THE ESCALATOR BOXCAR TRAIN

USER'S MANUAL

Version 4.1

André M. de Roos

Department of Pure and Applied Ecology
University of Amsterdam
Kruislaan 320, 1098 SM Amsterdam
THE NETHERLANDS
E–mail: aroos@bio.uva.nl

April 3, 2006

Contents

List of Tables

Chapter 1

Introduction

The *Escalator Boxcar Train* (frequently abbreviated hereafter as EBT) is a program package, that can be used to study the dynamics of physiologically structured, biological populations. Unfortunately the terminology is slightly ambivalent, since the name *Escalator Boxcar Train* in essence refers to the numerical method that was specifically developed to integrate the type of equations, in terms of which the models for these structured populations are couched. The EBT–package is my implementation of this numerical technique and this manual describes how to use it for a particular problem.

This manual will not tell you what a physiologically structured population model is, nor how to formulate one. Moreover, the manual will only do a half–hearted job to describe in detail the numerical method itself. Basically, the formulation of a physiologically structured population model gives rise to one or more partial differential equations (PDEs), which describe the dynamics of the populations itself. These equations can be accompanied by one or more ordinary differential equations (ODEs) which describe the dynamics of quantities that characterize the environment of the population, such as food density. The *Escalator Boxcar Train* as a numerical integration technique constitutes a method to approximate consistently this mixture of partial and ordinary differential equations into a (unfortunately rather large) set of ordinary differential equations only. As one of the special properties of the *Escalator Boxcar Train* this approximate system of ODEs can change in size during the integration. The EBT–package is designed to carry out the numerical integration of this system and to handle properly the changing number of ODEs. The package can handle the $2nd$–order version of the *Escalator Boxcar Train*, as discussed in (1).

To use the EBT–package properly it is necessary to have a firm understanding of physiologically structured population models in general. The basic theory of these models is described in a number of different publications to be found in the literature of the last decade (see, for example, (2; 3; 4)). Moreover, it is a prerequisite to know how to obtain the EBT–formulation of the particular model that you want to study. A description of the *Escalator Boxcar Train*, which could be seen as a numerical method for partial differential equations as well as a modelling methodology in itself can be found in (5; 6; 1). This background knowledge is really required to use this program package in a sensible way. Physiologically structured population models are not at all easy to study and the numerical results obtained with the EBT–program, as all results from numerical computations, should be treated with caution. Only a good understanding of the model that is studied can lead to a proper interpretation.

The EBT–package is set up as a template program which can be adapted for any particular problem that the user wants to study. The philosophy behind this approach is that any program that implements the *Escalator Boxcar Train* would have a highly similar structure with a whole set of routines carrying out jobs that are completely model–independent (*e.g.*, the organization and bookkeeping of cohorts). Therefore, it is possible to choose one specific program structure, such that implementing a new problem only requires changing a handful of routines (actually, it ended up to be 7 routines in this version). The current package reflects my choice of that basic program structure.

Choosing a basic program structure implies that a decision has to be made between flexibility and generality on the one hand and the ease with which a new problem is implemented on the other. Implementing a particular structured population model could be made easier, if one is willing to make more restrictive assumptions about the form of the model. This package has developed over the years, in which I studied a variety of different problems with it, every single problem usually having one or more unique features. This diversity in models studied and their unique character has definitely tipped the balance towards flexibility and generality. Hence, implementing a new structured population model will require some skill and a decent understanding of the equations that are to be studied, which once more stresses the need for a basic knowledge about physiologically structured population models.

I will not spend any time on discussing the particular choices I have made during the development of the package. They all seemed reasonable at the time I made them, but undoubtedly better choices could have been made (had I thought about them). Also, the program has never been optimized to generate the fastest code possible. If somebody ever feels the need to construct a better and faster version, I would be delighted. As said, the EBT–package forms the program that I use myself to study the dynamics of many different structured population models. After a number of request from a variety of people I have done a, probably crude, polishing job to make it available for use by other people as well and to supply it with some sort of a manual. This leaves much to be desired, above all some kind of user–friendly interface for implementing the model and executing integration runs. These are features I have designated as desirable extensions that, given the time (which I usually do not have), I will try to implement one day. More serious short–comings than this lack of fancy features I will treat with a lot more gravity. At this very moment I'm not aware of any serious bugs in the program, but if anybody happens to find one, please contact me to help me fix it.

1.1 Manual organization

Studying the dynamics of a particular physiologically structured population model first of all requires its appropriate EBT–formulation (refer to (5; 6; 1)). With the EBT–formulation in hand studying the model dynamics boils down to a cyclic process involving the following steps:

- Model implementation:
  1. Specification of the model dimensions
  2. Specification of the model equations
- Program building:
  1. Compilation of the integration program from its source files
- Program execution:
  1. Specification of integration settings and parameters
  2. Specification of the initial population and environment data
  3. Execution of the program and inspection of output

The model implementation part is the topic of chapter 2 and 3. These chapters will describe the general structure of the program, the data structures that are used to represent the populations and the environment, and the syntax for addressing these data structures. Using a particular example of a physiologically structured population model, it will be shown which steps have to be carried out during implementation.

In chapter 4 it is shown how to build an executable program from the source files, consisting of the library source files contained in the EBT–package and the model–specific files programmed by the user.

Chapter 5 discusses how to execute the program and which input files are required. Program execution can be carried out an arbitrary number of times, varying parameters and initial conditions, without the need to re–compile the program anew. Compilation of the program is only necessary if the dimensions of the model or its equations have been changed. The specification of the model dimensions and equations and of the settings, parameters and initial data is carried out by changing specific files, for which one can use his or her favored text editor. Compilation and execution of the program is essentially done by typing a compilation command in a specific format on the command–line of the computer system that is used. The processing of the program output is left totally to the user. One or the other plotting program should be used to study the results, which are delivered in a simple text file with columns of numbers, representing quantities that were defined by the user in the model implementation phase. From this description it is clear that the current EBT–package is only a building block to be completed by a text editor and a plotting facility to form some kind of integrated environment for studying the dynamics of structured population models.

Since I'm working on a UNIX–system myself, the package includes some Bourne–shell scripts that make the compilation and execution steps somewhat easier. These scripts are described in chapter 6.

1.2 Notational conventions

In the following I will (try to) be consistent in my notation and use different typefaces for different purposes. The notation used is shown in table I.

1.3 Portability

The package is intended to crash most, if not all, computer systems, but there are, of course, some lucky exceptions. The program itself is written in the C–language, following the standard adopted by the American National Standards Institute (ANSI) and the International Standards Organization (ISO) as described by (7). This implies that the source code will probably compile without problems on any system when an "ANSI–like" C–compiler is used.

1.4 Package contents

The EBT–package contains a collection of files that can be subdivided into three categories: library files, template files and script files. Library files contain routines that are entirely independent of the structured population model itself, except for its dimensions (number of structured populations, number of environmental variables, etc.). The set of template files contains templates for the implementation of any particular structured population model in the 2*nd*– order version of the *Escalator Boxcar Train*–method. The template files actually

Table I: Notational convention used in this manual.

Typeface	Represents
bold	Bold typeface is used to indicate file and directory names.
<*italic*>	Italic typeface surrounded by brackets indicates a placeholder for information. For example, any reference to <*model*> should be replaced by the actual name that you have given to the model you are working on.
`typewriter`	Typewriter font is used for all statements in the C–language and all commands that have to be typed literally on the command–line of the operating system to invoke programs or start a particular shell script, such as described in chapter 6. The typewriter font is also used when only parts of these statements or commands are quoted.

implement a particular example of a structured population model that is described in section 2.3. The package also contains templates for the input files that specify the control variables and parameters for this example model, as well as templates for the input files that specify its initial condition. The script files are Bourne–shell scripts that can be used in a UNIX–environment for different tasks, such as building the executable program and running it at a lower priority.

In addition to these files that form the core of the EBT–package, a directory **examples** is included with the definition files for models that I have studied using the package and that have been described in one or the other publication. I intend to extend this **examples** directory over time, adding new models as they appear in the literature. This allows me to include examples of model implementations without the need to describe them in detail. The list in the following sections will give the reference to the publications in which the example model is described. Every model has its own subdirectory in the **examples** directory, each subdirectory containing the model dimension file, the model definition file, an appropriate control variable file and an example initial state file.

The following sections give a more detailed description of all the files and subdirectories in the EBT–package:

1.4.1 Library files

escbox.h:	Basic header file with global definitions that have to be accessible to all routines, including constant and type definitions that are used in the population data structures.
ebtmain.c:	The main program file containing the `main()` routine and the primary routines that handle initialization, integration, output generation and shut–down of the program.
ebtinit.c:	Routines used by the primary initialization procedure to read in the initial state of the population and the environment and to read in the values of the various control variables and parameters.
ebtcohrt.c:	Routines that are used to keep the cohort data structures organized and to deal with their increasing and decreasing number.
ebttint.c:	Routine for the specification of the actual time integration of the ODEs.
ebtrk2.c:	Routines for the 2nd order Runga-Kutta integration method with fixed stepsize.
ebtrk4.c:	Routines for the 4th order Runga-Kutta integration method with fixed stepsize.
ebtrkf45.c:	Routines for the Runge-Kutta-Fehlberg 4/5th order integration method with an adaptive step size.
ebtrkck.c:	Routines for the Cash-Karp Runga-Kutta integration method with adaptive stepsize.
ebtdopri.c:	Routines for the an explicit Runge-Kutta method of order (4)5 due to Dormand and Prince with step size control and dense output.
methtest.c:	This file is only processed by the preprocessor to issue a statement which time integration method is being used.
ebtstop.c:	Routines used during the shutting–down of the program to write a report with run statistics and to save the final state of the system.
ebtutils.c:	Low level routines mainly dealing with memory allocation and de–allocation and with error handling.

ebtmain.h, ebtinit.h, ebtcohrt.h, ebttint.h, ebtstop.h, ebtutils.h: The header files supplying the interface to the global variables and functions defined in **ebtmain.c, ebtinit.c, ebtcohrt.c, ebttint.c, ebtstop.c, ebtutils.h**, respectively.

ebttune.h:	Include file to tailor the EBT to the current system. Explicit support for Linux, DEC/Alpha, Sun/SPARC with gcc, CRAY C916.
shmem.h:	Include file for shared memory access.

