                                (introspection-filename filename
                                                        (if filetype
                                                            filetype
                                                            ""))))
                      (current-output-port))
                    (begin
                      (kdnl* '(log-* log-data) (name self) "[" (my 'name) ":"
                             (class-name-of self) "]"
                             "using stdout as the output file " )
                      (set-my! 'file (current-output-port))
                      )
                    )
```

```
                )
               (if (null? (my 'variables))
                   (let ((vars (reverse
                                (unique*
                                 (reverse
                                  (append
                                   '(name subjective-time)
                                   (apply append
                                          (map extra-variable-list
                                               (my 'introspection-list)))))))))
                     (slot-set! self 'variables vars)))
               )


(sclos-method <log-data> (agent-shutdown self . args)
               (kdnl* '(log-* log-data) (name self) "[" (my 'name) ":"
                      (class-name-of self) "]" "in agent-shutdown")
               (if (and (my 'file) (output-port? (my 'file))
                        (not (memq (my 'file)
                                   (list (current-output-port)
                                         (current-error-port)))))
                   (begin
                     (close-output-port (my 'file))
                     (set-my! 'file #f) ;; leave it the way it should be left
                     ))
               (agent-shutdown-parent) ;; Parents should shutdown last
               )

(sclos-method <log-data> (page-preamble self logger format)
               (page-preamble-parent) ;; opens the file
```

```
(if (not (output-port? (my 'file)))
    (abort "Serious problems getting an output port for "
           (my 'name)))

(let ((il (my 'introspection-list))
      (file (my 'file))
      (show-field-name (my 'show-field-name))
      (missing-val (my 'missing-val))
      )
  (case format
    ((ps)
     #f)
    (else


     (if (not (member 'header (my 'preamble-state)))
         (begin
           (if (and (pair? il)
                    (null? (cdr il))) ;; agent name
                                      ;; comes first
                        ;; since it is
                                      ;; easy to prune
                                      ;; the first line
               (begin
                 (display (string-append "# " (name (car il)))
                          (my 'file))
                 (newline file)))

           (let ((header
                  (if missing-val
                      (my 'variables)
```

```
                (let loop ((all-vars '())
                           (entities il))
                  (if (null? entities)
                      (intersection
                       (uniq
                        (map
                         string->symbol
                         (sort (map symbol->string
                                    all-vars)
                               string<?)))
                       (my 'variables))
                      (loop
                       (append
                        (map car
                             (class-slots
                              (class-of (car entities))))
                        (extra-variable-list (car entities))
                        all-vars) (cdr entities))))
                )
              ))
        (display "# " file)
        (for-each
         (lambda (x) (display " " file) (display x file))
         header)
        (newline file))
      (set-my! 'preamble-state
               (cons 'header (my 'preamble-state)))
      )
    )
 )
)
```

```
              )
            )

;; This is typically never called since it "logs" the logfile.  Mostly
;; here as an example.
(sclos-method (<log-data> <procedure> <procedure> <symbol> <list> <boolean>)
              (log-data self logger format caller targets . file)
              (kdnl* '(log-* log-data) (name self) "[" (my 'name) ":"
                     (class-name-of self) "]" "in log-data")
              (let ((file (my 'file))
                    (show-field-name (my 'show-field-name))
                    (subjects (my 'introspection-list))
                    (targets (my 'variables))
                    )
                (for-each (lambda (target)
                            (display "**" file)
                            (for-each ;; field in the variable list
                             (lambda (field)
                               (if show-field-name
                                   (begin
                                     (display " " file)
                                     (display field file)))

                               (cond
                                ((member field
                                         (map car
                                              (class-slots (class-of target))))
                                 (kdnl* '(log-* log-data logging-debug)
                                        "     Dumping " field "="
                                        (if (has-slot? self t)
                                            (slot-ref self t)
```

```
                                        "missing!"))

                                (display " " file)
                                (display (slot-ref target field) file)
                                )
                               ((member field (extra-variable-list target))
                                (display " " file)
                                (display (extra-variable target field) file)
                                )
                               (missing-val
                                (display " " file)
                                (display missing-val file)))
                              )
                             (if #t targets
                                 (filter (not-member (my 'dont-log)) targets)))
                           (newline file)
                           )
                         subjects)
                )
              )


(sclos-method (<log-data>) (page-epilogue self logger format)
              (kdnl* '(log-* log-data) (name self) "[" (my 'name) ":"
                     (class-name-of self) "]" "in page-epilogue")
              (if (and (pair? (my 'introspection-list))
                       (pair? (cdr (my 'introspection-list))))
                  (or #t (newline (my 'file)))
                  ;; We don't want a blank line between each record!
                  ;; -- change #t to #f to get lines between "pages"
                  )
```

```
              )


;-  The End


;;; Local Variables:
;;; mode: scheme
;;; outline-regexp: ";-+"
;;; comment-column:0
;;; comment-start: ";;; "
;;; comment-end:""
;;; End:

; -*- mode: scheme; -*-
;-  Identification and Changes

;--
;  landscape.scm -- Written by Randall Gray
;  Initial coding:
;     Date: 2016.07.13
;     Location: zero:/home/randall/Thesis/Example-Model/model/landscape.scm
;
;-  Discussion

;;  This pulls in the code for the landscape classes (which incorporates populations)

(register-submodel 'ecoservice)
(register-submodel 'patch)
(register-submodel 'dynamic-patch)
(register-submodel 'landscape)
```

```
(register-submodel 'habitat)

(load "landscape-classes.scm")
(load "landscape-declarations.scm")
(load "landscape-methods.scm")

;-  The End


;;; Local Variables:
;;; comment-end: "-;" -;
;;; comment-start: ";;; " -;
;;; mode: scheme -;
;;; outline-regexp: ";-+" -;
;;; comment-column: 0 -;
;;; End:

;-  Identification and Changes

;--
;  landscape-classes.scm -- Written by Randall Gray

;-  Code

;-- Environmental things

(define <ecoservice>
  (make-class (inherits-from <agent>)
              (state-variables
               name ;; Used in output
               sym ;; symbol (possibly used in updates by other agents
```

```
                patch ;; spatial agent associated with the ecoservice
                value ;; current value of the ecoservice
                capacity ;; maximum value of the ecoservice

                r        ;; recovery rate parameter
                ;; -- sharpness of
                ;; transition in growth
                ;; curve(sigmoid) ,
                ;; increase per unit of
                ;; time (linear)

                delta-T-max ;; largest stepsize the ecoservice can accept
                do-growth   ;; #t/#f
                growth-model ;; routine to effect growth of the form (f
                             ;; t dt currentvalue)
                history      ;; maintains  ((t value) ...)
                )))
(register-class <ecoservice>)

(Comment "An <ecoservice> stands for biogenic parts of the system (ground
water) The update function is of the form (lambda (self t dt) ...)
where the args are specific things passed (like location, rainfall)
and any specific 'parameters' ought to be part of the function's
closure. The special growth functions 'sigmoidal and 'linear  can
be specified by passing the appropriate symbol.

A typical construction of an ecoservice might look like
(make <ecoservice>
      'patch P
      'value 42000
      'capacity +inf.0
```

```
      'r 1
      'delta-T-max (days 2)
      'do-growth #t
      'growth-model
         ;;; 'sigmoid
         ;;; 'linear
         (lambda (t dt v)
            (let* ((tssn (/ t 365.0))
                   (ssn (- tssn (trunc tssn))))
                (cond
                  ((<= 0 0.3) (/ r 12))
                  ((<= 0 0.7) (/ r 3))
                  ((<= 0 0.8) (/ r ))
                  (#t (/ r 6)))))
      ;;
)
")


(define <population-system>
  (make-class (inherits-from <ecoservice>)
              (state-variables
               patch ;; spatial agent associated with the ecoservice
               value ;; current value of the ecoservice
               capacity ;; maximum value of the ecoservice
               r        ;; recovery rate parameter (sharpness of
                        ;; sigmoidal growth)
               delta-T-max ;; largest stepsize the ecoservice can
                           ;; accept
               do-growth   ;; #t/#f
               growth-model ;; routine to effect growth of the form
```

```
                        ;;   (f t dt currentvalue)
                history      ;; maintains  ((t value) ...)
                )))


(define <circle> (make-class (inherits-from <object>)
                             (state-variables locus radius)
                             ))
(register-class <circle>)

(define <polygon> (make-class (inherits-from <object>)
                              (state-variables locus point-list)
                              ))
(register-class <polygon>)


(define <boundary> (make-class (inherits-from <object>)
                               (state-variables rep)))
(register-class <boundary>)
(Comment "<boundary> provides the spatial context for <patch>
in the way that <landscape> provides a terrain for <habitat>")



(define <patch> (make-class (inherits-from  <environment> <boundary>)
                            (state-variables service-list)))

(Comment "A patch is a geographic region with a list of ecological
services.")
```

```
(register-class <patch>)

;;
(define <dynamic-patch>
  (make-class (inherits-from <patch>)
              (state-variables
               population-names ;; list of strings associated with
                                ;; each population

               population-symbols ;; unique symbols for each of the
                                  ;; pops

               population-definitions  ;; list of the form ((nameA
                                       ;; dA/dt) ...)
               do-dynamics       ;; #t/#f whether to do the dynamics
                                 ;; or not if this is false, state
                                 ;; values are modified externally
                                 ;; with calls to (set-value!

               do-growth   ;; #t/#f whether to do growth using the
                           ;; ecoservice growth-model or not

               dP    ;; This is generated using rk4* and d/dt-list

               d/dt-list   ;; differential equations which describe
                           ;; the dynamics

               subdivisions)))  ;; the definitions are kept for
                                ;; debugging
```

```
(Comment "<dynamic-patches> are patches that have a system of
differential equations which stand in the place of the simpler
representation of patches with ecoservices.

A straightforward instantiation of a <dynamic-patch> might look like
  (define P (make <dynamic-patch> 'location loc 'radius radius 'type
                  'patch 'representation 'patch
                  'do-growth #f 'do-dynamics #t
                  'population-definitions
                    (list (list \"plants\" 'plant dplant/dt)
                          ... (list \"spiders\" spider dspider/dt)) ))
or
  (define P (make <dynamic-patch> 'location loc 'radius radius 'type
          'patch 'representation 'patch
          'do-growth #f 'do-dynamics #t
          'population-names
            '(\"plants\" \"seeds\" \"aphids\" \"ants\" \"spiders\")
          'population-symbols '(plant seed aphid ant spider)
          'dp/dt-list
             (list dplant/dt dseed/dt daphid/dt dant/dt dspider/dt)
  ))


Definitions look like ((name1 dn1/dt) ... (nameN dnN/dt))
or (eventually) like ((name1 prey-lst pred-lst helped-by-lst
competitor-lst) ...)  but NOT a mixture of them.

Probably need to use some sort of Bayesian probability for populating
'random' patches, lest we get unlikely species mixes (like camels and
penguins)
```

```
Sticking to the service-update-map is critical for getting the right
answers, y'know.

NOTE: update values are calculated by lambdas which take the set of
values associated with the services present in the patch.  The
nominated services are listed in the service-indices list either
categorically (the types) and strings (the names).  The types are the
aggregate values of the names -- if you want to deal with a named
entity separately, the update equation must explicitly remove it. They
will be of the form (lambda (t ...) ...)  and *all* of the indicated
services must be there or Bad Things Happen.
")

;; species-index is an association list which pairs ecoservice names
;; to indices in the
(register-class <dynamic-patch>)

(define <landscape> (make-class (inherits-from <environment>)
                                (state-variables terrain-function)))
;; terrain-function is a function in x and y that returns a DEM
(register-class <landscape>)

(define <habitat> (make-class (inherits-from <landscape>)
                              (state-variables patch-list dump-times scale)))
;; sites is a list of patches -- the patches can be either patches,
;; dynamic-patches or a mix -- the list is passed in at initialisation.

(register-class <habitat>)


;-  The End
```

```
;;; Local Variables:
;;; mode: scheme
;;; outline-regexp: ";-+"
;;; comment-column:0
;;; comment-start: ";;; "
;;; comment-end:""
;;; End:

;--
;  landscape-declarations.scm -- Written by Randall Gray

;--- <ecoservice> generics (also <patch>)

(declare-method add! "add!")
(declare-method scale! "scale!")
(declare-method enable-growth! "disables the growth model for the ecoservice")
(declare-method disable-growth! "disables the growth model for the ecoservice")
(declare-method growth-model "returns the growth model for the ecoservice")
(declare-method rvalue
                (string-append "returns the r value (sharpness "
                               "or exponent) of an ecoservice's growth model"))

;--- <boundary>, <circle> and <polygon> generics
(declare-method inside?
                "indicates whether a location is within a boundary object")
(declare-method my-rep "returns the underlying agent which defines the domain")
(declare-method distance-to-boundary
                "returns the distance to a boundary")
(declare-method rep "returns the representation object")
```

```
(declare-method centre "returns the centre/centroid of an object")
(declare-method set-centre! "returns the centre/centroid of an object")

;--- <patch> generics

(declare-method install-boundary "set/reset the boundary for a patch")
(declare-method service? "service?")
(declare-method add-service "add-service")
(declare-method remove-service "remove-service")
(declare-method service "service") ;; returns value
(declare-method service-list "service-list") ;; returns value
(declare-method services "services") ;; returns value
(declare-method specific-services "names of species (or services)")
(declare-method service-values
                (string-append "values of species/services (aggregates things "
                               "with the same id)")) ;; returns value
(declare-method set-services! "set-services!") ;; sets value
(declare-method distance-to-centre "distance-to-centre")
(declare-method distance-to-interior "distance-to-interior")
(declare-method radius "radius")
(declare-method set-radius! "sets the radius")
(declare-method capacity "returns the carrying capacity of an ecoservice")
(declare-method total-capacity "returns the total capacity of all the niches")

;--- <dynamic-patch> generics

(declare-method service-list-index
                "returns the index of a species in the species list")
(declare-method service-matrix-index
                "returns the index of a species in the species matrix")
(declare-method set-population-dynamics!
```

```
                (string-append "Sets things up for population dynamics "
                               "using differential equations and rk4*"))
(declare-method define-population-dynamics!
                (string-append "Sets things up for population dynamics using "
                               "definitions, alternative to "
                               "set-population-dynamics!"))
(declare-method growth-model
                (string-append "returns the growth model for the patch (ties "
                               "all the represented populations together)"))
(declare-method enable-growth!
                "enables the system level growth model for the dynamic-patch")
(declare-method disable-growth!
                "disables the system level growth model for the dynamic-patch")
(declare-method enable-service-growth!
                (string-append "enables the service level growth model for a "
                               "service in the dynamic-patch"))
(declare-method disable-service-growth!
                (string-append "disables the service level growth model for "
                               "for a service in the dynamic-patch"))
(declare-method enable-all-service-growth!
                "enables the service level growth model for the dynamic-patch")
(declare-method disable-all-service-growth!
                "disables the service level growth model for the dynamic-patch")



;--- <landscape> generics

;--- <environment> generics
;(declare-method contains? "contains?")
```

```
;(declare-method value "value")
;(declare-method set-value! "set-value!")
;(declare-method min-bound "min-bound")
;(declare-method max-bound "max-bound")
(declare-method mean-value "mean value of a resource or service")

;--- <habitat> (and possibly <gridenv>) generics

(declare-method service-sites "returns the list of patches with particular services")
(declare-method add-patch "add a patch to a habitat")
(declare-method remove-patch "remove a patch from a habitat")
(declare-method patch-list "return a list of all patches or only the ones with particular services")
(declare-method aggregate-value "aggregate-value of a services within a nominated circle")
(declare-method spatial-scale "an indication of some 'natural scale' of the habitat based on the
   dist between patches")