1.4.2 Template files

ebttmpl.c: Template of the model definition file for the implementation of a structured population model, using the $2nd$–order *Escalator Boxcar Train*–method.

ebttmpl.h: Model dimension file, specifying the dimensions of the structured population model that is implemented in **ebttmpl.c**.

tmpl.cvf: Input file specifying the values of the control variables and the parameters for the model implemented in **ebttmpl.c**.

tmpl.isf: Input File specifying a possible initial state for the model implemented in **ebttmpl.c**.

1.4.3 Script files

ebt: Bourne–shell script to be used in an UNIX–environment for building the executable EBT–program from the model definition and library source files.

ebtclean: Bourne–shell script to be used in an UNIX–environment for deleting files. Asks to confirm the removal of executable programs, object files or output files generated by an EBT–program during integration.

1.4.4 Contents of the examples directory

ebttmpl.c: Template of the model definition file for the implementation of a structured population model, using the $2nd$–order *Escalator Boxcar Train*–method.

ebttmpl.h: Model dimension file, specifying the dimensions of the structured population model that is implemented in **ebttmpl.c**.

tmpl.cvf: Input file specifying the values of the control variables and the parameters for the model implemented in **ebttmpl.c**.

tmpl.isf: Input File specifying a possible initial state for the model implemented in **ebttmpl.c**.

Chapter 2

Program overview

2.1 The EBT–formulation of a structured population model

A physiologically structured population model describes the dynamics of an arbitrary number of biological populations, in which the individuals are characterized and distinguished from each other by a set of physiological traits that are called *individual state variables* (*i–state* variables) (3; 4). The number of *i–state* variables with which every individual is labelled is again arbitrary, although the EBT–package is based on the assumption that this number is identical for every structured population in the model. To study the dynamics of a physiologically structured population model numerically (as is the aim of the EBT–package), the structured population has to be subdivided into distinct groups of individuals that are more or less similar. These groups of individuals are called cohorts and every structured population consists of a collection of these cohorts. Every cohort is now characterized by a specific set of statistics, notably the total number of individuals in the cohort and the mean *i–state* within the cohort. The dynamics of these statistics in every single cohort is described by ordinary differential equations. The exact form of these ODEs can be found in or at least inferred from (5; 6; 1).

The cohorts that make up a structured population come in two different varieties: *internal cohorts* and *boundary cohorts* (or "cohorts in creation"). The fundamental difference between these two types is that the boundary cohorts are in the process of being created from the individuals that are born at a particular time. Even if a structured population is initially subdivided into distinct cohorts, the production of new offspring can in principle be (and is most likely) a continuous process. Hence, to keep the population subdivided into cohorts, the continuous inflow of newborns has to be organized into new cohorts. The cohorts that receive such a reproductive input are said to be "in creation" or boundary cohorts. Internal cohorts do not receive such an input of new individuals and are hence virtually closed off groups (see (5; 6; 1) for a more elaborate description).

The number of boundary cohorts that are created at a particular time may be arbitrary large, and as a consequence the number of internal cohorts will fluctuate over time as well. Just like the internal cohorts, the boundary cohorts are characterized by a set of statistics, although these statistics are defined in a slightly different way (5; 6; 1). At specific points during the integration the existing set of boundary cohorts is absorbed into the structured population, which it belongs to, and a new set is created. During this inclusion into the population, the statistics that characterize a boundary cohort are transformed into their counterparts characterizing an internal cohort. The time interval between two of these moments at which the sets of boundary cohorts are changed will be called the *cohort interval*. In principle the length of this cohort interval is fixed and hence changing the boundary cohorts happens at equidistant time values during the integration. However, the current version of the EBT–package incorporates a method to force closure of the boundary cohorts in between these equidistant time values (see section 3.5.8 on page 30). The whole cycle of creating the sets of boundary cohorts for every population, the integration of the statistics during the cohort interval and the transformation of the boundary cohorts into internal cohorts at the end of a cohort interval, will be referred to as a *cohort integration cycle*.

As a special feature of the current version of the EBT–package, cohorts can, in addition to the characterization with *i–state* variables, also be labelled with *individual state constants* (or *i–state* constants). The difference between these two classes of variables is just that the *i–state* variables change continuously over time, while the *i–state* constants change in a discontinuous manner only. The dynamics of the *i–state* variables are hence specified in terms of ODEs, the changes in the *i–state* constants are to be implemented by the user as instantaneous transformations that will mostly occur at the end of a cohort integration cycle. An *i–state* constant might, for example, be used to model the instantaneous changes in length of animals with an exo–skeleton, that grow by molting.

In non–linear problems, the dynamics of the structured populations also depend upon a set of variables that characterize the environment in which the populations live, referred to as the *environmental variables* (or *E–state* variables). The number of these *E–state* variables is again arbitrarily large. The *E–state* variables are either explicit, time–dependent functions or dynamically varying. In the latter case the dynamics are also described by ODEs. Again, you have to consult (5; 6; 1) for the exact formulation of these ODEs.

2.2 The basic program cycle

From the previous section it is clear that the main task of the program boils down to integrating during a cohort interval simultaneously all environmental variables and all statistics of the internal and boundary cohorts that make up the structured populations in the model. As stated, the dynamics of these quantities are described by ODEs, and the program is therefore in principle nothing else than an implementation of an often used numerical integration method for ordinary differential equations. The numerical integration method used is the $4th/5th$–order Runge–Kutta–Fehlberg method with adaptive step size. This method performs every integration step twice but uses both times a different method: the first time a $4th$–order Runge–Kutta method and subsequently a $5th$–order Runge–Kutta method. The difference between the integration results obtained with the two different methods is used to adapt the step size in the integration (see (9)).

The subdivision of the populations into distinct cohorts and the concomitant organization of the cohort data structures make the program, however, different than just an ordinary differential equation solver. After the statistics have been integrated over time until the end of the cohort cycle, a variety of transformations and re–organizations of the cohort data are initiated or allowed:

- At the end of a cohort cycle the statistics characterizing the boundary cohorts are first turned into their counterparts that characterize the internal cohorts (5; 6; 1).
- If implemented, the program can subsequently carry out user–defined transformations, which make the cohort data change in a discontinuous manner. These transformations can, for example, remove individuals that have reached a maximum lifespan or implement a reproduction process that is pulsed in time. In addition, they can be used to change the current values of the *i–state* constants.

Table I: The equations modelling the individual consumer behavior and the autonomous food dynamics, which form the ingredients of the simple example model of an (age,length)–structured consumer population feeding on an unstructured food source.

Individual feeding rate:	$I(S,\ell)$	$=$	$\dfrac{S}{1+S}\,\ell^2$	
Growth rate in length:	$g(S,\ell)$	$=$	$\dfrac{S}{1+S} - \ell$	
Reproduction rate:	$b(S,\ell)$	$=$	$\alpha\dfrac{S}{1+S}\,\ell^2$	
Mortality rate:	$q(a)$	$=$	δ	for $a < A_{max}$
		$=$	∞	otherwise
Food recovery rate:	$R(S)$	$=$	$r_m S(1 - S/K)$	

- Next, the current boundary cohorts for every population are absorbed into the collection of internal cohorts that make up each of these populations.
- After the incorporation of the boundary cohorts, the program screens all the cohorts present to determine whether the total number of individuals in the cohorts has become negligible or whether cohorts have become identical in the value of their statistics (more specifically in the value of their mean *i–state*). "Negligible" and "identical" in this respect mean that the number of individuals in the cohort or the difference between the mean *i–state* variables is less than a set of tolerance values that is under the control of the user. Negligible cohorts are removed from the population, identical ones are lumped together, in which the weighted mean of their statistics is taken as the new value.
- Subsequently, the program determines if output is requested at this time during the integration. If so, the program writes the output quantities, as defined by the user, to the output file.
- After all these operations the program starts a new cohort integration cycle by setting up new sets of boundary cohorts for every population. The number and nature of these boundary cohorts, being part of the model–specific implementation, are under control of the user. The basic program cycle is now repeated, starting with the integration of the variables up to the end of the next cohort integration interval.

2.3 Example of a structured population model

In the next chapter it will be explained, how in general one implements a particular structured population model using the EBT–package. The required steps, which involve the specification

Table II: The equations describing the dynamics of the size and state of the consumer population, as represented by the density function $n(t, a, \ell)$, and the concentration of food particles, as represented by the variable S. The model is based on the modelled individual behavior, specified in table I.

$$\frac{\partial n(t,a,\ell)}{\partial t} + \frac{\partial n(t,a,\ell)}{\partial a} + \frac{\partial g(S,\ell)\, n(t,a,\ell)}{\partial \ell} = -\, q(a)\, n(t,a,\ell)$$

$$\begin{aligned} n(t,0,\ell) &= \int_0^1 \int_0^\infty b(S,\ell)\, n(t,a,\ell)\; da\, d\ell && \text{if } \ell = 0 \\ &= 0 && \text{otherwise} \end{aligned}$$

$$n(0,a,\ell) = \Psi(a,\ell)$$

$$\frac{dS}{dt} = R(S) - \int_0^1 \int_0^\infty I(S,\ell)\, n(t,a,\ell)\; da\, d\ell\,, \qquad S(0) = S_0$$

of the model dimensions and equations, will be illustrated using a simple example model. This example model describes the dynamics of an (age,length)–structured consumer population feeding on a dynamically varying food source. The model is essentially the same as the model, which is used in (1) to illustrate the *Escalator Boxcar Train* technique. The single difference is that individuals can only live for a finite time, *i.e.*, all individuals die with certainty upon reaching a specific maximum age. Since only a few individuals will reach the maximum age and die instantaneously when realistic parameter values are used, the finite lifespan will only have a minimal effect upon the dynamics. However, due to the finite lifespan the number of cohorts in the structured population is bounded. In this section I will briefly discuss the model and the resulting equations.

The individuals of the consumer population are characterized by both their age, denoted by a, and their length ℓ. The surface area and the weight of the individuals are assumed to be proportional to, respectively, the second and third power of their length. The individuals feed with a feeding rate proportional to their surface area. Moreover, the feeding rate per unit surface area increases with the concentration of food particles S, following a hyperbolic relation, the Holling type II functional response. The food source itself recovers from this grazing activity by means of a logistic growth process. The energy intake which is assumed to be proportional to the feeding rate, is allotted in constant proportions to growth plus maintenance on the one hand and reproduction on the other hand.

The energy needed for maintenance is taken to be proportional to the weight of the individual. From the proportion of energy channelled to growth plus maintenance, the maintenance requirements are covered first and the remainder is spent on tissue growth. Individuals are assumed to be capable of shrinking in size when starving for food and hence the energy spent

Table III: The *Escalator Boxcar Train* formulation of the example model, based on the description of the individual behavior given in table I and the population level dynamics specified in table II. $t+\Delta$ indicates the time at which the cohort cycle ends, while $t+\Delta^-$ and $t+\Delta^+$ indicate values just prior and afterwards, respectively.

Continuous time dynamics for the 0*th* cohort, during cohort cycle interval:	$\begin{cases} \frac{d\lambda_0}{dt} = -\delta\,\lambda_0 + \sum_{i=0}^{i=n} b(S,\mu_i^\ell)\,\lambda_i\,^\dagger \\ \frac{d\pi_0^\ell}{dt} = g(S,0)\,\lambda_0 + g_\ell(S,0)\,\pi_0^\ell - \delta\,\pi_0^\ell \\ \frac{d\pi_0^a}{dt} = \lambda_0 - \delta\,\pi_0^a \end{cases}$
Continuous time dynamics for all other cohorts, during cohort cycle interval $(i=1,\ldots,n)$:	$\begin{cases} \frac{d\lambda_i}{dt} = -q(\mu_i^a)\,\lambda_i \\ \frac{d\mu_i^\ell}{dt} = g(S,\mu_i^\ell) \\ \frac{d\mu_i^a}{dt} = 1 \end{cases}$
Renumbering equations and new initial values for cohort in creation at the end of the cohort cycle:	$\begin{cases} \lambda_1(t+\Delta^+) = \lambda_0(t+\Delta^-) \\ \mu_1^\ell(t+\Delta^+) = \frac{\pi_0^\ell(t+\Delta^-)}{\lambda_0(t+\Delta^-)} \\ \mu_1^a(t+\Delta^+) = \frac{\pi_0^a(t+\Delta^-)}{\lambda_0(t+\Delta^-)} \\ \lambda_0(t+\Delta^+) = 0 \\ \pi_0^\ell(t+\Delta^+) = 0 \\ \pi_0^a(t+\Delta^+) = 0 \end{cases}$
Renumbering equations for all other cohorts at the end of the cohort cycle $(i=1,\ldots,n-1)$:	$\begin{cases} \lambda_{i+1}(t+\Delta^+) = \lambda_i(t+\Delta^-) \\ \mu_{i+1}^\ell(t+\Delta^+) = \mu_i^\ell(t+\Delta^-) \\ \mu_{i+1}^a(t+\Delta^+) = \mu_i^a(t+\Delta^-) \end{cases}$
Dynamics of the food density in the environment:	$\frac{dS}{dt} = R(S) - \sum_{i=0}^{i=n} I(S,\mu_i^\ell)\,\lambda_i\,^\dagger$

† Include the 0*th* cohort if $\lambda_0 \neq 0$: Take $\mu_0^\ell := \pi_0^\ell/\lambda_0$

on tissue growth can essentially become negative. A constant amount of energy is needed per unit weight increase. Under constant food conditions these assumptions lead to an age–dependent growth process of the vonBertalanffy type. In this model the individual length can now be scaled in such a way that it takes values between 0 and 1. All individuals are then born with a scaled length value of 0. The model can be scaled even further by choosing the time unit equal to the characteristic time for individual growth (which is equal to the inverse of the growth rate in length) and by expressing the density of food particles in terms of the maximum amount of particles that can be eaten per unit of time by an individual of maximum length.

Because of the constant subdivision of energy intake channelled into reproduction and growth plus maintenance, reproduction is also proportional to the surface area of the individual and equally dependent upon the food concentration. The reproduction rate is the quotient of the energy input into reproduction and the, constant, energy requirement to produce one newborn individual. Newborn individuals all have the same length at birth and start reproducing immediately. The maximum reproduction rate of an individual of maximum length under very high food densities will be denoted by α.

The natural mortality rate is assumed to be independent of both age and length and constant in time. As mentioned before, all individuals die after a finite lifespan. The random mortality rate will be denoted by δ, while A_{max} will represent the maximum age that an individual can attain.

The last equation needed as ingredient of the structured population model, describes how the food source itself recovers from the grazing activity of the consumer. It is assumed that the food source grows logistically with a maximum growth rate denoted by r_m and a carrying capacity denoted by K.

The outlined individual behavior and autonomous food dynamics, which form the ingredients of the structured population model, can be described by the equations that are given in table I. These equations depend on the age and length of an individual consumer and on the current food density. They are similar to the equations in (1, Eqs.(32)), except that in the current model there is a finite lifetime.

On the basis of the modelled individual behavior, as given in table I, the dynamics of the system can now be described by a partial differential equation (PDE) for a density function $n(t, a, \ell)$, which represents the state of the consumer population, and an ordinary differential equation for the food concentration S. These differential equations have to be accompanied by the appropriate boundary and initial conditions to complete the model. The complete set of population level equations is given in table II. I will not discuss these equations any further here, the formulation of these population equations is dealt with in (3; 4; 1).

To study the example model numerically with the EBT–package, the set of equations in table II has to be reformulated using the *Escalator Boxcar Train* method into a system of ODEs and a set of renumbering equations for the changes taking place at the end of a cohort cycle (5; 6; 1). The resulting system of equations, which is referred to as the *Escalator Boxcar Train formulation* of the model, is given in table III. The equations in this table result from applying the $2nd$–order version of the method to the example model. This $2nd$–order *Escalator Boxcar Train* formulation of the model is implemented in the template file **ebttmpl.c**. This file, and the accompanying header file, **ebttmpl.h** will be discussed next. For a discussion of the theory of the *Escalator Boxcar Train* formulation in general, see (5; 6; 1).

Chapter 3

Model–dependent files

To implement a particular structured population model for study with the EBT–package, two model–dependent files have to be created from templates supplied with the package. These files define the parts of the program that are specific for the current model. All the parts of the program that are model–independent, such as the routines that handle the initialization of the program, the integration of all the statistics during a cohort cycle, the rearranging processes at the end of a cohort cycle, the generation of output and the shutdown of the program, are contained in the library files (see section 1.4.1 on page 5).

The first file to be created is a header file specifying the dimensions of the model, the second file is a program file containing the equation specifications of the model. In the following the header file will frequently be referred to as the "(model) dimension file", the program file as the "(model) definition file". (Note that in the previous version 1.7 of the package these files were usually called "problem–specific header file" and "problem–specific program file"). Both the dimension and the definition file have to have the same basename, only their extensions should be different. The dimension file, being a header file, should have a "**.h**" extension, the definition file, being a program file, a "**.c**" extension.

The two files **ebttmpl.h** and **ebttmpl.c** (see section 1.4) constitute the dimension and definition file for the example structured population model, implemented using the $2nd$–order *Escalator Boxcar Train*–method (see section 2.3 in the previous chapter). These files can be used as templates for the implementation of the $2nd$–order *EBT* formulation of any arbitrary structured population model.

The sections 3.3 and 3.5 below discuss the necessary steps to implement a particular structured population model into the EBT–package. However, before embarking on this discussion the difference between *i–state* variables and *i–state* constants (section 3.1) should be pointed out and and the distinction between parameters and constants in the model (section 3.2) should be explained. In addition, section 3.4 discusses the data structures that are defined within the EBT–package and the syntax to address them. These data structures, which are crucial to the programming of the model definition file, contain the current values and derivatives of all environmental and population variables that are present in the model. The EBT–implementation will be illustrated using the example model discussed in section 2.3, of which the $2nd$–order *Escalator Boxcar Train*–formulation is given in table III. All examples of C-coding, shown in the next sections, are hence stemming from the **ebttmpl.h** and **ebttmpl.c** file, which pertain to this particular model.

3.1 On *i–state* variables and *i–state* constants

The individual members of a physiologically structured population are characterized and distinguished from each other by a set of physiological traits. These quantities are usually referred to as *i–state* variables. The value of these variables change over time in a continuous way. Within the EBT–package the mean values of these *i–state* variables are used to characterize the cohorts of individuals, into which the population is subdivided. In addition, the EBT–package allows the cohorts to be characterized by a set of quantities that I will refer to as *i–state constants*. The fundamental difference between these two types of characteristic quantities is the way in which they change over time. The *i–state variables* change continuously over time and their dynamics are hence specified by sets of ODEs (see, for example, table III on page 11). The user has to specify these differential equations in the routine `Gradient()` (see section 3.5.7 on page 28) and the program takes care of the integration of these ODEs over time. Good examples of *i–state* variables are age and size, as used, for example, in the model of section 2.3.

The *i–state constants*, on the other hand, change in a discontinuous fashion as specified by transformations implemented by the user. Such transformations should be implemented in the routine `InstantDynamics()` (see section 3.5.10 on page 32) or, during initialization, in the routine `UserInit()` (see section 3.5.4 on page 26). The user is completely responsible for the values of these *i–state* constants and the way they change. The *i–state* constants can be viewed as temporary storage space, just like a note–pad tagged onto a cohort, in which the user can store values of interest that pertain to a particular cohort. The *i–state* constants can, for example, be used to remember the state, with which the individuals in a cohort were born.

3.2 Parameters versus constants

Another important distinction in the context of the EBT–package is between *parameters* and *constants*. Both terms refer to non–dynamic quantities that are usually called parameters in the structured population model to be studied. However, some of these quantities you will want to vary more often than others. The ease with which these quantities are varied between different integration runs, determines the distinction between parameters and constants in the EBT–implementation.

Pure parameters (or shortly parameters) are non–dynamic quantities that can be varied from one integration run to the other without the need to re–compile the program. Within the EBT–program parameters are therefore defined to be variables, the value of which is only set during initialization. The values of the parameters to be used in a particular integration run hence have to be given to the program as input, using the *control variable file* (see section 5.2.1 on page 38). The parameters in the EBT–program form an array of variables, called `parameter[]`. The model dimension file has to specify the dimension of this array, that is, how many of the non–dynamic quantities in your model are parameters and can hence be varied from one integration run to the other. This is done with the `PARAMETER_NR` constant, described in section 3.3.6 (see below).

A *constant* is a non–dynamic quantity that is defined within the model definition file as an alias for an explicit number. This constant, or rather the explicit (usually real–valued) number it stands for, figures at one or more places in the model. The definition of a constant is hence entirely restricted to the definition file and therefore changing it involves editing the file and re–building the program.

The distinction between parameters and constants is slightly academic: all non–dynamic quantities in a structured population model could be defined as either parameters or as constants. The distinction is only made to simplify implementation of a particular model. If all non–dynamic quantities are taken to be parameters, their values have to be inputted to the program during initialization via the *control variable file* (see section 5.2.1 on page 38). If all these quantities are taken to be constants, every change to such a quantity would call for a re–build of the integration program. The result is the same, but it is left to the user which way of specifying non–dynamic quantities is preferred.