;(declare-method contains? "contains?")  -- defined for environment
;(declare-method value "value")      -- defined for environment
;(declare-method set-value! "set-value!") -- defined for environment
;(declare-method min-bound "min-bound")  -- defined for environment
;(declare-method max-bound "max-bound")  -- defined for environment
;(declare-method services "services") ;; returns patches


;; where there are queries for things like value

(declare-method populate-cells "sets up the cell agents")
(declare-method dimensions "returns the number of dimensions")
(declare-method origin "returns the origin")
(declare-method scale "returns the scale vector or the scale associated with a nominated ordinate")
(declare-method ordinates "returns the indices for a given location")
```

```
(declare-method patch-at "returns a list of any patches containing a supplied location")


;-  The End


;;; Local Variables:
;;; mode: scheme
;;; outline-regexp: ";-+"
;;; comment-column:0
;;; comment-start: ";;; "
;;; comment-end:""
;;; End:

;-  Identification and Changes

;--
;  landscape.scm -- Written by Randall Gray
;  Initial coding:
;     Date: 2012.11.19
;     Location: odin:/home/gray/study/src/new/landscape.scm
;
;  History:
;


;-  Copyright

;
;   (C) 2012 CSIRO Australia
;   All rights reserved
```

```
;

;-  Discussion

;-  Configuration stuff

;-  Included files

(load "postscript.scm")

;-  Variables/constants both public and static

;--    Static data

;--    Public data

;-  Code


;-  Variables/constants both public and static

(define PATCHGREY 0.2)
(define HABITATGREY 0.1)

;- Environmental code

;-- Supporting routines

;--- Generally useful routines
```

```
;(define (I->E f) (inexact->exact (round f)))
(define (I->E f) (inexact->exact (truncate f)))

(define (repro xy res m)
  (map I->E (map (lambda (x) (/ x res)) (map - xy m))))


;;(define (logistic-growth  dt domain 0  value capacity  rvalue)

;(define service? (make-generic))
;(define add-service (make-generic))
;(define remove-service (make-generic))
;(define service-list (make-generic))
;(define service (make-generic))
;(define services (make-generic)) ;; returns value
;(define set-services! (make-generic)) ;; sets value
;(define value (make-generic))      -- defined for environment
;(define set-value! (make-generic)) -- defined for environment

;--- oriented toward habitats, patches  and ecoservices
;---- ecoservice lists, nearest suppliers....

(define (locate-nearest-ecoserv habitat ecoserv loc)
  (let* ((patches (service-sites habitat ecoserv))
         (dists (map (lambda (x) (distance-to-centre x loc)) patches))
         (sdists (sort
                  (filter
                   (lambda (x) (and (number? (car x))
                                    (not (null? (cdr x)))))
                   (map cons dists patches))
                  (lambda (x y) (< (car x) (car y)))
```

```
                ))
        )
    (if (null? sdists) #f (cdar sdists))))

(define (sorted-ecoservices habitat ecoserv loc . weighted-by-value)
  (let* ((patches (service-sites habitat ecoserv)))
    (let ((dists (map (lambda (x) (distance-to-centre x loc)) patches)))
      (let ((sdists (sort
                     (filter
                      (lambda (x) (and (number? (car x))
                                       (not (null? (cdr x)))))
                      (map cons dists patches))

                     (if (null? weighted-by-value)
                         (lambda (x y)
                           (< (car x) (car y)))
                         (lambda (x y)
                           (< (/ (total-value (cdr x) ecoserv) (+ 1 (car x)))
                              (/ (total-value (cdr y) ecoserv) (+ 1 (car y)))
                              ))
                         )
                     )))

        (if (null? sdists)
            #f
            (map (lambda (x) (list (car x) (cdr x))) sdists))))
    ))


;---- Applicable to patches and patch lists
```

```
(define (patchsize domain)
  (* 0.25 (apply min (map - (list-head (cadr domain) 2)
                            (list-head (car domain) 2)))))

(define (total-patch-list-value patchlist symlist)
  (map (lambda (x) (total-value x symlist)) patchlist))


(define (total-patch-list-capacity patchlist symlist)
  (map (lambda (x) (total-capacity x symlist)) patchlist))


(define (translate-trace addend trace)
  (map
   (lambda (v)
     (map + addend v))
   trace))

(define (scale-trace s trace)
  (if (number? s) (set! s (make-list (length (car trace)) s)))
  (if (= (length s) (length (car trace)))
      (map
       (lambda (v)
         (map * s v))
       trace)
      (error "Incompatible vectors" s trace)
      ))
;---- For habitats

(define (def-res H)
  (let* ((m (min-bound H))
```

```
        (M (max-bound H))
        (extent (map - M m))
        )
    (/ (apply min extent) 20.0)))


;---- Generating postscript

(define print-environment-data
  (lambda (ps p x n ns loc rad)
    ;;(dnl (pno (value x)))
    (ps 'moveto (map p (map + (list-head loc 2)
                            (map p (list (* 1.0 rad)
                                         (* (- (/ ns 2.0) n) 1.0))))))
    (if adjust-grey (ps 'setgray PATCHGREY)) ;; zero is white...
    (ps 'Helvetica 7)
    (ps 'show (string-append (slot-ref x 'name) ": "
                             (number->string (value x))))
    ))


(define crop-caption
  (lambda (ps p x . pt)
    (if (null? pt) (set! pt 10))
    (let ((loc (map p (list-head (location x) 2)))
          (rad (p (radius x))))
      (ps 'moveto (map - loc
                       (list (* 0.5 rad)  (* -1 (+ 5 (* 1 rad) )))))
      (ps 'Helvetica pt)
      (if adjust-grey (ps 'setgray PATCHGREY))
      (ps 'show-right (string-append
```

```
                    (slot-ref x 'name) " at "
                    (number->string (slot-ref x 'subjective-time))))
      )
    )
  )

;--- (make-population-structure predation-matrix efficiency-matrix
;                   service-data-list ecoservice-template)

(define (make-population-structure predation-matrix efficiency-matrix
                                   service-data-list ecoservice-template)
  ;;(pp service-data-list)
  ;;(pp ecoservice-template)
;(abort 3)
  (let* ((predation-matrix predation-matrix)
         (PM predation-matrix)
         (EM (efficiency-matrix 'transpose)) ;; orient it for easy predator use
         (ecoservice-template (deep-copy ecoservice-template))
         (service-data-list (deep-copy service-data-list))
         (service-id-list (map (lambda (x) (list-head x 3)) service-data-list))
         (service-name-list (map car service-id-list))
         (service-symbol-list (map string->symbol service-name-list))
         (service-type-list (map cadr service-id-list))
         (service-eqn-sym-list (map caddr service-id-list))

         ;; This is analogous to the "(service-list-index self sym)"
         ;; call in <dynamic-patch>, but it relies on globals
         (service-index (lambda (sym)
                          (let ((m (memq sym service-eqn-sym-list))
                                (n (memq sym service-name-list)))
```

```
                    (if m (- (length service-eqn-sym-list)
                             (length m))
                        (if n (- (length service-eqn-sym-list)
                                 (length n))
                            #f)))))

(pd (lambda (species)
      (let* ((species-list service-eqn-sym-list)
             (as-prey-ratio
              (list-sym-ref ((PM 'transpose))
                            service-eqn-sym-list species))
             (as-predator-ratio
              (list-sym-ref (PM) service-eqn-sym-list species))
             (predator-efficiency
              (list-sym-ref (EM) service-eqn-sym-list species))
             (cap (list-ref
                   (list-ref service-data-list
                             (service-index species)) 3))
             (gr (list-ref
                  (list-ref service-data-list
                            (service-index species)) 4))
             (mort (list-ref
                    (list-ref service-data-list
                              (service-index species)) 4))

             (pop-growth-func (list-ref
                               (list-ref service-data-list
                                         (service-index species)) 6))
             )

        (let ((dP/dt
```

```
                    (lambda (t . populations)
                      (pop-growth-func t species cap gr
                                       as-predator-ratio predator-efficiency
                                       mort as-prey-ratio populations)))
                  )
                dP/dt))))

    (d/dt (map (lambda (species) (pd species))
               (map caddr service-data-list)))
    )
(let ((population-structure
       (lambda args
         (cond
          ((null? args)
           (abort "null  passed as an argument to a population structure"))
          ((eq? (car args) 'template)
           ecoservice-template)
          ((eq? (car args) 'predation-matrix)
           predation-matrix)
          ((eq? (car args) 'efficiency-matrix)
           (EM 'transpose)) ;; send it back in the same form we got it
          ((memq (car args) '(service-data species-data))
           service-data-list)
          ((memq (car args) '(service-ids species-ids))
           service-id-list)
          ((memq (car args) '(service-names species-names))
           service-name-list)
          ((memq (car args) '(service-symbols species-symbols))
           service-symbol-list)
          ((memq (car args) '(service-types species-types))
           service-type-list)
```

```
             ((memq (car args) '(service-eqn-syms species-eqn-syms))
              service-eqn-sym-list)
             ((eq? (car args) 'd/dt-list)
              d/dt)
             ((eq? (car args) 'index)
              (if (null? (cdr args))
                  #f
                  (if (pair? cddr)
                      (map service-index (cdr args))
                      (service-index (cadr args)))))
             ))))
       population-structure)
    ))


;; the predation matrix is oriented so that if we consider
;; grass-cow-leopard across the columns, the subdiagonal will be the
;; one with non-zero entries

;; the efficiency matrix is oriented the same way as the predation
;; matrix (on input).

;; non-sigmoidal-growth must either be absent, #f, growth function
;; with a form
;;
;;    (growth-func t population-level-list)
;;
;; where domain is the "time to cap" or some such thing, where P_0 is
;; the starting value, P is the current value, K is the capacity and r
;; is the "exponent" or a list of such functions (one for each species).
```

```
;; In practice, we probably ought to never get just a function.

;--- (make-population-structure predation-matrix efficiency-matrix
;                               service-data-list ecoservice-template)

;;; For examples look at savannah-parameters....

;-- <environment> methods and bodies

;; Stops things going off the rails

;--- (services...) returns services matching the sym or in the symlist
(default-initialization <environment>)

(sclos-method (<environment>) (services self syms)
              '())

(sclos-model-body <environment>
                  (parent-body)
                  dt)


;-- <ecoservice> methods and bodies

;; By convention we give ecoservices names which are strings, types
;; which are symbols ... neither needs to be unique
;;
;; value, set-value!, add! scale!


(Comment " Ecoservices are able to update their state themselves.  If
```

```
they aren't *nested*, this may have irregular interactions with any
dynamics being forced on them from a dynamic-patch since the order of
insertion in the queue is not prescribed. The best way of dealing with
this situation would be to ensure that the timestep associated with
ecoservices is half (or less) of the timestep of the patch.
")


(default-initialization <ecoservice> 'do-growth #t 'history #f)

(define (simple-ecoservice N n V C r maxdt growing? growthmodel . P)
  (if (pair? P) (set! P (car P)))

  (make <ecoservice>
    'name N
    'sym n
    'patch P
    'value V
    'capacity C
    'r r
    'delta-T-max maxdt
    'do-growth growing?
    'growth-model growthmodel)
         ;;; 'sigmoid
         ;;; 'linear
         ;;; (lambda (t dt v) ...)
      ;;
)


(sclos-method <ecoservice> (dump self . count)
```

```
            (set! count (if (null? count) 0 (car count)))
            (display (make-string count #\space))
            (display "<ecoservice>\n")

            (let* ((slots (map car (class-slots (class-of self))))
                   (vals  (map (lambda (x) (slot-ref self x)) slots)))
              (for-each (lambda (x y)
                          (display (make-string (+ 2 count) #\space))
                          (display x)
                          (display ": ")
                          (display y)
                          (newline))
                        slots vals)))


;---- query & set
(sclos-method (<ecoservice> <symbol>) (service? self sym)
              (or (eq? (my 'type) sym) (eq? (my 'name) sym)))

(sclos-method (<ecoservice> <pair>) (service? self symlist)
              (or (memq (my 'type) symlist) (memq (my 'name) symlist)))

(sclos-method <ecoservice> (value self)
              (my 'value))

(sclos-method <ecoservice> (capacity self)
              (my 'capacity))

(sclos-method (<ecoservice>) (set-value! self val)
              ;;(dnl* "Setting" (name self) "from" (my 'value) "to" val)
              (set-my! 'value val))
```

```
(sclos-method <ecoservice> (growth-model self)
              (my 'growth-model))

;--- adjustment

(sclos-method (<ecoservice>) (disable-growth! self) (set-my! 'do-growth #f))
(sclos-method (<ecoservice>) (enable-growth! self) (set-my! 'do-growth #t))

(sclos-method (<ecoservice>) (add! self val)
              (let ((v (my 'value)))
                (if (number? v)
                    (set-my! 'value (+ v val) )
                    (aborts "ecoservice:add!: value is not a number")
                    )))

(sclos-method (<ecoservice>) (scale! self val)
              (let ((v (my 'value)))
                (if (number? v)
                    (set-my! 'value (* v val) )
                    (aborts "ecoservice:scale!: value is not a number")
                    )))


;;--- ecoservice model-body support routines (mainly about growth)


(sclos-model-body <ecoservice>

                  (kdnl* "[" (my 'name) ":" (class-name-of self) "]"
                         'model-bodies "In " (class-name-of self) t)
```

```
;;;(dnl* 'b1)
(let ((h (slot-ref self 'history)))
  (if h
      (slot-set! self 'history
                 (cons (cons t (my 'value)) h)))
  )
;;;(dnl* 'b2)


(if (my 'do-growth) ;; may be suppressed in
                    ;; dynamic-patches, for example
    (begin
      ;;;(dnl "Running <ecoservice> model body for "
      ;;;     (my 'name))
      (let* ((capacity (my 'capacity))
             (value  (my 'value))
             (domain (my 'delta-T-max))
             (ecoserv-growth (my 'growth-model))
             ;; The growth-model expects the start
             ;; of the time-step and and an
             ;; interval. Some services should use
             ;; the current _value_ to calculate a
             ;; putative "time" from it's zero point
             ;; and then generate the value for
             ;; t+dt. Others may have more
             ;; straightforward ways of calculating
             ;; the value at the end of the timestep
             ;; (such as a table, or a regular
             ;; increment, for example).
             (newvalue
```

```
                    (cond
                     ((procedure? ecoserv-growth)
                      ;;;(dnl 'c1)
                      (growth-model t dt value))
                     ((eq? ecoserv-growth 'sigmoid)
                      ;;;(dnl 'c2)
                      (let* (;; This is a logistic update
                             (ipt (/ value capacity))
                             (pt (inverse-sigmoid* ipt))
                             (rdt (/ dt domain))
                             )
                        (* capacity (sigmoid* (+ pt rdt)))))
                     ((eq? ecoserv-growth 'linear)
                      ;;;(dnl 'c3)
                      (min capacity (+ value (* dt (my 'r)))))
                     (#t
                      ;;;(dnl 'c4)
                      value)))
                   )
                 ;;(dnl* (name (my 'patch)) "/" (my 'name)
                 ;;      "value =" value "| newvalue =" newvalue)

                 (set-my! 'value  newvalue)
                 )
               ))
         (parent-body)
         dt
         )


(sclos-method <ecoservice> (radius self)
```

```
              (radius (my 'patch)))

(sclos-method (<ecoservice> <number>)(set-radius! self r)
              (set-radius! (my 'patch) r))

(sclos-method <ecoservice> (location self)
              (location (my 'patch)))