3.3 The model dimension file

The model dimension file defines all the dimensions of the model that is to be implemented. The name of the model dimension file should always consist of the same basename as the model definition file (see the next section) and have a **".h"** extension. Therefore, for the example model of section 2.3, which is implemented in the model definition file **ebttmpl.c**, the EBT–package contains a model dimension file that is called **ebttmpl.h**. This file **ebttmpl.h** is an appropriate template for the dimension file of any model and I hence strongly advice you to use it as such. By editing its contents with your favorite text editor, this file has to be adapted for the particular problem you are implementing.

The model dimension file contains a total of 6 definition statements, which will be discussed in the next sections. All these definitions are obligatory, none of them may be missing. While reading through the next sections, you are advised to follow the steps as well, by looking at the template **ebttmpl.h** for the model dimension file.

3.3.1 The `POPULATION_NR` constant

This constant specifies the number of *structured* populations that are part of the model to be implemented. Definition of this constant is done in the following type of statement,

```
#define POPULATION_NR     <number>
```

which should occur in the dimension file at an arbitrary place.

In the example model described in section 2.3, only one structured population is present, *i.e.* the (age, length)–structured consumer population. Hence, in the dimension file **ebttmpl.h** the following definition is found:

```
#define POPULATION_NR     1
```

Note that it is impossible to specify 0 *structured* populations here!

3.3.2 The `I_STATE_DIM` constant

This constant specifies the number of *i–state* variables that characterize the individuals in the structured populations present in the model. This *number* of *i–state* variables is the same for the individuals of *all* populations. However, their *nature* does not have to be identical. Hence, it is entirely plausible to have a model with two structured populations, the individuals of which are characterized by the same number of *i–state* variables that have otherwise different interpretations. The user is ultimately responsible to be consistent in the model definition file (see section 3.5) with respect to the appropriate interpretation of the *i–state* variables for the different populations. The program is totally oblivious of this interpretation. If the

individuals of one of the structured populations are characterized by a smaller number of *i–state* variables, an appropriate number of dummy variables can be introduced to complete the total number of *i–state* variables.
Definition of this constant is done in the following type of statement,

```
#define I_STATE_DIM          <number>
```

which should occur in the dimension file at an arbitrary place.
In the example model described in section 2.3, the individuals of the consumer population are characterized by their age *and* by their length (see table III). Hence, in the dimension file **ebttmpl.h** the following definition is found:

```
#define I_STATE_DIM          2
```

Note that it is impossible to specify 0 *i–state* variables here!

3.3.3 The `I_CONST_DIM` constant

This constant specifies the number of *i–state* constants that characterize the individuals in the structured populations present in the model.
As with the number of *i–state variables*, this number of *i–state constants* is the same for the individuals of *all* populations, but not their nature. Again, the user is ultimately responsible to be consistent in the model definition file (see section 3.5) with respect to the appropriate interpretation of the constants for the different populations, as the program is totally oblivious of it. If the individuals of one of the structured populations are characterized by a smaller number of *i–state* constants, an appropriate number of dummy constants can be introduced to complete the total number.
Definition of this constant is done in the following type of statement,

```
#define I_CONST_DIM          <number>
```

which should occur in the dimension file at an arbitrary place.
In the example model described in section 2.3, the individuals of the consumer population are not characterized by any *i–state* constants at all. Hence, in the dimension file **ebttmpl.h** the following definition is found:

```
#define I_CONST_DIM          0
```

Observe that it is indeed possible to have 0 *i–state* constants, as opposed to *i–state* variables.

3.3.4 The `ENVIRON_DIM` constant

This constant specifies the number of environment variables (*E–state* variables) that characterize the (dynamic part of the) environment in which all individuals of the structured populations in the model live. Essentially the `ENVIRON_DIM` constant refers to the number of ODEs that have to be integrated simultaneously with the integration of the quantities that characterize the structured populations. The environment can in addition be characterized by quantities that change over time in an explicitly specified way. Examples of the latter include temperature or solar influx. These environmental characteristics do change over time, but not dynamically, that is, not in a way that is described by a differential equation. Hence they should not be counted among the *E–state* variable number that has to be specified here. It should be noted that *time* is always the *first* environmental variable within the EBT–package. The total number of *E–state* variables has to be increased accordingly.

Definition of this constant is done in the following type of statement,

```
#define ENVIRON_DIM        <number>
```

which should occur in the dimension file at an arbitrary place.

In the example model described in section 2.3, the individuals of the consumer population feed on the unstructured resource population. The concentration of food particles in the environment is therefore necessarily part of the set of *E–state* variables, together with the time as standard *E–state* variable. Hence, in the dimension file **ebttmpl.h** the following definition is found:

```
#define ENVIRON_DIM        2
```

3.3.5 The `OUTPUT_VAR_NR` constant

This constant specifies the number of quantities that have to be written to the output file (see section 5.3.2 on page 45). Quantities of interest that serve as output from a structured population model, usually comprise the values of the *E–state* variables in the model and the values of specific *measures* of the structured population, such as the total number of individuals, the total biomass of the population, or the mean size of all individuals (see (5; 6; 1)). Assigning the current values of these quantities of interest to the appropriate array of variables that will be written to the output file, is done in the routine `DefineOutput()` (see section 3.5.11 on page 33). The constant `OUTPUT_VAR_NR` tells the EBT–program how long this array of output variables should be and hence how much memory to allocate for it.

Definition of this constant is done in the following type of statement,

```
#define OUTPUT_VAR_NR      <number>
```

which should occur in the dimension file at an arbitrary place.

In the example model described in section 2.3, I have chosen the total number of consumer individuals in the structured population, their total biomass and the concentration of food particles as output quantities of interest. Hence, in the dimension file **ebttmpl.h** the following definition is found:

```
#define OUTPUT_VAR_NR      3
```

Note that it is impossible to specify 0 output variables!

3.3.6 The `PARAMETER_NR` constant

This constant specifies the number of parameters in the model. The distinction between parameters and constants in a structured population model was explained in section 3.2 on page 14. The constant `PARAMETER_NR` tells the EBT–program the size of the array of parameters that will only be specified during initialization by means of values in the *control variable file*. The EBT–program will allocate the appropriate amount of memory.

Definition of this constant is done in the following type of statement,

```
#define PARAMETER_NR        <number>
```

which should occur in the dimension file at an arbitrary place.

In the example model described in section 2.3, I have chosen to vary the quantities α, δ, r_m and K more frequently than the parameter A_{max}. Therefore, A_{max} is taken to be a constant and will be defined in the model definition file. The other quantities are chosen to be pure parameters and will hence have to be specified in the *control variable file* (see section 5.2.1 on page 38). Hence, in the dimension file **ebttmpl.h** the following definition is found:

```
#define PARAMETER_NR        4
```

Observe that it is indeed possible to have 0 parameters!

3.3.7 The `TIME_METHOD` constant

Optionally, the constant `TIME_METHOD` can be set to either `RK2`, `RK4`, `RKF45`, `RKCK` (which is the default value), or `DOPRI5`, indicating the time integration method to use:

- `RK2`: the 2nd order Runge-Kutta integration method with fixed step size,
- `RK4`: the 4th order Runge-Kutta integration method with fixed step size,
- `RKF45`: the Runge-Kutta-Fehlberg 4/5th order integration method with an adaptive step size, or
- `RKCK`: the Cash-Karp Runga-Kutta integration method with adaptive step size,
- `DOPRI5`: an explicit Runge-Kutta method of order (4)5 due to Dormand and Prince with step size control and dense output.

3.3.8 The `DYNAMIC_COHORTS` constant

This constant specifies when new cohorts are to be generate. New cohorts can be generated at the at of a cohort cyle, but also during a cohort cycle at an event specified by the user (section 3.5.8). If `DYNAMIC_COHORTS` equals 1, new cohorts are only formed whenever the return value of the function `ForceCohortEnd()` equals 1. If `DYNAMIC_COHORTS` equals 0, new cohorts are formed whenever the return value of the function `ForceCohortEnd()` equals 1, in addition to the usual formation of new cohorts at the end of a cohort cycle.

NB: For dynamic cohort closure the `DOPRI5` method is needed. Therfore, `TIME_METHOD` will automatically be changed to `DOPRI5` if `DYNAMIC_COHORTS` is set to 1.

3.4 Data structures and addressing syntax

The data structures within the EBT–program can be classified into (1) data representing the state of the environment, (2) data representing the state of the internal cohorts in every population, (3) data representing the state of the boundary population cohorts in every population and (4) the derivatives of all these quantities. In the following sections the addressing syntax for these data structures will be discussed one by one. The syntax explained should be used throughout the entire model definition file, as it is identical within any of the routines discussed in later sections.

3.4.1 The environment variables

The current values of the environmental variables are always stored within an array of real valued numbers. The name of this array is `env[]` and at any time its length equals `ENVIRON_DIM`, as specified in the model dimension file. The i*th* environmental variable is simply addressed as `env[i]`, with $0 \leq i <$ `ENVIRON_DIM`.
The interpretation of the individual environmental variables is for the largest part up to the user. Only the 0*th* environmental variable `env[0]` is always fixed to be the time, the program is oblivious to the meaning of the other environmental variables. The user is responsible for a consistent interpretation.

3.4.2 The cohort statistics

The special property of the program resides in the subdivision of the populations into cohorts of more or less similar individuals, which are characterized by a set of statistics. A second important feature is that the number of these cohorts present can change over time. For these two reasons a specific layout has been chosen for the population data inside the program. This layout will be the topic of this and the following sections.
Every cohort of individuals in a particular structured population is represented inside the program by a three-dimensional array, `pop`. The syntax for addressing the individual cohorts in a particular population will be discussed later on in this section.
Every `pop` array contains a variable that bears the name

```
number
```

the value of which indicates the total number of individuals within the cohort it represents. In the same way, every cohort contains a number of variables, to be precise `I_STATE_DIM` quantities in total, which equal the mean value of the different *i–state* variables that characterize the individuals within the particular cohort. Within the `pop` array these `I_STATE_DIM` variables have the following names:

```
i_state(0)
i_state(1)
i_state(2)
    ⋮
i_state(I_STATE_DIM-2)
i_state(I_STATE_DIM-1)
```

Table I: The elements of the arrays `pop` and `popIDcard`. The elements represent the (physiological) quantities that characterize the individuals within a single cohort of a structured population. Addressing the quantities should be done using the variable names shown. `n`: population index, `i`: cohort index, `ISD`: value of the constant `I_STATE_DIM` (section 3.3.2), `ICD`: value of the constant `I_CONST_DIM` (section 3.3.3).

Array element	Refers to:
`pop[n][i][number]`	Number of individuals within the cohort.
`pop[n][i][i_state(0)]`	Mean of the 0*th* *i–state* variable.
`pop[n][i][i_state(1)]`	Mean of the 1*st* *i–state* variable.
`pop[n][i][i_state(2)]`	Mean of the 2*nd* *i–state* variable.
⋮	
`pop[n][i][i_state(ISD-2)]`	Mean of the one–to–last *i–state* variable.
`pop[n][i][i_state(ISD-1)]`	Mean of the last *i–state* variable.
If I_CONST_DIM $\neq 0$:	
`popIDcard[n][i][i_const(0)]`	Value of the 0*th* *i–state* constant.
`popIDcard[n][i][i_const(1)]`	Value of the 1*st* *i–state* constant.
`popIDcard[n][i][i_const(2)]`	Value of the 2*nd* *i–state* constant.
⋮	
`popIDcard[n][i][i_const(ICD-2)]`	Value of the one–to–last *i–state* constant.
`popIDcard[n][i][i_const(ICD-1)]`	Value of the last *i–state* constant.

If the constant `I_CONST_DIM` is defined unequal to 0, a second array is created, `popIDcard`. This array contains the current value of the *i–state* constants for the particular cohort. Exactly `I_CONST_DIM` elements in the array `popIDcard` belong to a single cohort, and these are labelled with the following names:

```
i_const(0)
i_const(1)
i_const(2)
          ⋮
i_const(I_CONST_DIM-2)
i_const(I_CONST_DIM-1)
```

This naming convention is summarized in table I. The elements of a cohort array should always be referred to using this naming convention, irrespective of the structured population that the cohort is part of, and independent of the routine in which the quantity is addressed.

The internal population cohorts

The internal cohorts of any structured population in the model can be accessed within any of the routines in the model definition file, using the three–dimensional array called `pop[][][]`. How to address a specific quantity that characterizes the i*th* cohort in the n*th* population, is shown in table I on page 20. For example, `pop[1][5][number]` refers to the total number of individuals in the 6*th* cohort within the 2*nd* structured population in the model (Note that in the C–language the lowest array index equals 0).

The total number of structured populations in the model (equal to the maximum value of `n`, plus 1) is of course explicitly defined in the model dimension file with the constant `POPULATION_NR`. The number of cohorts in the different structured populations can, however, vary over time. In all routines a variable can therefore be accessed which specifies the number of internal cohorts in each structured population. This variable is an array of integer values, called `cohort_no[]`, with a length equal to the value of `POPULATION_NR`. `cohort_no[n]` specifies the number of internal cohorts that are present in the n*th* population at a particular time. Note that this array is defined as a global variable and does not occur in the headers of the routines. Acceptable values for the indices `n` and `i` in the expression `pop[n][i][]` therefore are: 0 ≤ `n` < `POPULATION_NR` and 0 ≤ `i` < `cohort_no[n]`, respectively.

In summary, to address the quantities within a specific internal cohort of a particular population use the following syntax:

- Use `pop[n][i][number]` to address the total number of individuals in the i*th* cohort of the n*th* population.
  (0 ≤ `n` < `POPULATION_NR`, 0 ≤ `i` < `cohort_no[n]`)
- Use `pop[n][i][i_state(k)]` to address (the mean of) the k*th* *i–state* variable in the i*th* cohort of the n*th* population.
  (0 ≤ `n` < `POPULATION_NR`, 0 ≤ `i` < `cohort_no[n]`, 0 ≤ `k` < `I_STATE_DIM`)
- Use `popIDcard[n][i][i_const(k)]` to address (the value of) the k*th* *i–state* constant in the i*th* cohort of the n*th* population.
  (0 ≤ `n` < `POPULATION_NR`, 0 ≤ `i` < `cohort_no[n]`, 0 ≤ `k` < `I_CONST_DIM`)

The order of the internal cohorts in a structured population is not straightforward. It is absolutely wrong to assume that the cohort with index 0 is one of the cohorts that has been most recently added to the population. The EBT–program orders the cohorts in each population according to the value of their mean *i–state* variables, *but in descending order*. That is, the cohort with index 0 in a particular population actually has the largest value for the quantity `i_state[0]`. If two cohorts are equal in this first *i–state* variable, the second *i–state* variable is taken as the ordering key, and so on. However, it is not good practice to rely on this specific ordering of the cohorts. Although the program tries to keep the order intact, the user has so much opportunity to change values of *i–state* variables that the described order cannot be assured without sorting all the cohorts anew every time step. This operation is very time consuming and hence has not been implemented.

The boundary population cohorts

The boundary cohorts of any structured population in the model can be accessed within any of the routines in the model definition file, using the three–dimensional array called `ofs[][][]`. How to address a specific quantity that characterizes the i*th* boundary cohort in the n*th* population, is shown in table II on page 22. For example, `ofs[0][2][number]` refers to the total number of individuals in the 3*rd* boundary cohort within the 1*st* structured population in the model (Note that in the C–language the lowest array index equals 0). The addressing of the boundary cohorts in the model is hence completely analogous to the addressing of the internal cohorts, as described in the previous section.

Table II: The elements of the arrays `ofs` and `ofsIDcard` characterize a single boundary cohort. These arrays are completely analogous to the arrays `pop` and `popIDcard` (table I). `n`: population index, `i`: cohort index, `ISD`: value of the constant `I_STATE_DIM` (section 3.3.2), `ICD`: value of the constant `I_CONST_DIM` (section 3.3.3).

Array element	Refers to:
`ofs[n][i][number]`	Number of individuals within the cohort.
`ofs[n][i][i_state(0)]`	Mean of the 0*th* *i–state* variable.
`ofs[n][i][i_state(1)]`	Mean of the 1*st* *i–state* variable.
`ofs[n][i][i_state(2)]`	Mean of the 2*nd* *i–state* variable.
⋮	
`ofs[n][i][i_state(ISD-2)]`	Mean of the one–to–last *i–state* variable.
`ofs[n][i][i_state(ISD-1)]`	Mean of the last *i–state* variable.
If I_CONST_DIM $\neq 0$:	
`ofsIDcard[n][i][i_const(0)]`	Value of the 0*th* *i–state* constant.
`ofsIDcard[n][i][i_const(1)]`	Value of the 1*st* *i–state* constant.
`ofsIDcard[n][i][i_const(2)]`	Value of the 2*nd* *i–state* constant.
⋮	
`ofsIDcard[n][i][i_const(ICD-2)]`	Value of the one–to–last *i–state* constant.
`ofsIDcard[n][i][i_const(ICD-1)]`	Value of the last *i–state* constant.

Acceptable values for `n`, the number of the structured population in the expression `ofs[n][i][]`, depend on the definition of the constant `POPULATION_NR` in the model dimension file: 0 $\leq$ `n` $<$ `POPULATION_NR`. As was the case for the internal cohorts, the number of boundary cohorts in the different structured populations in the model can vary over time. Therefore, in all routines, a variable can be accessed which specifies the number of boundary cohorts in each structured population. This variable is an array of integer values, called `bpoint_no[]`, with a length equal to the value of `POPULATION_NR` (compare the global variable `cohort_no[]`, discussed in the previous section). `bpoint_no[n]` specifies the number of boundary cohorts that are present in the n*th* population at a particular time. The array `bpoint_no[]` is defined as a global variable and hence does not occur in the headers of the routines. Acceptable values for the index `i` in the expression `ofs[n][i][]` hence are: 0 $\leq$ `i` $<$ `bpoint_no[n]`.

In summary, to address the quantities within a specific boundary cohort of a particular population use the following syntax (see also the previous section):

- Use `ofs[n][i][number]` to address the total number of individuals in the i*th* boundary cohort of the n*th* population.
  (0 $\leq$ `n` $<$ `POPULATION_NR`, 0 $\leq$ `i` $<$ `bpoint_no[n]`)
- Use `ofs[n][i][i_state(k)]` to address the value of the k*th* *i–state* variable in the i*th* boundary cohort of the n*th* population.
  (0 $\leq$ `n` $<$ `POPULATION_NR`, 0 $\leq$ `i` $<$ `bpoint_no[n]`, 0 $\leq$ `k` $<$ `I_STATE_DIM`)
- Use `ofsIDcard[n][i][i_const(k)]` to address (the value of) the k*th* *i–state* constant

in the i*th* boundary cohort of the n*th* population.
(0 ≤ `n` < `POPULATION_NR`, 0 ≤ `i` < `cohort_no[n]`, 0 ≤ `k` < `I_CONST_DIM`)

Addressing the derivatives

The derivatives of the various variables are always labelled with the name of the variable itself followed by the addition `grad`. Hence, the derivatives of the environment variables, the statistics of the internal population cohorts and the statistics of the boundary population cohorts are contained in data structures, called `envgrad[]`, `popgrad[][][]` and `ofsgrad[][][]`, respectively. The structure and addressing of these derivatives is completely equivalent with those of the variables themselves. For example:

- The derivative of the i*th* environment variable is accessed by `envgrad[i]`.
  (0 ≤ i < `ENVIRON_DIM`)
- The derivative of the k*th* *i–state* variable in the i*th* cohort of the n*th* population is accessed by `popgrad[n][i][i_state(k)]`.
  (0 ≤ `n` < `POPULATION_NR`, 0 ≤ `i` < `cohort_no[n]`, 0 ≤ `k` < `I_STATE_DIM`)
- The derivative of the total number of individuals in the i*th* boundary cohort of the n*th* population is accessed by `ofsgrad[n][i][number]`.
  (0 ≤ `n` < `POPULATION_NR` and 0 ≤ `i` < `bpoint_no[n]`)

3.5 The model definition file

The model definition file has to implement the equations that define the structured population model under study. In essence the definition file is the EBT–implementation of the particular structured population model. The name of the definition file should always have the same basename as the dimension file (see the previous section) and have a **".c"** extension. Therefore, for the example model of section 2.3, the EBT–package contains a model definition file that is called **ebttmpl.c**. The accompanying model dimension file for the model (**ebttmpl.h**) was discussed in section 3.3. This file **ebttmpl.c** is an appropriate template for the definition file of any model and should hence be copied and renamed freely. By editing with your favorite text editor, this file has to be modified into the definition file for the particular problem you want to study.