(sclos-method <ecoservice>
              (log-data self logger format caller targets . args)
              (let ((f (if (pair? args) (car args) #f))
                    (p (if (and (pair? args)
                                (pair? (cdr args)))
                           (cadr args)
                           #f)))
                (let ((file (slot-ref logger 'file)))
                  (kdnl* '(log-* log-ecoservice)
                         "[" (my 'name) ":" (class-name-of self) "]"
                         "in log-data")
                  (let ((leading-entry #f))
                    (for-each
                     (lambda (field)
                       (kdnl* '(log-* log-ecoservice) "[" (my 'name) ":"
                              (class-name-of self) "]" "checking" field)
                       (if (has-slot? self field)
                           (let ((r (slot-ref self field)))
                             (case format
                               ((ps)
                                (file 'show (string-append
                                             (if (string? r)
                                                 r
```

```
                                (object->string r)) " "))
                )
;               ((dump)
;                (with-output-to-port file
;                    (lambda ()
;                      (dump self))))

                 ((text table dump)
                  (let ((show-field-name
                         (slot-ref logger 'show-field-name))
                        (missing-val
                         (slot-ref logger 'missing-val))
                        )
                    (if show-field-name
                        (begin
                          (if leading-entry
                              (display " " file)
                              (set! leading-entry #t))
                          (display field file)))

                    (let ((val (if (eq? field 'name)
                                   (if (slot-ref self 'patch)
                                       (string-append
                                        (slot-ref
                                         (slot-ref self 'patch)
                                         'name) ":" (name self))
                                       (name self))
                                   (if (has-slot? self field)
                                       (slot-ref self field)
                                       (slot-ref logger
                                                 'missing-val)))))
```

```
                            (if leading-entry
                                (display " " file)
                                (set! leading-entry #t))
                            (display val file))
                          )
                        )

                        (else
                         (kdnl* '(log-* log-ecoservice)
                                "[" (my 'name) ":" (class-name-of self) "]"
                                "Ignoring " field " because I don't have it")
                         'ignore-unhandled-format)))
                    (begin
                      (kdnl* '(log-* log-ecoservice)
                             "[" (my 'name) ":" (class-name-of self) "]"
                             "no service" field)
                      #f)))
                (uniq (if #t
                          targets
                          (filter (not-memq (slot-ref logger 'dont-log))
                                  targets)))
                )
               (newline file)
               )
             ))
          )


;--- <circle>
(default-object-initialization <circle>)
```

```
(define (make-circle centre radius)
  (make <circle> 'locus centre 'radius radius))

(object-method (<circle>) (dump self . count)
               (set! count (if (null? count) 0 (car count)))

               (display (make-string count #\space))
               (display "<circle>\n")
               (let* ((slots (map car (class-slots (class-of self))))
                      (vals  (map (lambda (x) (slot-ref self x)) slots)))
                 (for-each (lambda (x y)
                             (begin
                               (display (make-string (+ 2 count) #\space))
                               (display x)
                               (display ": ")
                               (display y)
                               (newline)))
                 slots vals))
               )


(object-method (<circle> <list>) (inside? self loc)
               (<= (distance (my 'locus) loc) (my 'radius)))

(object-method (<circle>) (centre self)
               (my 'locus))

(object-method (<circle>) (radius self)
               (my 'radius))

(object-method (<circle>) (min-bound self)
```

```
              (my 'radius))

(object-method (<circle>) (max-bound self)
              (my 'radius))

(object-method (<circle> <list>) (distance-to-boundary self loc)
              (if (inside? self loc)
                  0
                  (- (distance loc (my 'locus)) (my 'radius))))

;--- <polygon>
(default-object-initialization <polygon>)

(define (make-polygon centre polygon)
  (if (not (eq? (car polygon) (car (reverse polygon))))
      (set! polygon (append polygon (list (car polygon)))))
  (make <polygon> 'locus centre 'perimeter (copy-list polygon)))

(object-method (<polygon>) (dump self . count)
              (set! count (if (null? count) 0 (car count)))

              (display (make-string count #\space))
              (display "<polygon>\n")
              (let* ((slots (map car (class-slots (class-of self))))
                     (vals  (map (lambda (x) (slot-ref self x)) slots)))
                (for-each (lambda (x y)
                            (begin
                              (display (make-string (+ 2 count) #\space))
                              (display x)
                              (display ": ")
                              (display y)
```

```
                          (newline)))
               slots vals))
             )
(object-method (<polygon> <list>) (inside? self loc)
             (point-in-polygon loc (my 'perimeter)))


(object-method (<polygon>) (radius self)
             (max-bound self)
             )

(object-method (<polygon> <list>) (distance-to-boundary self loc)
             (if (inside? self loc)
                 0
                 (distance-to-boundary loc (my 'perimeter))))

(object-method (<polygon>) (min-bound self)
             (let ((c (my 'locus))
                   (p (my 'polygon)))

               (let loop ((r (distance c (car p)))
                          (l (cdr p)))
                 (if (null? l)
                     r
                     (loop ((min r (distance c (car p)))
                            (cdr p)))))))

(object-method (<polygon>) (max-bound self)
             (let ((c (my 'locus))
                   (p (my 'polygon)))
```

```
                (let loop ((r (distance c (car p)))
                           (l (cdr p)))
                  (if (null? l)
                      r
                      (loop ((max r (distance c (car p)))
                             (cdr p)))))))


;-- <boundary> -> <circle>, <polygon>

;-- <boundary> methods and bodies

(default-object-initialization <boundary> 'rep #f)

(define (make-boundary rep centre arg)
  (let ((M (make <boundary>)))
    (case rep
      ((circle)
       (slot-set! M 'rep (make <circle> 'locus centre 'radius arg))
       )

      ((polygon absolute-polygon)
       (slot-set! M 'rep (make <polygon> 'locus centre
                               'perimeter (copy-list arg)))
       )

      ((relative-polygon)
       (slot-set! M 'rep (make <polygon> 'locus centre
                               'perimeter (translate-trace centre arg)))
       )
```

```
      (else (error "Bad representation specified for a <boundary>" rep)))))

(object-method (<boundary>) (dump self . count)
               (set! count (if (null? count) 0 (car count)))

               (display (make-string count #\space))
               (display "<boundary>\n")
               (let* ((slots (map car (class-slots (class-of self))))
                      (vals  (map (lambda (x) (slot-ref self x)) slots)))
                 (for-each (lambda (x y)
                             (begin
                               (display (make-string (+ 2 count) #\space))
                               (display x)
                               (display ": ")
                               (display y)
                               (newline)))
                 slots vals))
               (display (make-string (+ 2 count) #\space))
               (display "rep:\n")
               (dump (my 'rep))
               )

(object-method (<boundary>) (rep self)
               (my 'rep))

(object-method (<boundary> <list>) (inside? self loc)
               (inside? (my 'rep) loc))

;;
;; min-bound, max-bound contains? services
```

```
(object-method (<boundary>) (centre self)
               (centre (my 'rep) ))

(object-method (<boundary> <list>) (set-centre! self c)
               (slot-set! (my 'rep) c))

(object-method (<boundary>) (min-bound self)
               (min-bound (my 'rep) ))

(object-method (<boundary>) (max-bound self)
               (max-bound (my 'rep) ))

(object-method (<boundary> <list>) (distance-to-boundary self loc)
               (distance-to-boundary (my 'rep) loc))

;-- <patch> methods and bodies
;;
;; min-bound, max-bound contains? services



;--- (initialize...)
(default-initialization <patch>)

(sclos-method <patch> (dump self . count)
               (set! count (if (null? count) 0 (car count)))

               (display (make-string count #\space))
               (display "<patch>\n")
               (let* ((slots (map car (class-slots (class-of self))))
                      (vals  (map (lambda (x) (slot-ref self x)) slots)))
```

```
                (for-each (lambda (x y)
                            (if (not (memq x '(boundary service-list)))
                                (begin
                                  (display (make-string (+ 2 count) #\space))
                                  (display x)
                                  (display ": ")
                                  (display y)
                                  (newline))))
                          slots vals))
              (display (make-string (+ 2 count) #\space))
              (display "boundary and service lists:\n")
              (for-each (lambda (x) (dump x (+ 4 count)))
                        (cons (my 'rep) (my 'service-list)))
              )

(sclos-method (<patch> <boundary> <list>) (install-boundary self bdry centre)
              (set-my! 'rep bdry)
              (slot-set! bdry 'locus centre)
              )

(sclos-method <patch> (location self)
              (slot-ref (my 'rep) 'locus))

;--- (service?...) queries if a service is present
(sclos-method (<patch> <symbol>) (service? self sym)
              (not (null? (services self sym))))

(sclos-method (<patch> <list>) (service? self symlist)
              (not (null? (services self symlist))))
```

```
;--- (set-services!...) sets the value of the services list
(sclos-method (<patch> <list>) (set-services! self servlist)
              (set-my! 'service-list servlist))


;--- (add-service...) adds a service to a patch

(sclos-method (<patch> <ecoservice>) (add-service self new-service)
              (set-services! self (append (my 'service-list)
                                          (list new-service))))


;--- (remove-service...) removes all services that match the predicate
;                        in a patch
;;                       the predicate will probably be something like
;;                       (using-name-keep? 'wobble)
(sclos-method (<patch> <procedure>) (remove-service self predicate)
              (set-services! self (filter predicate (my 'service-list))))



;--- (distance-to-centre...) returns the distance to the centre of the patch
(sclos-method (<patch> <list>) (distance-to-centre self loc)
              (let ((sqr (lambda (x) (* x x))))
                (sqrt (apply + (map sqr (map - (list-head
                                                (slot-ref (my 'rep) 'locus) 2)
                                               (list-head loc 2)))))))

;--- (distance-to-interior...) returns the distance to the boundary of
;the patch (more expensive than the dist to centre)
(sclos-method (<patch> <list>) (distance-to-interior self loc)
```

```
              (let* ((sqr (lambda (x) (* x x)))
                     (R (- (sqrt (apply
                                  + (map sqr
                                         (map -
                                              (list-head
                                               (slot-ref (my 'rep) 'locus) 2)
                                              (list-head loc 2)))))
                           (my 'radius)))
                     )
                (if (< R 0) 0 R)))

;--- (contains?...) predicate to indicate if something is in the patch
(sclos-method <patch> (contains? self . bit)
              (if (null? bit)
                  (abort "Missing argument to contains?")
                  (set! bit (car bit)))
              (cond
               ((pair? bit)
                (let ((R (distance-to-interior self bit)))
                  (zero? R)))
               ((isa? bit <thing>)
                (let ((R (distance-to-interior self (location bit))))
                  (zero? R)))
               (else #f)))

;;(sclos-method (<patch> <thing>) (contains? self entity)
;;              (contains? (location entity))
;;              )


;--- (services...) returns services matching the sym or in the symlist
```

```
(sclos-method (<patch>) (service-list self . ss)
              (if (and (pair? ss) (pair? (car ss))) (set! ss (car ss)))
              (let ((S (my 'service-list)))
                (if (null? ss)
                    S
                    (filter
                     (lambda (x) (or (member (type x) ss) (member (name x) ss)))
                     S))))


(sclos-method (<patch>) (services self . ss)
              (if (null? ss)
                  (map (lambda (x) (type x)) (my 'service-list))
                  (map type (apply service-list (cons self ss)))))


(sclos-method (<patch>) (specific-services self . ss)
              (if (null? ss)
                  (map (lambda (x) (name x)) (my 'service-list))
                  (map type (apply service-list (cons self ss)))))


(sclos-method (<patch> <symbol>) (value self servlist)
              (set! servlist (list servlist))
              (let ((sl (if (member #t servlist)
                            (my 'service-list)
                            (service-list self servlist))))
                (if (null? sl)
                    0
                    (apply + (map value sl)))))
```

```
(sclos-method (<patch> <string>) (value self servlist)
              (set! servlist (list servlist))
              (let ((sl (if (member #t servlist)
                            (my 'service-list)
                            (service-list self servlist))))
                (if (null? sl)
                    0
                    (apply + (map value sl)))))

(sclos-method (<patch> <symbol>)(extra-variable self field)
              (value self (symbol->string field)))

(sclos-method (<patch> <string>)(extra-variable self field)
              (value self field))

(sclos-method (<patch>) (extra-variable-list self)
              (map string->symbol (map name (my 'service-list))))

;; (add-method representation
;;              (make-method (list <agent>)
;;                           (lambda (representation-parent self)
;;                             (my 'representation))))



(sclos-method (<patch> <pair>) (value self servlist)
              (let ((sl (if (member #t servlist)
                            (my 'service-list)
                            (service-list self servlist))))
                (if (null? sl)
```

```
                    0
                    (apply + (map value sl)))))

(sclos-method (<patch> <pair>) (capacity self servlist)
              (let ((sl (if (member #t servlist)
                            (my 'service-list)
                            (service-list self servlist))))
                (if (null? sl)
                    0
                    (apply + (map capacity sl)))))

(sclos-method (<patch> <pair>) (mean-value self servlist)
              (let ((sl (service-list self servlist)))
                (if (null? sl)
                    0
                    (/ (apply + (map value sl))
                       (* 1.0 (length servlist))))))

;;;--- (set-value!...)
(sclos-method (<patch> <symbol>) (set-value! self sym val)
              (let ((s (filter (lambda (a) (eq? sym (type a)))
                               (my 'service-list))))
                (if (null? s)
                    #f
                    (begin
                      (for-each (lambda (x) (set-value! x val)) s)
                      #t))))

(sclos-method (<patch> <string>) (set-value! self sym val)
              (let ((s (filter (lambda (a) (string=? sym  (name a)))
                               (my 'service-list))))
```

```
                (if (null? s)
                    #f
                    (begin
                      (for-each (lambda (x) (set-value! x val)) s)
                      #t))))

;--- (scale!...)
(sclos-method (<patch> <symbol> <number>) (scale! self sym val)
              (let ((s (filter (lambda (a) (eq? (type a) sym))
                               (my 'service-list))))
                (if (null? s)
                    #f
                    (begin
                      (for-each (lambda (x) (scale! x val)) s)
                      #t))))

(sclos-method (<patch> <string> <number>) (scale! self sym val)
              (let ((s (filter (lambda (a) (string=? (name a) sym))
                               (my 'service-list))))
                (if (null? s)
                    #f
                    (begin
                      (for-each (lambda (x) (scale! x val)) s)
                      #t))))

;--- (add!...)
(sclos-method (<patch> <symbol> <number>) (add! self sym val)
              (let ((s (filter (lambda (a) (eq? (type a) sym))
                               (my 'service-list))))
                (if (null? s)
                    #f
```

```
                    (begin
                      (for-each (lambda (x) (add! x val)) s)
                      #t))))

(sclos-method (<patch> <string> <number>) (add! self sym val)
              (let ((s (filter (lambda (a) (string=? (name a) sym))
                               (my 'service-list))))
                (if (null? s)
                    #f
                    (begin
                      (for-each (lambda (x) (add! x val)) s)
                      #t))))

;--- (scale!...)
(sclos-method (<patch> <pair> <number>) (scale! self symlist val)
              (for-each (lambda (x) (scale! self x val)) symlist))


;--- (add!...)
(sclos-method (<patch> <pair> <number>) (add! self sym val)
              (for-each (lambda (x) (add! self x val)) symlist))


;--- (total-value ...) ;; needs to filter the services by membership
;in the indicated symlist
(sclos-method (<patch> <pair>) (total-value self symlist)
              (let ((ss (service-list self (if (symbol? symlist)
                                               (list symlist)
                                               symlist))))
                (if (or (not ss) (null? ss) )
                    0.0
```

```
                    (apply + (map (lambda (y) (value y)) ss)))))