Apart from a single statement to include the general header file **escbox.h**, which defines the necessary data structures, the main purpose of the model definition file is the implementation of 7 routines that are model–dependent. These routines are to be discussed in the sections 3.5.4–3.5.11 below. The routines specify the number and nature of the boundary cohorts, the continuous and instantaneous dynamics of all variables and the output of the program.

As mentioned, the template file **ebttmpl.c** contains a fully programmed set of routines, which implement the 2*nd*–order version of the *Escalator Boxcar Train*–formulation of the example model, described in section 2.3. The body of the routines in this file can easily be adapted, *but the headers of these routines should not be changed at all.*

N.B.: The functioning of the entire EBT–package depends crucially on the correct definition of the routine headers in the model definition file. Therefore, the user should only make changes to the body of the routines in the template file, that is, the part of the routine between its opening and closing brace.

Within the routines in the model definition file, the syntax of addressing the various data structures that contain all the information about the structured populations and their environment, is completely identical. This syntax for addressing the data was discussed in section 3.4 above.

Although the programming of the 7 routines is the essential part of the definition file, the implementation is often easier when the *i–state* variables and *i–state* constants and the parameters and other constants in the model are labelled with more meaningful names. This "aliasing" is discussed in the sections 3.5.2 and 3.5.3. It should be stressed that this aliasing is completely optional, but often very handy.

While reading through the next sections, you are advised to follow the steps also by looking at the template **ebttmpl.c** for the model definition file. All illustrations with lines of program code are taken from this template file. Implementing a particular structured population model following the steps outlined below and using the template file as the backbone, will in my opinion only require a minimal working knowledge of the C–programming language. A basic knowledge from structured population models and their formulation seems to me at least more essential.

3.5.1 Including the header file escbox.h

The first functional line of C–code in the model definition file should include the general header file **escbox.h**. This is done using the following statement:

```
#include  "escbox.h"
```

Including this header file takes care of defining all the necessary dimensions, the data structures (among them the cohort arrays) and the globally accessible variables that are to be used in the remainder of the file.

3.5.2 Labelling *i–state* variables and constants

To make the programming of subsequent routines easier, it is possible (and advisable!) to label *i–state* variables and constants with more meaningful names. All *i–state* variables and constants can be given more meaningful names, but it is impossible to assign *different* names to the *same* statistic (for example, the *i–state* variable with index 0) in *different* structured populations in the model.

Labelling variables is done using statements of the form:

`#define` *<name of* i–state *variable k>* `i_state(k)`

For example,

```
#define  age          i_state(0)
#define  length       i_state(1)
#define  birthday     i_const(0)
```

These definitions can simplify the addressing of the various statistics in the structured populations. For example, given the definitions above the following variable references are synonymous with each other and can hence be used interchangeably in the subsequent routines:

- `pop[n][i][age]` and `pop[n][i][i_state(0)]`,
- `ofsgrad[n][i][age]` and `ofsgrad[n][i][i_state(0)]`,

- `popgrad[n][i][length]` and `popgrad[n][i][i_state(1)]`,
- `ofs[n][i][length]` and `ofs[n][i][i_state(1)]`,
- `popIDcard[n][i][birthday]` and `popIDcard[n][i][i_const(0)]`,
- `ofsIDcard[n][i][birthday]` and `ofsIDcard[n][i][i_const(0)]`,

(The *i–state* constant `birthday` (or `i_const(0)`) could, for instance, be used to store for every cohort the time that it was born).

Obviously, labelling *i–state* variables and constants, but also other constants and parameters (see the next section), promotes a consistent interpretation of the various variables in the model.

In the template file **ebttmpl.c** the only two *i–state* variables are identified with the physiological traits *age* and *length*, respectively, by including the statements:

```
#define  age       i_state(0)
#define  length    i_state(1)
```

3.5.3 Defining constants and labelling parameters

As explained in section 3.2 on page 14 the EBT–package distinguishes *constants* and *pure parameters*, depending on where the values of these quantities is specified. In turn, the place where these non–dynamic quantities are specified determines how easily they can be modified between integration runs (see for an elaborate explanation section 3.2).

Constants are essentially explicit (real–valued) numbers within the model definition file that occur at some point in the structured population model. Specification of these constants can be done using statements of the form:

`#define` *<constant name>*   *<value>*

(It is good practice to use only uppercase letters in the constant name to avoid confusion with variables).

In the template file **ebttmpl.c** the quantity A_{max} is treated as a constant and can hence only be changed by editing the model definition file and re–building the executable program. The constant A_{max} is identified with the value 100. In addition, a second constant `MAX_COHORT_SIZE` is defined which will be used in the routine `EventLocation()` (see section 3.5.8 below) to restrict the maximum number of individuals ending up in a "cohort in creation" to 5 individuals. These two constants are defined by the statements:

```
#define  AMAX             100
#define  MAX_COHORT_SIZE  5.0
```

Parameters are non–dynamic quantities, the value of which is only specified during initialization of the integration program using the *control variable file* (see section 5.2.1 on page 38). They can therefore be varied between integration runs without re–building the executable program. The parameters constitute an array of real–valued numbers, called `parameter[]`, with length equal to the constant `PARAMETER_NR` (see section 3.3.6 on page 17). Similar to what was explained in the previous section for *i–state* variables, and constants, the parameters can be given more meaningful names by using statements of the form:

```
#define   <name of parameter #n>       parameter[n]
```

Defining these synonyms greatly promotes a consistent interpretation of the elements of the array `parameter[]`.

In the example model that is implemented in the file **ebttmpl.c** the quantities α, δ, r_m and K (see the model description in section 2.3) are taken as parameters and are assumed to be the first, second, third and fourth element of the array `parameter[]`, respectively. This specific identification of the various parameters with the elements of the array `parameter[]` entirely depends on the convention adopted by the user, which should be used consistently throughout the model definition and the *control variable file* (see section 5.2.1 on page 38).

Within the **ebttmpl.c** file the parameters α, δ, r_m and K are referred to by the names `ALPHA`, `DELTA`, `R` and `K`, respectively. Hence, the identification of the array elements `parameter[]` is made explicit in the following `#define` statements:

```
#define  ALPHA     parameter[0]
#define  DELTA     parameter[1]
#define  RM        parameter[2]
#define  K         parameter[3]
```

The parameters can now be referred to using either the appropriate element from the array `parameter[]` or the name with which it was declared synonymous in the `#define` statements above.

3.5.4 The `UserInit()` routine

After the integration program has started and the *control variable file* (see section 5.2.1) and *initial state file* (see section 5.2.2) have been read, the routine `UserInit()` is called by the main program. The routine allows the user to carry out some specific initializations. For example, it is possible within this routine to process possible command–line arguments (other than the obligatory one, see chapter 5). Other examples include transformations of the environmental and/or population data and the initialization of possible global variables defined by the user.

As an important point it must be noted that, although it is possible to change the value of all environmental and population variables, it is *not* possible to change their *number*. Therefore, it is not possible to add new cohorts to a population in this routine. (You can, of course, always initialize a population with a large number of cohorts with arbitrary contents following section 5.2.2, define the contents for the required number of cohorts in this routine and set the number of individuals in the unused cohorts to 0.0. These latter will be removed automatically.)

As seen in the file **ebttmpl.c**, the body of this routine `UserInit()` can also be empty. In the example model, no user–specified initialization is carried out.

3.5.5 The `SetBpointNo()` routine

As described in (1), the boundary cohorts (or "cohorts in creation") of each structured population are characterized by special points, the so–called boundary points. Up to now I have only encountered models with structured populations, in which all individuals are born equal. In this simplified case, the number of boundary cohorts at every moment in time equals one and the point characterizing this cohort equals the state that all individuals are born with. However, the EBT–package is more general and can handle multiple boundary cohorts (see

(1)). This and the following section describe the two routines in the model definition file that specify the number and value of the boundary points for each structured population.

As described in section 2.1 and 2.2 the integration takes place in a cyclical fashion, during which the boundary cohorts of a population receive a continuous inflow of individuals that are born during the integration cycle. At the end of the cycle the existing boundary cohorts are transformed into internal cohorts and a new set of boundary cohorts is formed. The boundary points are involved in these transformations into internal cohorts. The routine `SetBpointNo()` is called to determine how many new boundary cohorts have to be set up at the beginning of the coming cohort cycle. The user has to specify this number for each of the structured populations in the model, using the array `bpoint_no[]`. The length of this array obviously equals `POPULATION_NR`. Therefore, the element `bpoint_no[n]` of this array should be set equal to the required number of boundary cohorts for population `n` during the next cohort integration cycle.

Since the example model, implemented in **ebttmpl.c**, is one of this most common class of models, in which all individuals are born equal, the number of boundary cohorts required during a cohort cycle equals 1. Hence the one and only line of C–code encountered in the routine `SetBpointNo()` looks like:

```
bpoint_no[0]=1;
```

3.5.6 The `SetBpoints()` routine

The routine `SetBpoints()` is called by the main integration program after the requested number of boundary cohorts have been created for each structured population. The routine is used to define the value of the boundary points, characterizing these cohorts. For an extensive discussion of these boundary points refer to (1). In this manual only the general implementation will discussed and, for the example model, the specification of the boundary point in the simplified case that all individuals are born equal.

The variable that has to be defined in this routine is called `bpoints[][][]` and has exactly the same structure as the variables `pop[][][]` and `ofs[][][]` described in section 3.4.2 above. Therefore, this routine should assign values to all the quantities `bpoints[n][i][i_state(j)]`, for all values of `n`, `i` and `j` (0 ≤ `n` < `POPULATION_NR`, 0 ≤ `i` < `bpoint_no[n]`, 0 ≤ `j` < `I_STATE_DIM`). Since a point characterizing a boundary cohort is fully identified with the values of the *i–state*s, there is no analogue of the arrays `popIDcard[][][]` and `ofsIDcard[][][]` for the boundary cohort.

As mentioned in the previous section, the class of models in which all individuals are born with an equal state is rather special. In this case, there is only one boundary cohort (see previous section) and the unique state at birth can simply be taken as the boundary point. The example model is a member of this class and, as explained in section 2.3, individuals are born with age and length at birth both equal to 0. Therefore, the body of this routine in the file **ebttmpl.c** contains the line of code:

```
bpoints[0][0][age]=0.0; bpoints[0][0][length]=0.0;
```

which is equivalent to:

```
bpoints[0][0][i_state(0)]=0.0; bpoints[0][0][i_state(1)]=0.0;
```

given the alias definitions of section 3.5.2.