(sclos-method (<patch> <pair>) (total-value self symlist)
              (let ((ss (service-list self (if (symbol? symlist)
                                               (list symlist)
                                               symlist))))
                (if (or (not ss) (null? ss) )
                    0.0
                    (apply + (map (lambda (y) (value y)) ss)))))

;--- sclos-method (<patch> <pair>) (total-capacity self symlist)
(sclos-method (<patch> <pair>) (total-capacity self symlist)
              (let ((ss (service-list self (if (symbol? symlist)
                                               (list symlist)
                                               symlist))))
                (if (or (not ss) (null? ss) )
                    0.0
                    (apply + (map (lambda (y) (capacity y)) ss)))))

(sclos-method (<patch> <pair>) (total-capacity self symlist)
              (let ((ss (service-list self (if (symbol? symlist)
                                               (list symlist)
                                               symlist))))
                (if (or (not ss) (null? ss) )
                    0.0
                    (apply + (map (lambda (y) (capacity y)) ss)))))

;--- sclos-model-body <patch>
(sclos-model-body <patch>
                  (kdnl* 'model-bodies "In " (class-name-of self)
                         (name self) "@" t)
```

```
                ;; this does the growth  and endemic mortality
                (if (member 'nested-habitat nested-agents)
                    (for-each (lambda (x)
                                (run x t (+ t dt) (my 'kernel))
                                )
                              (service-list self)))
                (kdnl* 'nested-habitat (name self) "@" t "/"
                       (subjective-time self) ":" dt "/" (my 'dt))
                (parent-body)
                (kdnl* 'nested-habitat (name self) "@" t "/"
                       (subjective-time self) ":" dt "/" (my 'dt))
                dt
                )

;--- sclos-method (<patch> <agent> <symbol> <agent>) (log-data self logger format caller targets)
(sclos-method (<patch> <agent> <symbol> <agent>)
              (log-data self logger format caller targets)
              (let ((file (slot-ref logger 'file))
                    (p (slot-ref self 'map-projection)))
                (if (or (not p) (null? p)) (set! p (lambda (x) x)))
                (kdnl* '(log-* log-patch) "[" (my 'name) ":"
                       (class-name-of self) "]" "in log-data")

                (case format
                  ((ps)
                   (let* ((symlist (services h))
                          (name (slot-ref h 'name))
                          )

                     (if adjust-grey (file 'setgray patchgrey))
```

```
  (ps-circle file  (p (radius self))
             (p (list-head (location self) 2)) 0.7 0.0)


  (let* ((slist (slot-ref self 'service-list))
         (n (+ 1 (length slist))) ;; slist is
                                  ;; becoming
                                  ;; circular??
                                  ;; ....******
         (ns (length slist))
         (loc (location self))
         (rad (radius self))
         (mm-xoffset 2)
         (mm-yoffset 2)
         )
    (file 'moveto (map p (map +
                               (list-head loc 2)
                               (list (+ mm-xoffset
                                        (* 1.05 rad))
                                     (+ mm-yoffset
                                        (/ ns 2.0)))
                               )))
    (file 'show-table (map
                       (lambda (x) (string-append
                                    (slot-ref x 'name)
                                    ": " (pno (value x))))
                       slist))
    )
  (crop-caption file p self)
  )
)
```

```
                ;;((text table dump)
                ;; (log-data-parent)
                ;; )
                (else
                 (display (my 'name) file)
                 (for-each
                  (lambda (x)
                    (display " " file)
                    (display (value x)))
                  (map (lambda (x) (value x)) (my 'service-list))
                  (newline file))
                 ;;(log-data-parent)
                 )
                )
               )
              )


;-- dynamic-patch methods and body

;--- <dynamic-patch> (initialize  (initialize-parent self args)
(add-method initialize
            (make-method (list <dynamic-patch>)
                         (lambda (initialize-parent self args)
                           (initialise self (list 'population-names '()
                                                  'population-symbols '()
                                                  'd/dt-list '()
                                                  'do-dynamics #t
                                                  'population-definitions '()
                                                  'subdivisions 12
```

```
                    ;; 'cause 12 is a nice number?
                    ))
(let ((population-definitions
       (slot-ref self 'population-definitions))
      (population-names
       (slot-ref self 'population-names))
      (population-symbols
       (slot-ref self 'population-symbols))
      (d/dt-list (slot-ref self 'd/dt-list)))

  (if (or (pair? population-definitions)
          (pair? population-names)
          (pair? population-symbols)
          (pair? d/dt-list))
      (begin
        (if (and (pair? population-definitions)
                 (or (pair? d/dt-list)
                     (pair? population-names)
                     (pair? population-symbols)))
            (abort (string-append
                    "Dynamic patch specified a "
                    "population-definition *and* "
                    "one or more of\n"
                    "population-names "
                    "population-symbols "
                    "d/dt-list\n")))

        (if (and (null? population-definitions)
                 (not (and (pair? d/dt-list)
                           (pair? population-names)
                           (pair? population-symbols))))
```

```
            (abort (string-append
                    "Dynamic patch specified at "
                    "least one of population-names "
                    "population-symbols d/dt-list\n"
                    "but not all three."
                    )))

        ))
  (cond
   ((pair? population-definitions)
    (define-population-dynamics! self
      population-definitions))
   ((pair? population-names)
    (define-population-dynamics! self
      population-names
      population-symbols
      d/dt-list))
   (#t (slot-set! self 'population-definitions #f)
       (slot-set! self 'population-names #f)
       (slot-set! self 'population-symbols #f)
       (slot-set! self 'd/dt-list #f)))
  )
(initialize-parent) ;; call "parents" last
                    ;; to make the
                    ;; initialisation list
                    ;; work
)))
```

```
(define (bbox ll ur)
  (list ll (list (car ur) (cadr ll)) ur (list (car ll) (cadr ur)) ll))

(define unitbox '((0 0) (1 0) (1 1) (0 1) (0 0)))

(define (%patch-initialiser clss bdry name type centre radius box . therest)
  (if (memq clss (list <patch> <dynamic-patch>))
      (append (list clss)
              (cond
               ((eq? bdry <circle>)
                (list 'rep (make <circle> 'name name 'type type '
                                 locus centre 'radius radius)
                      ))
               ((eq? bdry <polygon>)
                (list 'rep (make <polygon> 'name name 'type type
                                 'locus centre 'point-list (copy-list box))
                      ))
               (#t
                (error "bad boundary class specified in patch-initialiser"
                       bdry)))
              therest)
      (error (string-append "patches may only be initialised as <patch> or "
             "<dynamic-patch> agents")
             clss)
      ))


;; make-grid is geared to making patches and dynamic-patches in a regular array
;; n m is the size of the grid, ll up are the ordinates of the ll cell and the ur cell
;; bdry should be <circle> or <polygon>, clss should be <patch> or <dynamic-patch>,
```

```
;; clss-initialiser is a function that returns a fully formed initialiser list
;; (apart from the centre and radius or perimeter), and P should be either null
;; or a <habitat> like class.
(define (make-grid n m ll ur clss bdry name type clss-initialiser . therest)
  (let* ((nscale (real->integer (/ (- (car ur) (car ll)) (* 1.0 n))))
         (mscale (real->integer (/ (- (cadr ur) (cadr ll)) (* 1.0 n))))
         (radius (min nscale mscale))
         (M (make-list* n m))
         (patchlist '())
         (init-class (lambda  x
                       (apply clss-initialiser (append (list clss bdry) x))))
         )
    (map-** (lambda (x i)
              (let ((centre (list (+ (car ll) (* nscale (+ 0.5 (car i))))
                                  (+ (cadr ll) (* mscale (+ 0.5 (cadr i))))))
                    (box (bbox (list (+ (car ll) (* nscale (car i)))
                                     (+ (cadr ll) (* mscale (cadr i))))
                               (list (+ (car ll) (* nscale (+ 1 (car i))))
                                     (+ (cadr ll) (* mscale (+ 1 (cadr i)))))))
                    (pname (string-append name "-" (number->string (car i)) ","
                                          (number->string (cadr i))))
                    )
                (let ((np (apply make (apply init-class
                                             (append (list pname type centre
                                                           radius box)
                                                     therest)))))
                  (slot-set! np 'name pname)
                  np)
                ))
            M)))
```

```
;; the services are defined by a list containing a name, a symbol, an
;; initial value, a capacity, its max dt, whether or not it grows, and
;; a growth model
(define (populate-patch p services . therest)
  (for-each (lambda (x)
              (cond
               ((agent? x) (add-service p x))
               ((list? x)  (add-service p (apply make (append (list <ecoservice>)
                                                              x therest))))
               (#t (error (string-append "populate-patch should be a list of "
                                         "patches, arguments  to "
                                         "(make <ecoservice> ....) or both"
                                         services))))
            services))


;;(define (patchsize domain) (* 0.25 (apply min (map - (list-head
;;  (cadr domain) 2) (list-head (car domain) 2)))))



;--- (make-patch services centre representation repspec . args)
(Comment "services should be a list of the form '((name type capacity) ...)
args can be  an update map or an update map and update equations
   -- see old-model-version1/Model-configuration.scm
")

;--- sclos-method (<dynamic-patch> <string>) (service-list-index self service)
(sclos-method (<dynamic-patch> <string>) (service-list-index self service)
              (let* ((si (my 'population-names))
```

```
                    (n (length si))
                    (ix (member service si))
                    (i (if ix  (- n (length ix)) #f)))
               i))

;--- sclos-method (<dynamic-patch> <symbol>) (service-list-index self service)
(sclos-method (<dynamic-patch> <symbol>) (service-list-index self service)
              (let* ((si (my 'population-symbols))
                    (n (length si))
                    (ix (member service si))
                    (i (if ix  (- n (length ix)) #f)))
               i))

;--- sclos-method (<dynamic-patch> <pair>) (service-list-index self service)
(sclos-method (<dynamic-patch> <pair>) (service-list-index self service)
              (map (lambda (x) (service-list-index self x)) service))


;; for predation matrix stuff ...
;--- sclos-method (<dynamic-patch> <symbol>) (service-matrix-index self service)
(sclos-method (<dynamic-patch> <symbol>) (service-matrix-index self service)
              (let ((si (service-list-index self service)))
                (if si (+ 1 si) si)))

;--- sclos-method (<dynamic-patch> <pair>) (service-matrix-index self service)
(sclos-method (<dynamic-patch> <pair>) (service-matrix-index self service)
              (map (lambda (x) (service-matrix-index x)) service))

;--- sclos-method (<dynamic-patch> <pair>) (service-values self)
(sclos-method (<dynamic-patch> <pair>) (service-values self)
              (map (lambda (x) (value self x)) (my 'service-update-map)))
```

```
;--- sclos-method <dynamic-patch> (dump self . count)
(sclos-method <dynamic-patch> (dump self . count)
              (set! count (if (null? count) 0 (car count)))

              (display (make-string count #\space))
              (display "<dynamic-patch>\n")
              (let* ((slots (map car (class-slots (class-of self))))
                     (vals  (map (lambda (x) (slot-ref self x)) slots)))
                (for-each (lambda (x y)
                            (if (not (eq? x 'service-list))
                                (begin
                                  (display (make-string (+ 2 count) #\space))
                                  (display x)
                                  (display ": ")
                                  (display y)
                                  (newline))))
                          slots vals))
              (display (make-string (+ 2 count) #\space))
              (display 'service-list)
              (display ":\n")
              (for-each (lambda (x) (dump x (+ 4 count))) (my 'service-list))
              )

;--- sclos-method (<dynamic-patch> <procedure> <symbol>
;                          <procedure>)(log-data self logger format caller targets)
(sclos-method (<dynamic-patch>  <procedure> <symbol> <procedure>)
              (log-data self logger format caller targets)
              (let ((file (slot-ref logger 'file))
                    (p (slot-ref self 'map-projection)))
                (if (or (not p) (null? p))  (set! p (lambda (x) x)))
```

```
(kdnl* '(log-* log-patch) "[" (my 'name) ":"
       (class-name-of self) "]" "in log-data")

(case format
  ((ps)
   (let* ((symlist (services h))
          (name (slot-ref h 'name))
          )

     (if adjust-grey (file 'setgray patchgrey))
     (ps-circle file  (p (radius self))
                (p (list-head (location self) 2)) 0.7 0.0)


     (let* ((slist (slot-ref self 'service-list))
            (n (+ 1 (length slist))) ;; slist is becoming circular?? ....******
            (ns (length slist))
            (loc (location self))
            (rad (radius self))
            (mm-xoffset 2)
            (mm-yoffset 2)
            )
       (file 'moveto (map p (map +
                                 (list-head loc 2)
                                 (list (+ mm-xoffset (* 1.05 rad))
                                       (+ mm-yoffset (/ ns 2.0)))
                                 )))
       (file 'show-table
             (map
              (lambda (x) (string-append
                           (slot-ref x 'name)
```

```
                                    ": "
                                    (pno (value x))))
                            slist))
                      )
                    (crop-caption file p self)
                    )
                  )

                 ;;((text table dump)
                 ;; (log-data-parent)
                 ;; )
                 (else
                  (log-data-parent))
                 )
               )
             )

;--- sclos-method <dynamic-patch> (enable-service-growth! self service-name)
(sclos-method <dynamic-patch> (enable-service-growth! self service-name)
              (enable-growth! (service self service-name)))

;--- sclos-method <dynamic-patch> (disable-service-growth! self)
(sclos-method <dynamic-patch> (disable-service-growth! self)
              (disable-growth! (service self service-name)))


;--- sclos-method <dynamic-patch> (enable-all-service-growth! self)
(sclos-method <dynamic-patch> (enable-all-service-growth! self)
              (for-each enable-growth! (service-list self)))

;--- sclos-method <dynamic-patch> (disable-all-service-growth! self)
```

```
(sclos-method <dynamic-patch> (disable-all-service-growth! self)
              (for-each disable-growth! (service-list self)))

;--- sclos-method <dynamic-patch> (enable-growth! self)
(sclos-method <dynamic-patch> (enable-growth! self) (set-my! 'do-dynamics #t))
;--- sclos-method <dynamic-patch> (disable-growth! self)
(sclos-method <dynamic-patch> (disable-growth! self) (set-my! 'do-dynamics #f))

;--- sclos-method <dynamic-patch> (growth-model self)
(sclos-method <dynamic-patch> (growth-model self) (slot-ref self 'd/dt-list))


;; this expects a list of functions which return reals
;--- sclos-method <dynamic-patch> (set-population-dynamics! self . d/dt-list)
(sclos-method <dynamic-patch> (set-population-dynamics! self . d/dt-list)
              (slot-set! self 'population-definitions #f)

              (if (null? d/dt-list)
                  (abort!))

              (if (pair? (car d/dt-list))
                  (set! d/dt-list (car d/dt-list))) ;; a *list* of
                                                    ;; functions was
                                                    ;; passed in the
                                                    ;; "rest" part of
                                                    ;; the line

              (if (andf (map procedure? d/dt-list))
                  (slot-set! self 'd/dt-list d/dt-list)
                  (abort!))
              )
```

```
;--- sclos-method (<dynamic-patch>) (define-population-dynamics! self pn ps pf)
(sclos-method (<dynamic-patch>) (define-population-dynamics! self pn ps pf)
              (if (not (and (list? pn) (list? ps) (list? pf) (= (length pn)
                                                                (length ps)
                                                                (length pf))))
                  (abort (string-append "define-population-dynamics! was passed"
                                        " lists of differing lenghth!")))
              (let ((tpn (apply andf (map string? pn)))
                    (tps (apply andf (map symbol? ps)))
                    (tpf (apply andf (map procedure? pf))))
                (if (not (and  tpn tps tpf))
                    (begin
                      (display (string-append "There was at least one erroneous "
                                              "argument passed to define-population-dynamics!\n"))
                      (if (not tpn)
                          (let ((culprits (!filter string? pn)))
                            (dnl "The following members of the name list should be strings:")
                            (apply dnl* (cons "   " culprits)))
                          )
                      (if (not tps)
                          (let ((culprits (!filter symbol? ps)))
                            (dnl "The following members of the symbol list should be symbols:")
                            (apply dnl* (cons "   " culprits)))
                          )
                      (if (not tpf)
                          (let ((culprits (!filter procedure? pf)))
                            (dnl "The following members of the d/dt list should be functions:")
                            (apply dnl* (cons "   " culprits)))
                          )
                      )))
```

```
              (slot-set! self 'population-names pn)
              (slot-set! self 'population-names ps)
              (slot-set! self 'population-names pf)

              (slot-set! self 'population-definitions (map list n pn ps pf))

              )



;--- sclos-method (<dynamic-patch>) (define-population-dynamics! self . defns )
(sclos-method (<dynamic-patch>) (define-population-dynamics! self . defns )
              (if (and (pair? defns)
                       (pair? (car defns))
                       (pair? (caar defns))
                       (null? (cdr defns))) ;; passed a list
                  (set! defns (car defns)))


              (if (and (apply andf (map (lambda (x) (= (length x) 3))
                                        defns))
                       (>= (length defns) 1)
                       )
                  (begin
                    (slot-set! self 'population-definitions defns)

                    (let ((pn (map car defns))
                          (ps (map cadr defns))
                          (pf (map caddr defns)))
                      (slot-set! self 'population-names pn)
                      (slot-set! self 'population-symbols ps)
```

```
                    (slot-set! self 'd/dt-list pf))
                  )
                 (abort
                  (string-append
                   " the format for defining a system of "
                   "populations is:\n"
                   "  (define-population-dynamics! self \n"
                   "     ("Grass" grass dgrass/dt)\n"
                   "     ("Rabbit" rabbit drabbit/dt)\n"
                   "     ("Fox" fox dfox/dt)\n"
                   "     ("Bear" bear dbear/dt))"
                   "where the arguments to each of the d/dt "
                   "are t grass rabbit fox bear"
                   )
                  )
                 )))



;---  sclos-model-body <dynamic-patch
(sclos-model-body <dynamic-patch>
                  (kdnl* 'model-bodies "In " (class-name-of self) t)
                  ;; Ok, I need to be able to refer to service
                  ;; directly (names) and to classes (types). Type
                  ;; values are aggregates of the members of the
                  ;; service-list of that type excluding any of those
                  ;; members specified by name members.

                  ;; We can tell the difference because names are
                  ;; required to be strings and types are required to
```

```
;; be symbols.

;; Changes in a type value are implemented pro-rata.

;;(dnl "Running <dynamic-patch> model body")

(if (<= dt 1e-12)
    (abort
     "Bad dt passed to <dynamic-patch> sclos-model-body"))
(let ((pop-values (map (lambda (x) (value self x))
                       (my 'population-names)))
      (d/dt-list (my 'd/dt-list))
      )

  (if (not (null? pop-values))
      (let ((dP (if (not (and d/dt-list (pair? d/dt-list)))
                    (lambda args 0)
                    (rk4* d/dt-list
                          t
                          (+ t dt)
                          (/ dt (my 'subdivisions))
                          pop-values)))
            )
        ;;(dnl "Got past the rk4* for " (my 'name) "
        ;;and " (my 'population-names))

        ;;(set-my! 'dP dP) ;; Don't really *need*
        ;;this, except perhaps for debugging
        (let ((deltas (dP (+ t dt))))
          (for-each
           (lambda (x v)
```

```
                        (if (not (zero? (imag-part v)))
                            (abort "got complex number, wanted a real"))

                        ;;(dnl (class-name-of self))


                        (add! self x (- v (value self x)))
                        ;; These are the adjustments due to
                        ;; consumption and predation (slot-set!
                        ;; self x v)
                        (if (< (value self x) 0.0)
                            (set-value! self x 0.0))
                        )
                       (my 'population-names) deltas))
                 )

              (aborts (symbol->string (class-name-of self))
                      " has no population names defined!"))
          )

        (kdnl* 'nested-habitat (name self) "@" t "/"
               (subjective-time self) ":" dt "/" (my 'dt))
        (parent-body)
        (kdnl* 'nested-habitat (name self) "@" t "/"
               (subjective-time self) ":" dt "/" (my 'dt))
        dt)


;-- <landscape> methods and body

;;        (filter (lambda (x) (and (contains? x loc) arg) ) (my 'patch-list)))
```

```
;; Default landscape only has the default value, oddly enough
(sclos-method (<landscape> <pair>) (value self loc)
              (if (contains? self loc)
                  ((my 'terrain-function) loc)
                  (my 'default-value)))

(sclos-method (<landscape> <pair>) (capacity self loc)
              (if (contains? self loc)
                  ((my 'terrain-function) loc)
                  (my 'default-value)))

;; This is to keep the "run" chain consistent
(sclos-model-body <landscape>
                  (kdnl* 'model-bodies "In " (class-name-of self) t)
                  (kdnl* 'nested-habitat (name self) "@" t "/"
                         (subjective-time self) ":" dt "/" (my 'dt))
                  (parent-body)
                  (kdnl* 'nested-habitat (name self) "@" t "/"
                         (subjective-time self) ":" dt "/" (my 'dt))
                  dt
                  )



;;(sclos-model-body <landscape>
;;                  (kdnl* 'running (my 'name) ":" (my 'representation) " is running")
;;                  (for-each (lambda (x)
;;                              (run-model-body x t dt))
```

```
;;                          (my 'patch-list))
;;                (parent-body)
;;                dt)



;-- <habitat> methods and body

;---- (make-habitat name default-ht domain terrain-function
;; domain is a list  ((minx miny minz) (maxx maxy maxz))
;; patch-data is a list of  services lists for make-patch

(define (make-habitat name default-ht domain terrain-function
                      patch-list
                      )
  (let* ((H (make <habitat> 'name name 'default-value default-ht
                  'minv (car domain) 'maxv (cadr domain)
                  'terrain-function terrain-function
                  'patch-list patch-list))
         )
    H)
  )


;--- Add things to the runqueue

;---- (add-habitat-to-queue Q h)
(define (add-habitat-to-queue Q h) ;; returns the queue, so us it like
                                   ;; (set! Q (add-...-queue Q hab))
  (let ((p (patch-list h))
        (s (service-list h))
```

```
        )
    (uniq (append Q (list h) p s))))


;--- sclos-method <habitat> (agent-prep self . args) Set preconditions
;                                                    for running
(sclos-method <habitat> (agent-prep self . args)
              (agent-prep-parent)
              )


;--- initialize <habitat> (initialize-parent self args) -- make a new habitat
(add-method initialize
            (make-method (list <habitat>)
                         (lambda (initialize-parent self args)
                           (initialise self (list 'scale #f))
                           (initialize-parent) ;; call "parents" last
                                               ;; to make the
                                               ;; initialisation list
                                               ;; work
                           )))

;--- sclos-method <habitat> (dump self . count) dumps the state of the habitat
;                                                agent in a readable way
(sclos-method <habitat> (dump self . count)
              (set! count (if (null? count) 0 (car count)))
              (display (make-string count #\space))
              (display "<habitat>\n")

              (let* ((slots (map car (class-slots (class-of self))))
```

```
                    (vals  (map (lambda (x) (slot-ref self x)) slots)))
               (for-each (lambda (x y)
                           (if (not (eq? x 'patch-list))
                               (begin
                                 (display (make-string (+ 2 count) #\space))
                                 (display x)
                                 (display ": ")
                                 (display y)
                                 (newline))))
                         slots vals))
             (display (make-string (+ 2 count) #\space))
             (display 'patch-list)
             (display ":\n")
             (for-each (lambda (x) (dump x (+ 4 count))) (my 'patch-list)))


;--- sclos-method (<habitat> <patch>) (add-patch self patch) add a
;                                                          patch to the habitat
(sclos-method (<habitat> <patch>) (add-patch self patch)
              (set-my! 'patch-list (uniq (cons patch (my 'patch-list)))))

;--- sclos-method (<habitat> <procedure>) (remove-patch self pfilter)
;  keep only patches which match a filter
(sclos-method (<habitat> <procedure>) (remove-patch self pfilter)
              (set-my! 'patch-list (filter pfilter (my 'patch-list))))

;--- (services...) returns services matching the sym or in the symlist
(sclos-method <habitat>
              (services self . ss)
              (if (not (null? ss))
                  (begin
```

```
                  (if (not (symbol? (car ss))) (set! ss (car ss)))
                  (filter (lambda (x) (member x ss)) (services self))
                  )
                (unique
                 (map string->symbol
                      (sort (map symbol->string
                                 (apply append
                                        (map services
                                             (patch-list self))))
                            string<?)))
                ))

;--- sclos-method (<habitat>) (service-list self . ss)
;; returns the slist of provided services: ss is an optional list of
;; patch names/symbols
(sclos-method (<habitat>)
              (service-list self . ss)
              (uniq
               (if (null? ss)
                   (apply append (map (lambda (x)
                                        (service-list x))
                                      (patch-list self)))
                   (apply append
                          (map (lambda (x)
                                 (service-list x
                                               (if (symbol? (car ss))
                                                   ss
                                                   (car ss))))
                               (patch-list self))))))
```

```
;--- sclos-method (<habitat>) (service-list self) ret a list of all ecoservices
(sclos-method (<habitat>) (service-list self)
              (let* ((P (patch-list self))
                     (S (map service-list P)))
                (apply append S)))


;--- (service?...) queries if a service is present
(sclos-method (<habitat> <symbol>) (service? self sym)
              (not (null? (services self sym))))


(sclos-method (<habitat> <pair>) (service? self symlist)
              (not (null? (services self symlist))))


;--- sclos-method (<habitat> <symbol>) (service-sites self sym)
; returns a list of patches with services
(sclos-method (<habitat> <symbol>) (service-sites self sym)
              (let loop ((rslt '())
                         (pl (my 'patch-list)))
                (cond
                 ((null? pl) rslt)
                 ((service? (car pl) sym)
                  (loop (cons (car pl) rslt) (cdr pl)))
                 (else (loop rslt (cdr pl))))))


(sclos-method (<habitat> <pair>) (service-sites self symlist)
              (let loop ((rslt '())
                         (pl (my 'patch-list)))
```

```
                (cond
                 ((null? pl) rslt)
                 ((service? (car pl) symlist)
                  (loop (cons (car pl) rslt) (cdr pl)))
                 (else (loop rslt (cdr pl))))))



;--- sclos-method (<habitat>) (patch-list self . arg)
; returns the listof patches (possibly filtered by names, symbols,
; procedures....
(sclos-method (<habitat>) (patch-list self . arg)
              (cond
               ((null? arg)
                (my 'patch-list))
               ((and arg (symbol? (car arg)))
                (let ((symlist arg))
                  (filter (lambda (p)
                            (let ((s (services p symlist)))
                              (and s (not (null? s)))))
                          (my 'patch-list))))
               ((and arg (pair? (car arg)))
                (let ((symlist (car arg)))
                  (filter (lambda (p)
                            (let ((s (services p symlist)))
                              (and s (not (null? s)))))
                          (my 'patch-list))))
               ((and arg (procedure? (car arg)))
                (let ((pfilter (car arg)))
                  (filter pfilter (my 'patch-list))))
```

```
               (else (my 'patch-list))))


;--- (aggregate-value self location radius servicelist)

(sclos-method (<habitat> <pair> <number> <pair>) (aggregate-value self location radius servicelist)
 (let* ((sl (service-sites self servicelist))
        (lsl (filter
              (lambda (patch)
                (> (intersection-of-two-circles
                     (distance-to-centre patch location)
                     radius (slot-ref patch 'radius))
                   0.0))
              sl))
        (lslv (if (null? lsl)
                  0.0
                  (apply + (map (lambda (patch)
                                  (* (value patch servicelist)
                                     (overlap-decay
                                      0.0 ;; 1.0 gives us 1% of the
                                          ;; pop at the radius, 0
                                          ;; gives us a uniform dist
                                      (distance (list-head (centre patch) 2)
                                                location)
                                      radius
                                      #t
                                      0.0
                                      (slot-ref patch 'radius)
                                      #t)
                                     ))
                                lsl))))
```

```
        )
   lslv))


;--- sclos-method <habitat> (min-bound self)
(sclos-method <habitat> (min-bound self)
              (let* ((v (map min-bound (slot-ref self 'patch-list)))
                     (vx (apply min (map car v)))
                     (vy (apply min (map cadr v)))
                     )
                (list vx vy)))

;--- sclos-method <habitat> (max-bound self)
(sclos-method <habitat> (max-bound self)
              (let* ((v (map max-bound (slot-ref self 'patch-list)))
                     (vx (apply max (map car v)))
                     (vy (apply max (map cadr v)))
                     )
                (list vx vy)))

;;1.0/(1.0 + exp(-2*pi*l*(2*(x+(0.5 - off)) - 1.0)) )
;; l = 1.0, off = 0.5



;; p is usually something like mm->points

;--- (<habitat> <procedure>) (map-log-data self logger format caller targets)
(sclos-method (<habitat> <procedure>)
              (map-log-data self logger format caller targets)
  (let* ((symlist (services H))
```

```
        (name (slot-ref H 'name))
        (plist (slot-ref H 'patch-list))
        (locs (centroid (map location plist)))
        )
    (let ((file (slot-ref logger 'file))
          (p (slot-ref self 'map-projection)))
      (if (or (not p) (null? p))  (set! p (lambda (x) x)))


      (ps 'moveto (list (p (car locs)) (p (cadr locs))))
      (if adjust-grey (ps 'setgray HABITATGREY))
      (ps 'Helvetica 12)
      (ps 'show (string-append (slot-ref H 'name)))

      (if (member 'nested-habitat nested-agents)
          (for-each (lambda (lpch)
                      (ps 'Helvetica 7)
                      (map-log-data lpch format caller targets p ps)
                      )
                    plist))
      )))

;--- (<habitat> <procedure>...) (log-data self logger format caller targets)
(sclos-method (<habitat> <procedure> <symbol> <procedure>)
     (log-data self logger format caller targets)
     (let ((file (slot-ref logger 'file))
           (p (slot-ref self 'map-projection)))
       (if (or (not p) (null? p))  (set! p (lambda (x) x)))
       (case format
         ((ps)
          (map-log-data self logger format caller targets p ps)
```

```
          (if (member 'nested-habitat nested-agents)
              (for-each (lambda (lpch)
                          (log-data lpch logger format caller targets ps p)
                          )
                        plist))
          )
         ((dump)
          (with-output-to-port
              (lambda ()
                (dump self))))

         ((text table)
          (log-data-parent)

          ))
       )
     )

;--- sclos-method (<habitat>) (spatial-scale self)
(sclos-method (<habitat>) (spatial-scale self)
              (if (not (my 'scale))
                  (let ((lscale (apply
                                 append
                                 (map
                                  (lambda (x)
                                    (map (lambda (y)
                                           (distance (location x)
                                                     (location y)))
                                         (patch-list self)))
                                  (patch-list self)))))
                    (set-my! 'scale
```

```
                              (/ (apply + lscale)
                                 (+ 1 (- (length lscale)
                                         (length (my 'patch-list))))))
                          ;;  This gets rid of the "self-distances" which are zero
                          )
                        )
                    (my 'scale))


;--- sclos-model-body <habitat>
(sclos-model-body <habitat>
   (kdnl* 'model-bodies "In " (class-name-of self) (name self) "@" t)