3.5.7 The `Gradient()` routine

In the routine `Gradient()` the user has to specify the derivatives of all the environmental and population statistics of which the dynamics are described by ODEs. As explained in section 2.1, the exact form of these ODEs can be found in or at least inferred from (5; 6; 1). These ODEs heavily depend on the functions describing the growth, birth and death processes of the individuals in the problem under study. The programming of this routine `Gradient()` certainly requires a basic knowledge of the C programming language.

The derivatives of the environmental variables, whose values at any specific time are given by the variable `env[]`, have to be assigned to the elements of the variable `envgrad[]`. The derivatives of the statistics (total number, mean *i–state*) of all the internal cohorts in the various populations have to be assigned to the elements of the variable `popgrad[][][]`. The actual values of these statistics are contained in the variable `pop[][][]`. Similarly, the derivatives of the statistics (total number, mean *i–state*) of all the boundary cohorts in the populations have to be assigned to the elements of the variable `ofsgrad[][][]`. The actual values of these quantities are given by the variable `ofs[][][]`. The complete addressing of all these variables and their elements has been discussed extensively in section 3.4 above.

For the example model that is implemented in **ebttmpl.c** the complete EBT–formulation is given in table III on page 11. The routine `Gradient()` implements all ODEs that are specified in this table. In the following the implementation of `Gradient()` in **ebttmpl.c** will be discussed step–by–step.

The first step implemented computes the current value of the functional response $\frac{S}{(1+S)}$ that occurs in the functions describing individual feeding, growth and reproduction in the model (see table I on page 9). This computation is carried out by the following statement:

```
functional_response = env[1]/(1+env[1]);
```

(Note that "time" is always the first environment variable `env[0]`, while in this model `env[1]` represents the food density in the environment).

The second step is to compute the total population feeding and reproduction rate at the current time. In the analytic formulation of the population model, these total population rates are equivalent to integrals over the population density function, weighted with the individual feeding and reproduction rate, respectively (see table II on page 10). In the EBT–formulation of the model, these total population rates are given by summing the contribution of all the individual cohorts to the feeding and reproduction, respectively (see table III on page 11). Since both the individual feeding and reproduction rate are proportional to the individual surface area, which is identified with the squared individual length, the total surface area of the entire population is computed as a common factor:

```
for(i=0, total_area=0; i<cohort_no[0]; i++)
  {
    total_area +=
        pow(pop[0][i][length], 2.0)*pop[0][i][number];
  }
```

The implemented loop over the index `i` ($0 \leq$ `i` $<$ `cohort_no[0]`) ensures that all internal cohorts are included. As is also indicated in table III, the contribution of the boundary cohort should be included, if it is not empty:

```
if(ofs[0][0][number] > 0.0)
  {
    mean_offspring_length =
        ofs[0][0][length]/ofs[0][0][number];
    total_area +=
        pow(mean_offspring_length, 2.0)*ofs[0][0][number];
  }
```

(Note that the quantity π_0^ℓ is first converted into the mean length of the newborn individuals present at the time, which is the equivalent quantity characterizing an internal cohort.)

The next step deals with the specification of the derivatives of the environmental variables. The first of these derivatives is the time derivative. The second is the derivative of the food density, which equals the difference between the logistic growth rate in food density and the total food consumption rate. This total consumption rate is the product of the functional response and the total population surface area, as computed before:

```
envgrad[0] = 1;
envgrad[1] = RM*env[1]*(1-env[1]/K)
    - functional_response*total_area;
```

Subsequently, the derivatives of the number, mean age and mean length of the individuals in every internal cohort is defined, as given in table III on page 11:

```
for(i=0; i<cohort_no[0]; i++)
  {
    popgrad[0][i][number] = -DELTA*pop[0][i][number];
    popgrad[0][i][age]    = 1;
    popgrad[0][i][length] =
        functional_response - pop[0][i][length];
  }
```

The last part of the routine contains the definition of the ODEs for the quantities characterizing the only boundary cohort of the population, which are also given in table III on page 11. The total population birth rate, which equals the product of the parameter α, the functional response and the total population surface area, appears in the derivative for the total number of individuals in the boundary cohort (see table 2.1 for the form of the ODEs):

```
ofsgrad[0][0][number] = -DELTA*ofs[0][0][number]
    + ALPHA*functional_response*total_area;
ofsgrad[0][0][age]    = -DELTA*ofs[0][0][age]
    + ofs[0][0][number];
ofsgrad[0][0][length] = -DELTA*ofs[0][0][length]
    + functional_response*ofs[0][0][number]
        - ofs[0][0][length];
```

The specification of the various derivatives is thus completed. It should be noted that the transformations listed in table III on page 11 that deal with the renumbering of all the cohorts and the initialization of the new "cohorts in creation" are implemented in the library files. The user does not have to deal with these transformations at all, other than specifying the value of the boundary point (*i.e.* the individual state at birth) as is described in section 3.5.6 on page 27.

3.5.8 The `EventLocation()` routine

As is described in section 2.1 and 2.2, the entire integration time is divided up into *cohort integration cycles* (see also (5; 6; 1)). The main event that characterizes the end of a cohort integration cycle is the transformation of the current boundary cohorts of each structured population into internal cohorts and the subsequent starting up of a new set of boundary cohorts. Essentially, the length of the *cohort integration cycle* is fixed and determined by a quantity that is read as input from the *control variable file* (see section 5.2.1). However, the current version of the EBT–package allows for a dynamic cohort closure, that is, the end of a cohort cycle can dynamically be enforced during the integration. The routine `EventLocation()` is used for this purpose.

When the DOPRI5 integration method is being used, the routine named `EventLocation()` is called to check whether a discontinuity in the time derivatives or an event has occurred. Such a discontinuity or event is indicated by a zero (0) return value of this routine. Hence, every time the return value changes from positive to negative or vice versa, this is interpreted as an event occurrence. If an event occurs, the integration will not step over the time point of the event, but will localize the moment of its occurrence by means of a root finding procedure. This localization will stop when it has found a time, for which the return value of this routine is within the tolerance bounds, distinguishing values from zero, as set in the CVF file. After the event time has been localized, integration will be carried out exactly up to the moment of event occurrence. Subsequently the routine `ForceCohortEnd()` (see section 3.5.9) is called and the integration is started from the moment of the event location onwards. If no event location is required, this routine should return a constant, negative value. The macro `NO_EVENT` can be used for this purpose.

At the end of the iterative process the most appropriate time integration step has been found and this step length is used to integrate all ODEs exactly up to the time point at which the cohort cycle should end (This integration exactly until the time that a cohort cycle should end, is also done when the cohort cycle ends for one of the other reason, mentioned below). After this final integration step, the boundary cohorts are closed off and the program carries out the usual steps that follow the end of a cohort cycle (see section 2.2).

Switching problems

Models that use the feature of dynamic cohort closure will often be referred to as *switching problems*, since the switch in sign from negative to positive of the result of `EventLocation()` determines the time of closing the cohorts. In these switching problems, a cohort integration cycle is hence ended:

1. if `DYNAMIC_COHORTS=0`: at regularly spaced points in time that are separated from each other by an interval, the length of which is controlled by the user (the "default length" of the cohort cycle; see section 5.2.1),
2. if `DYNAMIC_COHORTS=0` or if `DYNAMIC_COHORTS=1`: whenever the output of the defined output quantities (see section 3.5.11) or the output of the total state of the system (see section 5.2.1 and 5.3.4) is required, and
3. if `DYNAMIC_COHORTS=0` or if `DYNAMIC_COHORTS=1`: whenever the result of the routine `(EventLocation)` changes sign from negative to positive.

In non–switching problems this last type of event does not occur and a cohort cycle can only end because of one of the first two reasons.

N.B.: The distinction whether a problem is a switching or a non–switching problem is only made at the moment that the returned result of `EventLocation()` changes sign from negative to positive for the first time. In a non–switching problem the routine `EventLocation()` should therefore always return a *negative* value.

To facilitate the implementation of problems, in which dynamic cohort closure does not occur at all, a constant `NO_EVENT` has been defined that disables dynamic cohort closure and can be returned as the result of `(EventLocation)`. Therefore, in a non–switching problem the routine `(EventLocation)` should only contain the line:

```
return NO_EVENT;
```

Essentially, the example model presented in section 2.3 is not a switching problem. For illustrative purposes, however, the model has been turned into a switching problem by requiring that the maximum number of newborn individuals in the only boundary cohort is not larger than the limit defined as `MAX_COHORT_SIZE` (currently set to the value of 5 individuals; see section 3.5.3). The routine `EventLocation()` hence contains the following lines:

```
result = ofs[0][0][number] - MAX_COHORT_SIZE;
for (i=1; i<bpoint_no[0]; i++)
    result = max(result, ofs[0][i][number] - MAX_COHORT_SIZE);

return result;
```

In the first statement, the variable `result` is initialized to the difference between the current and the maximum allowable number (`MAX_COHORT_SIZE`) of individuals in the first boundary cohort. Subsequently, this quantity is compared with the same difference in all other boundary cohorts and the maximum of these values is selected (that is, the most positive or least negative value encountered). This maximum is returned as result of the `EventLocation()` routine. Obviously, the implemented loop structure is superfluous, since in this model there is only one boundary cohort present at any time. The loop is implemented for purely illustrative reasons.

- The iterative search for the appropriate cohort end is computationally intensive, resulting in long program execution times. Switching should hence as much as possible be avoided. If necessary, it is advisable to specify a rather large tolerance level (in the order of 10^{-3}–10^{-5}) in the *control variable file* (see section 5.2.1). This will keep the integration time within bounds.

3.5.9 The `ForceCohortEnd()` routine

When the DOPRI5 integration method is being used, the routine named `ForceCohortEnd()` is called after an event has been localized on the basis of the `EventLocation()` routine defined above. This routine should then return either `COHORT_END (=1)` or `NO_COHORT_END (=0)`, indicating whether a cohort closure is currently required or not. If the macro `DYNAMIC_COHORTS` in the program header file is set to 0 (its default value) closing off boundary cohorts and transforming them into internal cohorts takes place both at constant time intervals (as specified by the cohort cycle limit variable in the .CVF file) and whenever the current routine returns `COHORT_END`. This allows for dynamic closure of boundary cohorts, for instance, to ensure a limited number of individuals per cohort. If `DYNAMIC_COHORTS` is set to 1 in the program header file, closing off of boundary cohorts is exclusively performed when the current routine returns `COHORT_END`.

Remember: for dynamic cohort closure the `DOPRI5` integration method (section 3.3.7) is needed. If `DYNAMIC_COHORTS` is set to 1, `TIME_METHOD` will automatically be changed to `DOPRI5` (see also section 3.3.8).