   (if (member 'nested-habitat nested-agents)
       (for-each (lambda (x)
                   (if (< (subjective-time x) (+ t dt))
                       (run x t (+ t dt) (my 'kernel))
                       ;;(dnl* "Skipping" (name x))
                       )
                   )
                 (my 'patch-list)
                 )
       )
   (kdnl* 'nested-habitat (name self) "@" t "/"(subjective-time self)
          ":" dt "/" (my 'dt) 'A)
   (parent-body)
   (kdnl* 'nested-habitat (name self) "@" t "/"(subjective-time self)
          ":" dt "/" (my 'dt) 'A)
   dt
   )
```

```
;--- sclos-method (<habitat> <symbol>)(extra-variable self field)
(sclos-method (<habitat> <symbol>)(extra-variable self field)
              (value self (symbol->string field)))


;--- sclos-method (<habitat>) (extra-variable-list self)
(sclos-method (<habitat>) (extra-variable-list self)
              (let ((patch-vars
                     (uniq (apply append
                                  (map extra-variable-list (my 'patch-list)))))
                    )
                ;;(dnl* "HABITAT PATCH VARIABLES:" patch-vars)
                (uniq (append (list 'name 'subjective-time) patch-vars))))
;; returns a list of symbols


;-  The End


;;; Local Variables:
;;; mode: scheme
;;; outline-regexp: ";-+"
;;; comment-column:0
;;; comment-start: ";;; "
;;; comment-end:""
;;; End:

; -*- mode: scheme; -*-
;-  Identification and Changes
```

```
;--
;  animal.scm -- Written by Randall Gray
;  Initial coding:
;     Date: 2016.07.13
;     Location: zero:/home/randall/Thesis/Example-Model/model/animal.scm
;
;  History:

;-  Discussion

;;; This pulls in the files for the animal class

(register-submodel 'simple-animal)
(register-submodel 'animal)

(load "animal-classes.scm")
(load "animal-declarations.scm")
(load "animal-methods.scm")

;-  The End


;;; Local Variables:
;;; comment-end: "-;" -;
;;; comment-start: ";;; " -;
;;; mode: scheme -;
;;; outline-regexp: ";-+" -;
;;; comment-column: 0 -;
;;; End:
```

```
;-  Identification and Changes

;--
;  animal-classes.scm -- Written by Randall Gray

;-  Code

(define <simple-metabolism>
  (make-class (inherits-from <object>)
              (state-variables days-of-hunger hunger-limit)))
(register-class <simple-metabolism>)

(define <metabolism>
  (make-class
   (inherits-from <object>)
   (state-variables stomach-contents
                    mass            ;; kg
                    structural-prop ;; structural-mass increases as
                                    ;; mass increases -- all things
                                    ;; that have metabolism must have
                                    ;; mass
                    structural-mass ;; the maximum amount it could eat
                                    ;; in a given period, assuming
                                    ;; aninfinite stomach
                    max-consumption-rate ;; the amount of "stuff" or "stuff-equivalent"
                    ;; needed per kg per unit of time
                    metabolic-rate
                    starvation-level ;; death if
                                     ;; (mass/structural-mass) goes
                                     ;; below this value
                    gut-size        ;; stomach capacity as a
```

```
                                ;; proportion of structural-mass
                condition       ;; "fat" reserves
                food->condition-conversion-rate
                condition->food-conversion-rate
                food->mass-conversion-rate
                mass->food-conversion-rate
                condition-conversion-rate
                mass-conversion-rate
                max-growth-rate
                max-condition-rate
                )
        ))

(register-class <metabolism>)

;; structural-mass is pegged at (max (* (my 'mass) (my
;; 'structural-prop))) through time
;; structural-prop is multiplied by (my 'mass) to give a number less
;; than or equal to the structural mas
;; metabolic-rate: (* metabolic-rate (my 'mass) dt) is the mass
;; required to maintain body mass for dt
;; ... consumption above that rate is converted to "condition"
;; == _
;;   | base-rate is the amount of mass removed per dt for stomach contents
;;   |_ condition-rate is the mass removed per dt for conditon contents
;;
;; starvation-rate is the amount of body mass removed per dt
;; starvation-level is the value of (/ (my 'mass) (my
;; 'structural-mass)) at which an organism dies.  This really ought to
;; be a number like 1.3
```

```
;; stomach-contents is an absolute amount of food
;; gut-size is a scalar multiplier of the structural-mass which
;; indicates the max cap. of the stomach
;; condition is an absolute number which is equivalent to mass in the
;; stomach
;; food->condition-conversion-rate is the efficiency of conversion
;; from stomach food to condition food
;; condition->food-conversion-rate is the efficiency of conversion
;; from condition to food
;; food->mass-conversion-rate is the efficiency of conversion from
;; stomach food to mass food
;; mass->food-conversion-rate is the efficiency of conversion from
;; mass to food
;; max-consumption-rate is the rate at which things can move through
;; the system

(define <simple-animal>
  (make-class (inherits-from <simple-metabolism> <thing>)
              (state-variables age sex habitat searchradius
                               foodlist homelist breedlist
                               domain-attraction
                               food-attraction
                               ))
  ) ;; lists of attributes it looks for for eating, denning and breeding
(register-class <simple-animal>)


;; current-interest is a function which takes (self age t dt ...) and
;; returns a meaningful symbol
;;
```

```
(define <animal>
  (make-class (inherits-from <metabolism> <thing>)
              ;; lists of attributes it looks for for eating, denning
              ;; and breeding
              (state-variables
               current-interest age sex
               habitat searchradius foodlist homelist breedlist
               movementspeed
               searchspeed
               foragespeed
               wanderspeed
               objective
               domain-attraction
               food-attraction
               near-food-attraction
               )

              ))
(register-class <animal>)

;-  The End


;;; Local Variables:
;;; mode: scheme
;;; outline-regexp: ";-+"
;;; comment-column:0
;;; comment-start: ";;; "
;;; comment-end:""
;;; End:
```

```
;-  Identification and Changes
;--
;  animal-declarations.scm -- Written by Randall Gray

;- <metabolism> generics
(declare-method eat "add food to the stomach, process food from the
stomach, grow, return the amount eaten")


;--- <animal> generics
(declare-method age "return the age of the animal")
(declare-method set-age! "set the age of the animal")
(declare-method sex "return the sex of the animal")
(declare-method set-sex! "set the sex of the animal")
(declare-method wander-around
                "Wander around with a bias toward a nomintated point")



;;; Local Variables:
;;; mode: scheme
;;; outline-regexp: ";-+"
;;; comment-column:0
;;; comment-start: ";;; "
;;; comment-end:""
;;; End:

;-  Identification and Changes

;--
;  animal.scm -- Written by Randall Gray
```

```
;  Initial coding:
;     Date: 2012.11.19
;     Location: odin:/home/gray/study/src/new/animal.scm
;
;  History:
;

;-  Copyright

;
;   (C) 2012 CSIRO Australia
;   All rights reserved
;

;-  Discussion

;-  Configuration stuff

;-  Included files

;-  Variables/constants both public and static

;--    Static data



;-  Variables/constants both public and static

(define AMINGREY 0.2)
(define AMAXGREY 0.05)
(define ACIRCGREY 0.0)
```

```
;(require 'charplot)

;-  Code

;---- <metabolism> methods

;;(define food-attenuation 0.25)
(define food-attenuation 0.5)
;;(define food-attenuation (/ 2.0 (exp 1.0))) ;; ~1.3


;----- initialize

(sclos-method <simple-metabolism> (initialize self args)
              (initialise self (list 'hunger-limit 20.0 'days-of-hunger 0.0))
              (initialize-parent)
              ;; call "parents" last to make the initialisation list work
              )



(sclos-model-body <metabolism> ;; A thing with metabolism *must* have mass
                  (kdnl* 'model-bodies "In " (class-name-of self))
                  (parent-body)

                  (if (not (number? (my 'mass)))
                      (kdnl* 'stomach (my 'name) "is dead, long live the King"))

                  (if (number? (my 'mass))
                      (let* ((struct-mass (my 'structural-mass))
```

```
(struct-prop (my 'structural-prop))
(starve-level (my 'starvation-level))
(mass (my 'mass))
(mass->food-conv-rate
 (my 'mass->food-conversion-rate))
(food->mass-conv-rate
 (my 'food->mass-conversion-rate))

(condn (my 'condition))

(cond->food-conv-rate
 (my 'condition->food-conversion-rate))
(food->cond-conv-rate
 (my 'food->condition-conversion-rate))

(guts (* struct-mass (my 'gut-size)))
(stomach-cont (my 'stomach-contents))
(stuffed-to-the-gills
 (and (number? mass)
      (* mass (my 'max-consumption-rate) dt)))
(max-dinner
 (and stuffed-to-the-gills
      (min stuffed-to-the-gills stomach-cont)))

(mgr (my 'max-growth-rate))
(cgr (my 'max-condition-rate))
(met-rate (my 'metabolic-rate))
(dead #f)

(cost (and mass (* met-rate mass dt)))
)
```

```
(if (not (number? cgr)) (set! cgr mgr))
;; condition can be put on at the same rate as
;; mass unless specified

;; do calculations here....

;; mass and cost are set to false when the
;; organism is dead

(kdnl* 'stomach "Metabolism: cost =" cost
       "stomach-contents =" stomach-cont)

(if cost
    (begin ;; deal with metabolic costs

      (kdnl* 'metabolism "D0   stomach ="
             stomach-cont "| condition =" condn
             "| mass =" mass "|=| cost = " cost)

      ;; .... with stomach-contents
      (let ((dc (- stomach-cont cost)))
        (set! stomach-cont (max dc 0.0))
        (set! cost (max (- dc) 0.0))
        )

      (kdnl* 'metabolism "D1   stomach =" stomach-cont
             "| condition =" condn "| mass =" mass
             "|=| cost = " cost)

      ;; .... with condition
```

```
(let ((cc (* cond->food-conv-rate condn)))
  (let ((dc (- cc cost)))
    (set! cc (max dc 0.0))
    (set! cost (max (- dc) 0.0))

    (set! condn (max 0.0
                     (/ cc
                        cond->food-conv-rate)))))

(kdnl* 'metabolism "D2   stomach =" stomach-cont
       "| condition =" condn "| mass =" mass
       "|=| cost = " cost)

;; .... with body mass
(let ((cc (* mass->food-conv-rate mass)))
  (let ((dc (- cc cost)))
    (set! cc (max dc 0.0))
    (set! cost (max (- dc) 0.0))

    (set! mass (max 0.0
                    (/ cc
                       mass->food-conv-rate)))))


(kdnl* 'metabolism "D3   stomach =" stomach-cont
       "| condition =" condn "| mass =" mass
       "|=| cost = " cost)

(if (or (<= mass struct-mass)
        (<= (/ mass struct-mass) starve-level))
    (begin
```

```
            (set! mass #f)
            (set-my! 'mass #f)) ;; Dead, y'know
          )
        )
      )

(if (and mass (> stomach-cont 0.0))
    (begin ;; Growth and condition updates

      (kdnl* 'metabolism "G0   stomach =" stomach-cont
             "| condition =" condn "| mass =" mass
             "|=| cost = " cost)

      ;; growth first
      (let* ((mm (/ stomach-cont food->mass-conv-rate))
             (mg (* mgr dt))
             (delta-m (min mm mg))
             )
        (kdnl* 'metabolism "g0a          mm =" mm
               " | mg =" mg " | delta-m =" delta-m)

        (set! mass (max 0.0 (+ mass delta-m)))
        (set! stomach-cont
              (max 0.0
                   (- stomach-cont
                      (* delta-m food->mass-conv-rate))))
        (kdnl* 'metabolism "g0b          mm ="
               mm" | mg =" mg " | delta-m =" delta-m)
        )

      (kdnl* 'metabolism "G1   stomach =" stomach-cont
```

```
            "| condition =" condn "| mass =" mass
            "|=| cost = " cost)

      ;; now condition
      (let* ((mm (/ stomach-cont food->cond-conv-rate))
            (mg (* cgr dt))
            (delta-m (min mm mg))
            )

        (kdnl* 'metabolism "g1a          mm =" mm
               " | mg =" mg " | delta-m =" delta-m)
        (set! condn (max 0.0 (+ condn delta-m)))
        (set! stomach-cont
              (max 0.0
                   (- stomach-cont
                      (* delta-m
                         food->cond-conv-rate))))
        (kdnl* 'metabolism "g1b          mm ="
               mm" | mg =" mg " | delta-m =" delta-m)
        )

      (kdnl* 'metabolism "G2   stomach =" stomach-cont
            "| condition =" condn "| mass =" mass
            "|=| cost = " cost)

      ))

(kdnl* 'metabolism "X stomach =" stomach-cont
       "| condition ="
       condn "| mass =" mass "|=| cost = " cost)
```

```
                    (set-my! 'condition condn)
                    (set-my! 'stomach-contents stomach-cont)
                    (set-my! 'mass mass)
                    (set-my! 'structural-mass struct-mass)
                    dt
                    )
                  'remove)
                 )
```

```
(sclos-method (<metabolism> <number> <number>) (eat self available-food dt)
              (if mass
                  (let* ((struct-mass (my 'structural-mass))
                         (mass (my 'mass))
                         (guts (* struct-mass (my 'gut-size)))
                         (stomach-cont (my 'stomach-contents))
                         (stuffed-to-the-gills (* mass
                                                  (my 'max-consumption-rate)
                                                  dt))
                         (gap (- guts stomach-cont))
                         (max-dinner (min gap
                                          stuffed-to-the-gills
                                          available-food))
                         (ate 0.0)
                         )
```

```
                    ;; do calculations here....

                    (begin ; eating
                      (set! ate max-dinner)
                      ;;(set! available-food (- available-food ate))
                      (set! stomach-cont (max 0.0 (+ stomach-cont ate)))
                      )

                    ;; return value
                    (if (< (/ mass struct-mass) (my 'starvation-level))
                        #f ;; false means death
                        (let ()
                          ;;(if (> (* mass struct-prop) struct-mass)
                          ;;   (set-my! self 'struct-mass struct-mass)
                          ;;   )
                          (set-my! 'stomach-contents stomach-cont)
                          ate
                          )
                        )
                    )
                  0)
                )

;---- animal methods

;----- (initialize)
(add-method initialize
            (make-method (list <animal>)
                         (lambda (initialize-parent self args)
                           ;;(dnl "<animal> init")
                           (slot-set! self 'current-interest
```

```
                                    (lambda args
                                      (aborts
                                       (string-append
                                        "current-interest isn't "
                                        "defined for a <animal>: ")
                                       (slot-ref self 'name) ":"
                                       (slot-ref self 'type) ":"
                                       (slot-ref self 'representation))))
                                (initialise self (list 'age #f 'sex #f))
                                (initialize-parent)
                                ;; call "parents" last to make the initialisation list work
                                )))

;----- (age)
(add-method age
            (make-method
             (list <animal>)
             (lambda (call-parent-method self)
               (slot-ref self 'age))))


;----- (set-age!)
(add-method set-age!
            (make-method
             (list <animal> <number>)
             (lambda (call-parent-method self n)
               (if (not (number? n))
                   (aborts "thing:set-age! -- bad number")
                   (slot-set! self 'age n)))))
```

```
;----- (sex)
(add-method sex
            (make-method
             (list <animal>)
             (lambda (call-parent-method self)
               (slot-ref self 'sex))))


;----- (set-sex!)
(add-method set-sex!
            (make-method
             (list <animal> <symbol>)
             (lambda (call-parent-method self n)
               (if (not (member n '(female male)))
                   (aborts "thing:set-sex! -- symbol should be male or female")
                   (slot-set! self 'sex n)))))

;----- (map-log-track-segment
(sclos-method <animal> (map-log-track-segment self track wt p ps)
              (if track
                  (let* ((xytrack (map txyz->xy track))
                         (ptrack (map p xytrack)))
                    (if (>= (length ptrack) 2)
                        (let ((startseg (list-head ptrack
                                                   (1- (length ptrack))))
                              (finishseg (cdr ptrack)))
                          (if adjust-grey (ps 'setgray wt))

                          (for-each
                           (lambda (ss fs)
                             (ps 'moveto ss)
```

```
                (ps 'moveto ss)
                (ps 'lineto fs)
                (ps 'stroke)
                )
             startseg finishseg)
            (ps 'Helvetica 4.5)
            (ps 'moveto (p (list-head (location self) 2)))

            (let ((m (my 'mass)))
              (if m
                  (begin
                    (ps 'show-centered (number->string (my 'mass)))
                    (ps-circle ps
                               (p (min 0.31415
                                       (* 0.25 pi
                                          (sqrt (my 'mass)))))
                               (p (list-head (location self) 2))
                               1.2 0.0 ))
                  (begin
                    (ps 'show-centered "Dead")
                    (ps-circle ps  (p 0.31415)
                               (p (list-head (location self) 2))
                               1.2 0.0 ))))

            )

          )
        )
      )
#t)
```

```
(sclos-method <animal> (map-log-data self logger format caller targets)
              (let ((file (slot-ref logger 'file))
                    (p (slot-ref self 'map-projection)))
                (if (or (not p) (null? p))  (set! p (lambda (x) x)))

                (let ((track (my 'track))
                      (tracks (my 'tracked-paths)))

                  (if track (map-log-track-segment self track ACIRCGREY p ps))
                  (if tracks
                      (let loop ((n (length tracks))
                                 (k 1.0)
                                 (tr tracks))
                        (if (not (null? tr))
                            (begin
                              (map-log-track-segment
                               self (car tr)
                               (+ AMINGREY (* (/ k n) (- AMAXGREY AMINGREY)))
                               p ps)
                              (loop n (1+ k) (cdr tr))))))
                  )
                #t)
              )

(sclos-method <animal> (log-data self logger format caller targets)
              (let ((file (slot-ref logger 'file))
                    (p (slot-ref self 'map-projection)))
                (if (or (not p) (null? p))  (set! p (lambda (x) x)))

                (kdnl* '(log-* log-animal) ":" targets)
```

```
                (case format
                  ((ps)
                   (map-log-data self logger format caller targets p ps)
                   )
                  (else
                   (log-data-parent))
                  )

                (if (and (assoc 'track-segments
                                (my 'state-flags))
                         (state-flag self 'track-segments))
                    (new-track! self))

                )
              )


(sclos-method (<animal> <number> <pair> <number> <number>)
              (wander-around self dt point attr speed) ;;
  (let* ((loc (location self))
         (p (unit (vector-to loc point)))
         (v (slot-ref self 'direction))
         (theta (- (nrandom pi 10) pi))
         (spd speed)
        )

    ;; This calculation ought to use random-angle and rotated-velocity
    ;; (which applies to lists!)  I'll keep it this way for local
    ;; clarity.

    (let* ((new-v (unit
```

```
                  (append (list (- (* (car v) (cos theta))
                                   (* (cadr v) (sin theta)))
                                (+ (* (cadr v) (cos theta))
                                   (* (car v) (sin theta))))
                                (cddr v))))

          (new-loc
           (map + loc
                (map *
                     (map +
                          (map * (make-list (length v) (- 1.0 attr)) new-v)
                          (map * (make-list (length p) attr) p))
                     (append (make-list 2 (* spd dt))
                             (make-list (length (cddr v)) 1.0)))))
          )
      (slot-set! self 'location new-loc)
      (slot-set! self 'direction new-v)
      new-loc)))


(sclos-method (<animal> <number> <pair> <number> <symbol>)
              (wander-around self dt point attr speedtag) ;;
              (wander-around self dt point attr (my speedtag)))

(sclos-model-body <animal>
   (kdnl* 'model-bodies "In " (class-name-of self) t)
   (let ((dt/2 (/ dt 2.0))
         (SQRT (lambda (x) (if (>= x 0) (sqrt x)
                               (if #t
                                   (begin
                                     (dnl "SQRT got a value of " x)
```

```
                                0)
                               (abort "I see ... a rhinoceros ...")
                               )) ) )
    ;;(kdnl* dnl*)
    ;;(kdnl* dnl)
    )
(kdnl* 'animal-running "[" (my 'name)
       ":" (class-name-of self) "]"
       " at " t "+" dt)
(kdnl* "******" 'running (my 'name)
       " the " (my 'representation)
       " is running" "******")
(set-my! 'age (+ (my 'age) dt/2))
(set-my! 'subjective-time (+ (my 'subjective-time) dt/2))

(parent-body)
;; should execute body code for <metabolism> and <thing>

(let* ((foodlist (my 'foodlist))
       (homelist (my 'homelist))
       (breedlist (my 'breedlist))
       (H (my 'habitat))
       (here (my 'location))
       (food 0.0) ;; nothing unless we find some
       (struct-mass (my 'structural-mass))
       (mass (my 'mass))
       (starve-level (my 'starvation-level))
       (stomach-cont (my 'stomach-contents))
       (guts (* struct-mass (my 'gut-size)))
       (condition (my 'condition))
       (ate 0.0)
```

```
     (objective (let ((o (my 'objective))) (if (null? o) #f o)))
     )

(kdnl* 'stomach "[" (my 'name) ":" (class-name-of self)
       "]" "===> stomach =" stomach-cont "| condition ="
       condition "| mass =" mass)
(if (number? mass)
    (begin
      (let ((focus ((my 'current-interest) self
                    (my 'age) t dt condition stomach-cont guts))
            )
        (kdnl* 'focus "[" (my 'name) ":"
               (class-name-of self) "]" " in now focussed on " focus)

        (cond
         ((not focus)
          (wander-around self dt (map (lambda (x) (/ x 2.0)) domain)
                         (* 2.0 (my 'domain-attraction)) 'wanderspeed)
          #t)
         ;;(aborts "Dinna y' ken?"))
         ((eq? focus 'wander)
          (wander-around self dt (map (lambda (x) (/ x 2.0)) domain)
                         (my 'domain-attraction) 'wanderspeed)
          )

         ((eq? focus 'hungry)
          (kdnl* 'debugging-eating "[" (my 'name)
                 ":" (class-name-of self) "]" "hungry")

          (let* ((foodsites
                  (patch-list H (lambda (x)
```

```
                    (let ((s (services x foodlist)))
                      (and s (not (null? s)))))))
      (foodvalue
       (map
        (lambda (x)
          (if (= food-attenuation 0.5)
              (/ (SQRT (value x foodlist))
                 (+ (spatial-scale H)
                    (distance here (location x))))
              (/ (pow (value x foodlist) food-attenuation)
                 (+ (spatial-scale H)
                    (distance here (location x)))))
          ) foodsites))

      (fooddata (sort (map cons foodvalue foodsites)
                      (lambda (x y) (> (car x) (car y)))))
      )
(kdnl* "Food ranks" "[" (my 'name) ":"
       (class-name-of self) "]"
       (map (lambda (x)
              (cons (car x)
                    (distance here
                              (location (cdr x)))))
            fooddata))


(if (zero? (length fooddata))
    (begin ;; No food possible ...
      (kdnl* 'debugging-eating "[" (my 'name)
             ":" (class-name-of self) "]" "Hunting")
      (wander-around self dt
```

```
                (map (lambda (x) (/ x 2.0)) domain)
                (my 'domain-attraction) 'foragespeed)
  )
(let ((target (cdar fooddata)))
  (if (contains? target (location self))
      (begin
        (kdnl* 'debugging-eating "[" (my 'name) ":"
               (class-name-of self) "]"
               "Reading the menu")
        (let* ((TV (value target foodlist))
               (ate #f))
          (kdnl* 'debugging-eating "[" (my 'name) ":"
                 (class-name-of self) "]"
                 "ordering the lot ("
                 (value target foodlist) "/"
                 (value target (services target))
                 ")and eating it")
          (let* ((total-food (value target foodlist))
                 (prop (/ total-food
                          (capacity target foodlist)))
                 (available-food
                  (* 2.0 total-food
                     (/ (SQRT prop) (1+ prop)))))
            (set! ate (eat self available-food dt))
            )
          (kdnl* 'debugging-eating "[" (my 'name) ":"
                 (class-name-of self) "]" "removing "
                 ate " from the patch ...")
          (scale! target foodlist (- 1.0 (/ ate TV)))
          (kdnl* 'debugging-eating "[" (my 'name) ":"
                 (class-name-of self) "]" "done.")
```

```
                  )
                (wander-around self dt (location target)
                               (my 'near-food-attraction)
                               'foragespeed)
                )
              (begin
                (kdnl* 'debugging-eating "[" (my 'name) ":"
                       (class-name-of self) "]"
                       "move toward the food source")
                (wander-around self dt (location target)
                               (my 'food-attraction)
                               'movementspeed)
                )))
          )
      )
    )
   ((eq? focus 'flee)
    ;; Not implemented yet
    #f
    )
   ((eq? focus 'breed)
    ;; Not implemented yet
    #f
    )
   )
  )

(set-my! 'objective (if (null? objective) #f objective))
(set-my! 'age (+ (my 'age) dt/2))
(set-my! 'subjective-time (+ (my 'subjective-time) dt/2))
```

```
            ;;(track-locus self t (my 'location)) ;;
            ;;even if they aren't moving ::
            ;;automatically done in <thing>
            )
          )

      ;; Ok now test condition and return
      (if (or (not mass) (< (my 'mass) (* 1.3 (my 'structural-mass))))
          (list 'remove self)
          dt)
      )
    )
  )


;-  The End


;;; Local Variables:
;;; mode: scheme
;;; outline-regexp: ";-+"
;;; comment-column:0
;;; comment-start: ";;; "
;;; comment-end:""
;;; End:


;-  Identification and Changes

;--
```

```
;  model.scm -- Written by Randall Gray
;  Initial coding:
;     Date: 2013.02.05
;     Location: odin:/home/gray/study/src/model.scm
;
;-  Code


;(dump wally)(newline)
;(aborts "Incomplete initialisation is making things fail when it runs")

;- Define the model domain now


(define start 0) ;; day zero
(define end 741) ;; end at some day after the start

(if (< end start) (error "The model doesn't work in that direction"))

;;(define log-schedtimes
(define log-schedtimes (append
                   (cons 0 (seq 6))
                  (map (lambda (x) (* 10.0 (1+ x))) (seq (/ end 10))))
  ) ;; first six days, then on every tenth day from the beginning


(define A4domain (list 210 294 80)) ;; (x y z) corresponds to the size of an A4 page
(define mA4domain (list 178 250 80))

(define domain mA4domain)
```

```
(set! nested-agents '(nested-habitat)) ;; No, each patch does its own thing....

;(add-kernel-message! 'stuff)
;(add-kernel-message! 'stuff)

;(add-kernel-message! 'stuff)
;(remove-kernel-message! 'stuff)
;(set-kernel-message! '(stuff))

;-- Global data ------------------------------------------------------------

(define missing-value 0)  ;; things arrived at by mortality are likely
                          ;; to be inexact
(define Q '())            ;; This is the queue which holds the agents
                          ;; in the simulation

;(define end (* 365.25 32)) ;; simulate 32 years
;(define end (* 365.25 10)) ;; simulate 10 years
;(define end (* 365.25 6))  ;; simulate 6 years
(define end (* 365.25 4))  ;; simulate 4 years
;(define end (* 365.25 2))  ;; simulate 2 years

(define record-ecoservice-history #t) ;; track ecoservice changes in a
                                      ;; list

(define (filter-hist-by-patch s d)
  (filter (lambda (x) (string=? (car x) s)) d))


(define A4domain (list 210 294 80))  ;; (x y z) corresponds to the
                                     ;; size of an A4 page
```

```
(define mA4domain (list 178 250 80)) ;; Model domain (x-size y-size
                                     ;; z-size)

(define domain mA4domain)

;--- Set scheduled tick times.
;;Scheduled dump times for the logger: by default the first six days,
;;then every tenth day from the start.  I should make it so that it
;;can be a function rather than a list.
(define schedtimes (append
                   (cons 0 (seq 6))
                   (map (lambda (x) (* 10.0 (1+ x))) (seq 7400))
  ) ;; first six days, then on every tenth day from the beginning for
    ;; 74000 days
)

;-- Set kernel flags ------------------------------------------------------

;; The kernel will emit messages (with kdnl*) which have a label which
;; matches something in the kernel-messages list

;(set! kernel-messages (append '(*) kernel-messages))

;; Indicate which agents are "nested"; as an example patches may be
;; present either as independent things or as components within a
;; habitat

(set! nested-agents '(nested-habitat)) ;; No, each patch does its own thing....

;(set! kernel-messages (append kernel-messages '(*)))
(set! kernel-messages (append kernel-messages '(introspection*  log-*)))
```

```
;; options include focus stomach hunger-proximity eating log animal-running

;-- extensions to basic framework (more complex models)  ---------------------

;---- Load habitat support code

;;(load "habitat-support.scm")

;- Load the model properly ------------------------------------------------

;-- load specific models ----------------------------------------------------

;; (make <landscape> ...)
;; (make <habitat> ...)
;; ...
;(append! Q ...)

(dnl "Registered submodels: " submodel-register)

(let ((submodel-files
       (!filter
        null?
        (map
         cdr
         (!filter
          null?
          (!filter
           (lambda (x) (member (car x) logger-tags))
           submodel-register))))
       ))
```

```
  (if (pair? submodel-files)
      (begin
        (dnl "Submodels: " submodel-files)
        (for-each (if #t
                      load
                      (lambda (x)
                        (display "loading submodel: ")
                        (display x)
                        (newline)
                        (load x))
                      )
                  submodel-files))
      (dnl "No submodel files to be loaded"))
  )


;;; loggers get inserted at the head of the queue

(let ((logger-files
       (!filter
        null?
        (map
         cdr
         (!filter
          null?
          (filter
           (lambda (x) (member (car x) logger-tags))
           submodel-register))))
       ))

  (if (pair? logger-files)
```

```
      (begin
        (dnl "Loggers: " logger-files)
        (for-each (if #t
                      load
                      (lambda (x)
                        (display "loading logger: ")
                        (display x)
                        (newline)
                        (load x))
                      )
                  logger-files))
      (dnl "No logger files to be loaded"))
  )


;-- Example code to run things....

(define (Doit q) ;; Run till end without paus
  (if #f
      (check-service-data-lists service-name-list
                                service-type-list service-eqn-sym-list))
  (prep-agents q start end)
  (set! q (queue start end q))
  )


(define Dunnit #f)
(define *dunnit* #f)

(define (doit q . n)
  (set! Dunnit (lambda () (shutdown-agents q)))
```

```
  (set! n (if (pair? n) (car n) 1))
  (if (not *dunnit*) (begin (prep-agents q start end) (set! *dunnit* 0)))
  (set! q (queue *dunnit* (+ *dunnit* n) q))
  (set! *dunnit* (+ *dunnit* n))
  )



;-- nominate the models to include



;;; Not currently working
;(define psdumper
;  (make <log-map> (list 'name "Map"
;                        'format 'ps
;                        'timestep-schedule schedtimes
;                        'filename "map-" 'filetype "0.ps"
;                       )
;       ))

;; <log-data> is pretty forgiving, but at the expense of verbosity
;; <log-agent-table> insists that only one agent be logged
;; <log-table> insists that all the agents possess all the fields,

(define logger
  (make <log-data> (list 'name "Data"
                        'timestep-schedule schedtimes
                        'filename "Data"
                        'variables (list 'name 'subjective-time 'value))
;; log-table does not automatically log the name at the front of the line
```

```
                        )
        )


(define habitat
  (make-habitat "Kunlun" 300 (list (list 0 0 300)
                                   (append A4domain (list 900)))
                (lambda (x y)
                  (abs (+  (* (- 120 x)
                              (- x 60)
                              (+ x 10))
                           (* (- y 120)
                              (- (* y x)
                                 560)
                              (+ y 80))
                           )
                                   ))
                (make-grid 3 3 '(0 0) '(210 294)
                           <patch> <polygon> "kunlun"
                           'environs %patch-initialiser)))

(for-each
 (lambda (p)
   (slot-set!
    p
    'service-list
    (list
     (simple-ecoservice "Trees" 't (+ 60 (+ 1 (* (random-real) 30))) ;; value
                        (+ 200 (+ 1 (* (random-real) 60))) ;; Capacity
                        1.0 ;; steepness of sigmoid
                        (days 7) ;; max dt
```

```
                        #t       ;; do growth
                        'sigmoid p)
     (simple-ecoservice "Fruit" 'f (+ 200 (+ 1 (* (random-real) 30))) ;; value
                        (+ 850 (+ 1 (* (random-real) 20))) ;; Capacity
                        1.0 ;; steepness of sigmoid
                        (days 7) ;; max dt
                        #t       ;; do growth
                        'sigmoid p)
     (simple-ecoservice "Seeds" 's (+ 500 (+ 1 (* (random-real) 30))) ;; value
                        (+ 1200 (+ 1 (* (random-real) 20))) ;; Capacity
                        1.0 ;; steepness of sigmoid
                        (days 7) ;; max dt
                        #t       ;; do growth
                        'sigmoid p)
     )))
 (slot-ref habitat 'patch-list))

(define Q '());

(define (iQ agnt)
  (set! Q (q-insert Q agnt Qcmp)))



(iQ habitat)
(for-each iQ (slot-ref habitat  'patch-list))

;;(set-introspection-list! psdumper (copy-list Q))

(set-introspection-list! logger (copy-list (service-list habitat)))
```

```
(for-each (lambda (x) (set-map-projection! x mm->points)) Q)

;;(if use-psdumper
;;  (set! Q (cons psdumper Q))
    (set! Q (cons logger Q))
;;  )

(define terminating-condition-test
  (let* ((tct terminating-condition-test)
         (l (lambda (Q)
              (and (tct Q)
                   (number? (slot-ref wally 'mass))
                   (number? (slot-ref wilma 'mass)))
              )
            ))
    l))
```

```
;;;============================================================================

(dnl "Run with (Doit Q) to run from the start to the end\n")
(dnl "Run with (doit Q n) to run for n days (not necessarily n ticks!)")
(dnl "             so you can run the next step in a similar fashion\n")
(dnl "Close up shop with (shutdown-agents Q) -- this closes files and things.")


(display "Loaded: ")
(apply dnl* (map (lambda (x) (slot-ref x 'name)) Q))

;-  The End


;;; Local Variables:
;;; mode: scheme
;;; outline-regexp: ";-+"
;;; comment-column:0
;;; comment-start: ";;; "
;;; comment-end:""
;;; End:
```

BIBLIOGRAPHY

Guile reference manual: Goops. https://www.gnu.org/software/guile/manual/html_node/GOOPS.html, a.

Mit/gnu scheme – scheme object system. https://www.gnu.org/software/mit-scheme/documentation/mit-scheme-sos/Introduction.html, b.

Stklos clos in scheme. Download: http://www.stklos.net/.

Michael P Bailey and William G Kemple. The scientific method of choosing model fidelity. In *Proceedings of the 24th conference on Winter simulation*, pages 791–797. ACM, 1992.

Eli Barzilay. http://www.barzilay.org/Swindle/.

Georgiy V Bobashev, D Michael Goedecke, Feng Yu, and Joshua M Epstein. A hybrid epidemic model: combining the advantages of agent-based and equation-based approaches. In *Simulation Conference, 2007 Winter*, pages 1532–1537. IEEE, 2007.

Daniel B Botkin, James F Janak, and James R Wallis. Some ecological consequences of a computer model of forest growth. *The Journal of Ecology*, 60:849–872, 1972a.

D.B. Botkin, J.F. Janak, and J.R. Wallis. Rationale, limitations, and assumptions of a northeastern forest growth simulator. *IBM J. Res. Dev.*, 16:101–116, 1972b.

D.B. Botkin, J.F. Janak, and J.R. Wallis. Some ecological consequences of a computer model of forest growth. *J. Ecol.*, 60:849–872, 1972c.

William John Chivers. *Generalised, parsimonious, individual-based computer models of ecological systems*. University of Newcastle, 2009.

Nicholson Collier and Michael North. Parallel agent-based simulation with repast for high performance computing. *SIMULATION*, 89(10):1215–1235, 2013. doi: 10.1177/0037549712462620. URL `http://sim.sagepub.com/content/89/10/1215.abstract`.

Sandro Jerônimo de Almeida, Ricardo Poley Martins Ferreira, Álvaro E Eiras, Robin P Obermayr, and Martin Geier. Multi-agent modeling and simulation of an aedes aegypti mosquito population. *Environmental Modelling & Software*, 25(12):1490–1507, 2010.

D.L. DeAngelis and L.J. Gross, editors. *Individual-Based Models and Approaches in Ecology: Populations, Communities and Ecosystems*. Chapman and Hall, isbn-10: 0412031612 edition, 1992.

D.L. DeAngelis, L.J. Gross, M.J. Huston, W.F. Wolff, D.M. Fleming, E.J. Comiskey, and S.M. Sylvester. Landscape modelling for everglades ecosystem restoration. *Ecosystems*, 1:64–75, 1998.

Donald Lee DeAngelis. Model for the movement and distribution of fish in a body of water. Technical report, Oak Ridge National Lab., Tenn.(USA), 1978.

Timothy DelSole and Jagadish Shukla. Model fidelity versus skill in seasonal forecasting. *Journal of Climate*, 23(18):4794–4806, 2010.

Bret D Elderd, Jonathan Dushoff, and Greg Dwyer. Host-pathogen interactions, insect outbreaks, and natural selection for disease resistance. *The American Naturalist*, 172(6):829–842, 2008.

Stefano Farolfi, Jean-Pierre Müller, and Bruno Bonté. An iterative construction of multi-agent models to represent water supply and demand dynamics at the catchment level. *Environmental modelling & software*, 25(10):1130–1148, 2010.

E.A. Fulton. Approaches to end-to-end ecosystem models. *Journal of Marine Systems*, 81(1):171–183, 2010.

E.A. Fulton, A.D.M. Smith, and A.E. Punt. Which ecological indicators can robustly detect effects of fishing. *ICES Journal of Marine Science*, 62(3):540–551, 2004.

EA Fulton, R Gray, M Sporcic, R Scott, and M Hepburn. Challenges of crossing scales and drivers in modelling marine systems. *18th World IMACS Congress and MODSIM09 International Congress on Modelling and SimulationâĂŹ, July*, pages 2108–2114, 2009a.

EA Fulton, R Gray, M Sporcic, R Scott, and M Hepburn. Challenges of crossing scales and drivers in modelling marine systems. *18th World IMACS Congress and MODSIM09 International Congress on Modelling and SimulationâĂŹ, July*, pages 2108–2114, 2009b.

E.A. Fulton, R. Gray, M. Sporcic, R. Scott, L.R. Little, M. Hepburn, B. Gorton, B. Hatfield, M. Fuller, T. Jones, W. De la Mare, F. Boschetti, K. Chapman, P. Dzidic, G. Syme, Dambacher J, and D. McDonald. Ningaloo collaboration cluster: Adaptive futures for ningaloo. Technical Report 5.3, Ningaloo Collaboration Cluster, Hobart, Tasmania, October 2011. URL `http://www.ningaloo.org.au/www/en/NingalooResearchProgram/Background.htm`.

R. Gray, E. Fulton, R. Little, and R. Scott. Ecosystem model specification within an agent based framework. Final Report 16, CSIRO Australia, Hobart, Tasmania, 2006a. ISBN 1 921061 80 4 (pbk), ISBN 1 921061 82 0 (pdf).

R Gray, EA Fulton, LR Little, and R Scott. *Operating model specification within an agent based framework. North West Shelf Joint Environmental Management Study Technical Report*. Number 16 in CSIRO-CMAR NWSJEMS Technical Reports. CSIRO, Hobart, Tasmania, Hobart, Tasmania, 2006b. ISBN 1 921061 80 4 (pbk), ISBN 1 921061 82 0 (pdf).

R Gray, EA Fulton, LR Little, and R Scott. *Operating model specification within an agent based framework. North West Shelf Joint Environmental Management Study Technical Report*. Number 16 in CSIRO-CMAR NWSJEMS Technical Reports. CSIRO, Hobart, Tasmania, Hobart, Tasmania, 2006c. ISBN 1 921061 80 4 (pbk), ISBN 1 921061 82 0 (pdf).

Randall Gray and Simon Wotherspoon. Increasing model efficiency by dynamically changing model representations. *Environ. Model. Softw.*, 30:115–122, April 2012. ISSN 1364-8152. doi: 10.1016/j.envsoft.2011.08.012.

V. Grimm and S.F. Railsback. *Individual-based Modeling and Ecology.* Princeton University Press, Princeton, New Jersey, 2005.

Volker Grimm, Uta Berger, Finn Bastiansen, Sigrunn Eliassen, Vincent Ginot, Jarl Giske, John Goss-Custard, Tamara Grand, Simone K. Heinz, Geir Huse, Andreas Huth, Jane U. Jepsen, Christian JÃ¿rgensen, Wolf M. Mooij, Birgit MÃijller, Guy Pe'er, Cyril Piou, Steven F. Railsback, Andrew M. Robbins, Martha M. Robbins, Eva Rossmanith, Nadja RÃijger, Espen Strand, Sami Souissi, Richard A. Stillman, Rune VabÃ¿, Ute Visser, and Donald L. DeAngelis. A standard protocol for describing individual-based and agent-based models. *Ecological Modelling*, 198(1-2):115 – 126, 2006. ISSN 0304-3800. doi: DOI:10.1016/j.ecolmodel.2006.04.023. URL `http://www.sciencedirect.com/science/article/B6VBS-4K606T7-3/2/1dad6192bec683f32fce6dee9d665b51`.

C.J. Harvey, S.P. Cox, T.E. Essington, S. Hansson, and J.F. Kitchell. An ecosystem model of food web and fisheries interactions in the baltic sea. *ICES Journal of Marine Science*, 60:939–950, 2003.

M. Huston, D.L. DeAngelis, and W. Post. New computer models unify ecological theory. *BioScience*, 38:682–691, 1988.

Gregor Kiczales. Tiny clos. Download – ftp://ftp.parc.xerox.com/pub/mops/tiny/.

L.R. Little, E.A. Fulton, R. Gray, D. Hayes, R. Scott, A.D. McDonald, and K. Sainsbury. Management strategy evaluation results and discussion for the north west shelf. Final Report 14, CSIRO Australia, Hobart, Tasmania, 2006. ISBN 1 921061 74 X (pbk), ISBN 1 921061 76 6 (pdf).

S. Luke, C. Cioffi-Revilla, L. Panait, and K. Sullivan. Mason: A new multi-agent simulation toolkit. In *Proceedings of the 2004 SwarmFest Workshop*, 2004.

V. Lyne, R. Gray, K. Sainsbury, and R. Scott. Integrated biophysical model investigations. Final report, CSIRO Australia, Division of Fisheries, Hobart, Tasmania, 1994a.

V. Lyne, R. Gray, K. Sainsbury, and R. Scott. Integrated biophysical model investications. Final report, CSIRO Australia, Division of Fisheries, Hobart, Tasmania, 1994b.

Thomas Robert Malthus. *An Essay on the Principle of Population.* Library of Economics and Liberty, Internet, 16 feb 2010 edition, 1798. URL `http://www.econlib.org/library/Malthus/malPop.html`. First edition, originally published by J. Johnson, London.

N. Minar, R. Burkhart, C. Langton, and M. Askenazi. The swarm simulation system: A toolkit for building multi-agent simulations, 1996.

L. Monte. A methodological approach to develop *contaminant migration-population effects* models. *Ecological Modelling*, 220:3280–3290, 2009.

Kenneth A. Rose, J. Icarus Allen, Yuri Artioli, Manuel Barange, Jerry Blackford, FranÃğois Carlotti, Roger Cropp, Ute Daewel, Karen Edwards, Kevin Flynn, Simeon L. Hill, Reinier HilleRisLambers, Geir Huse, Steven Mackinson, Bernard Megrey, Andreas Moll, Richard Rivkin, Baris Salihoglu, Corinna Schrum, Lynne Shannon, Yunne-Jai Shin, S. Lan Smith, Chris Smith, Cosimo Solidoro, Michael St. John, and Meng Zhou. End-to-end models for the analysis of marine ecosystems: Challenges, issues, and next steps. *Marine and Coastal Fisheries: Dynamics, Management, and Ecosystem Science*, 2:115 – 130, 2010.

M. Scheffer, J.M. Baveco, D.L. DeAngelis, and K.A. Rose. Super-individuals: a simple solution for modelling large populations on an individual basis. *Ecological Modelling*, 80:161–170, 1995.

Jan C Thiele and Volker Grimm. Netlogo meets r: Linking agent-based models with a toolbox for their analysis. *Environmental Modelling & Software*, 25(8):972–974, 2010.

Christian Ernest Vincenot, Francesco Giannino, Max Rietkerk, Kazuyuki Moriya, and Stefano Mazzoleni. Theoretical considerations on the combined use of system dynamics and individual-based modeling in ecology. *Ecological Modelling*, 222(1):210 – 218, 2011a. ISSN 0304-3800. doi: DOI:10.1016/j.ecolmodel.2010.09.029. URL `http://www.sciencedirect.com/science/article/B6VBS-518MX70-2/2/199628e93c44d13863a48b3299472a32`.

Christian Ernest Vincenot, Francesco Giannino, Max Rietkerk, Kazuyuki Moriya, and Stefano Mazzoleni. Theoretical considerations on the combined use of system dynamics and individual-based modeling in ecology. *Ecological Modelling*, 222(1):210 – 218, 2011b. ISSN 0304-3800. doi: DOI:10.1016/j.ecolmodel.2010.09.029. URL `http://www.sciencedirect.com/science/article/B6VBS-518MX70-2/2/199628e93c44d13863a48b3299472a32`.

Carl J Walters and Steven JD Martell. *Fisheries ecology and management*. Princeton University Press, 2004.

webpage, August 2009. URL `http://www.ningaloo.org.au/www/en/NingalooResearchProgram/Background.htm`.

U. Wilensky. Netlogo, 1999. URL `http://ccl.northwestern.edu/netlogo/`.

W. F. Wolff. An individual-oriented model of a wading bird nesting colony. *Ecological Modelling*, 72(1-2):75 – 114, 1994. ISSN 0304-3800. doi: DOI:10.1016/0304-3800(94)90146-5. URL `http://www.sciencedirect.com/science/article/B6VBS-48YNSFT-2X/2/09ef35a0a8e95b3261adfc540cd87a23`.

WS Yip and Thomas E Marlin. The effect of model fidelity on real-time optimization performance. *Computers & chemical engineering*, 28(1):267–280, 2004.