3.5.10 The `InstantDynamics()` routine

In the routine `InstantDynamics()` the user can specify the instantaneous dynamics that take place at the end of a cohort integration cycle. The environmental variables contained in `env[]`, the internal cohort variables of all populations, contained in `pop[][][]`, and the boundary cohort variables, contained in `ofs[][][]`, can instantaneously be given a specific value. This allows the user to carry out a variety of discrete changes to the state of the system. Two examples will be discussed.

The first example concerns a finite lifespan for the individuals in a structured population. If all individuals of a population experience some form of mortality during their entire life, but will die with certainty upon reaching a maximum age, all the cohorts in that population can be tested on the mean age of its individuals. When this mean age exceeds the maximum age, the number of individuals in the specific cohort can instantaneously be set to 0.

The second example is a pulsed reproduction process. If reproduction in a population occurs as discrete, equally spaced events in time, the length of the cohort interval, specified by the user in the *control variable file* (see section 5.2.1), can be chosen equal to the inter–reproductive period, such that the transformation of the boundary cohorts takes place exactly after the moment of reproduction. The pulsed reproduction process can then be implemented by instantaneously setting the values of the boundary cohort variables equal to the number and the mean *i–state*of the newborns at the end of a cohort integration cycle.

Many more examples of instantaneous dynamics can be found and implemented in this routine. It should be noted, however, that the instantaneous dynamics take place *after* the transformation of the boundary statistics, but *before* the incorporation of the boundary cohorts into the collection of internal cohorts. Hence, the boundary cohorts are already characterized by the same quantities (total number of individuals, mean *i–state*) that represent an internal cohort. For example, if a pulsed reproduction process is implemented, the quantities, representing the boundary cohort, should exactly equal the state at birth of the newborn individuals.

In the example model that is implemented in **ebttmpl.c** the routine `InstantDynamics()` implements the immediate death on reaching the maximum lifespan A_{max}. Inside the following loop:

```
for(i=0; i<cohort_no[0]; i++)
  {
    if(pop[0][i][age] > AMAX) pop[0][i][number] = 0;
  }
```

the mean age of every internal cohort (not the boundary cohorts!), numbered from 0 to `cohort_no[0]-1`, is compared with the value of A_{max}. If larger, the number of individuals in the particular cohort is set to 0. Theoretically only one cohort could have passed the maximum lifespan after each cohort cycle. Nonetheless, for a correct implementation of the instantaneous death all the cohorts are tested, because assumptions about the ordering of the cohorts within the population cannot be made safely. The user should, for example, not assume that the cohort with the highest number (`cohort_no[0]-1`) in a particular population also has the largest mean age. As described in section 3.4.2, the cohorts with the lower indices are the ones with the larger mean ages. However, their order again depends upon the value of the other *i–state* variables as well.

3.5.11 The `DefineOutput()` routine

The dynamics of a structured population model are usually captured in terms of measures of the state of the environment and the structured populations in the model. The routine `DefineOutput()` should define these quantities of interest that are subsequently stored into the output file (see section 5.3.2). The output of results is carried out at regular time intervals, the length of which is defined by a variable that is specified in the *control variable file* (see section 5.2.1). Output is only allowed immediately after the sets of boundary cohorts have been incorporated into the structured populations and hence the program will always transform the boundary cohorts into internal cohorts when output is required. Therefore, only the environmental variables, again contained in the variable `env[]`, and the population variables, addressed using the variable `pop[][][]`, can be used for this definition. The output quantities should be stored in the array `output[]`. The length of this array is equal to the value of the constant `OUTPUT_VAR_NR`, as specified in the model dimension file (see section 3.3.5). The current time is automatically written to the output file before all the quantities defined in the routine `DefineOutput()`.

As already mentioned in section 3.3.5 on page 17, the quantities of interest in the example model described in section 2.3, are the concentration of food particles and the total number and biomass of consumer individuals in the structured population. In the routine `DefineOutput()`, the first output variable `output[0]` is set equal to the current food density, given by the value of `env[1]`, the second environmental variable, in the line:

```
output[0] = env[1];
```

(Remember that the first environmental variable, `env[0]`, is always the time).

The total number of individuals in the consumer population and their biomass are subsequently determined in the loop structure that follows:

```
for(i=0; i<cohort_no[0]; i++)
  {
    output[1] += pop[0][i][number];
    output[2] +=
        pow(pop[0][i][length], 3.0)*pop[0][i][number];
  }
```

The total population size is calculated by summing the number of individuals in all the cohorts (with indices ranging from 0 to `cohort_no[0]-1`), presently constituting the population. This sum is accumulated in the second output quantity `output[1]`. The total biomass of the population is calculated by summing the biomasses of all the cohorts, which is equal to the cube of the mean length of the individuals times the number of individuals in the cohort. This sum is accumulated in the third output quantity `output[2]`. The main EBT–program subsequently writes the current time and the values of these output quantities to the output file (see section 5.3.2).

Chapter 4

Program building

After programming the model dimension and definition file an executable program has to be generated from the library modules that are supplied with the EBT–program package and the two model–dependent files. When working within a UNIX–environment this can most easily be carried out, using the **ebt**–script that is described in section 6.1 on page 47. The description below is therefore primarily aimed at users that do not employ that script.

To build a program the following library files have to be compiled into object files, using an appropriate C–compiler (see also section 1.4.1):

ebtmain.c:	The main program file with primary routines.
ebtinit.c:	Routines carrying out the initialization.
ebtcohrt.c:	Routines for organizing the cohort data structures.
ebttint.c::	Routine for the the time integration.
ebtrk2.c:	Integration method routines.
ebtrk4.c:	Integration method routines.
ebtrkf45.c:	Integration method routines.
ebtrkck.c:	Integration method routines.
ebtdopri.c:	Integration method routines.
ebtutils.c:	Low level service routines.
ebtstop.c:	Routines used during shut–down of the program.

In addition to these library files, the model definition file has to be compiled as well. The final step to obtain an executable program involves linking the compiled library modules and model definition file together.

However, for a successful compilation of these files the command–line, with which the C–compiler is invoked, should contain some specific switches. These requirements concerning the compile command–line result from the following properties of the EBT–package:

- The constants defined in the model dimension file determine the sizes of the data structures in the EBT–program (see section 3.3), on which all library files depend. As a first step, the model dimension file is always included into the the general header file **escbox.h** in which these data structures are defined. The general header file is subsequently included into every file during compilation, including the model definition file. Since the library files are designed to be model–independent the name of the model dimension file is left undefined within the file **escbox.h** and substituted by a macro

`PROBLEMFILE`. This macro `PROBLEMFILE` functions as a placeholder, which has to be specified appropriately at the time of compilation to point the compiler at the dimension file that belongs to the model for which the library files are presently compiled. Defining this macro `PROBLEMFILE` can be carried out by using the `-D` switch as an additional command–line option for your C–compiler. For example, if the model dimension file is called **example.h**, the macro `PROBLEMFILE` is correctly defined by including the following switch into the C–compiler command–line:

```
-DPROBLEMFILE="<example.h>"
```

- Since both the model dimension file and the general header file **escbox.h** have to be included into the source code files during compilation, the C–compiler has to be aware of the directories in which these files are located. These locations can be specified, using the `-I` switch as an additional command–line option. For example, if the model dimension file is located in the current directory and the general header file is to be found in a directory **/usr/local/escbox**, the compiler can be notified of these locations by including the following switches into the C–compiler command–line:

```
-I. -I/usr/local/escbox
```

- The current EBT–package can be used for the $2nd$–order version of the *Escalator Boxcar Train* only, i.e. not for the the $3rd$–order version.

Given that the command–line, with which the C–compiler is invoked, conforms to the three requirements discussed above, the building and linking of the various object modules into an executable program is straightforward. It is possible to compile and link all the library modules and the model definition file with one single call to the C–compiler. In this case all the source code files (those with a **.c** extension) should be specified on the command–line at the same time. The method I prefer involves generating a separate object module for every source file and subsequently linking them all together. The reason is that the model definition file will be changed much more often than the model dimension file. Therefore, a new compilation of all the library files, which only depend on the model dimension file, is required much less frequently than the compilation of the model definition file. Generating separate object modules for every source file makes it possible to just compile the model definition file anew and subsequently link it with the still valid object files of the library modules.

As an example, I will show the necessary commands to generate an executable program from the **ebttmpl.c** definition file, whose implementation was discussed in the previous chapter. The statements that I show below to illustrate the various steps were actually carried out on a SUN–IPX workstation using SUN–OS version 4.1.2 and its accompanying *cc* compiler. In this example it will be assumed that all the library files that make up the EBT–package are located in the directory **/usr/local/escbox** and that the model files to be compiled (in this case the **ebttmpl.c** and **ebttmpl.h** files) are in the current directory.

The following 7 statements call the C–compiler to generate the object modules from the 6 library files (**ebtmain.c**, **ebtinit.c**, **ebtcohrt.c**, **ebtrkf45.c**, **ebtutils.c**, and **ebtstop.c**) and the model definition file (**ebttmpl.c**). These statements only show the minimally required compiler options at the command–line, obviously the user can add his/her own preferred ones:

```
> cc -I. -I/usr/local/escbox -DPROBLEMFILE="<ebttmpl.h>" \
    -o ebtmain.o -c /usr/local/escbox/ebtmain.c
> cc -I. -I/usr/local/escbox -DPROBLEMFILE="<ebttmpl.h>" \
    -o ebtinit.o -c /usr/local/escbox/ebtinit.c
> cc -I. -I/usr/local/escbox -DPROBLEMFILE="<ebttmpl.h>" \
    -o ebtcohrt.o -c /usr/local/escbox/ebtcohrt.c
> cc -I. -I/usr/local/escbox -DPROBLEMFILE="<ebttmpl.h>" \
    -o ebtrkf45.o -c /usr/local/escbox/ebtrkf45.c
> cc -I. -I/usr/local/escbox -DPROBLEMFILE="<ebttmpl.h>" \
    -o ebtutils.o -c /usr/local/escbox/ebtutils.c
> cc -I. -I/usr/local/escbox -DPROBLEMFILE="<ebttmpl.h>" \
    -o ebtstop.o -c /usr/local/escbox/ebtstop.c
> cc -I. -I/usr/local/escbox -DPROBLEMFILE="<ebttmpl.h>" \
    -o ebttmpl.o -c ebttmpl.c
```

After these 7 compile statements, the current directory will contain the generated object modules, which are linked together into an executable program with the following command:

```
> cc -o ebttmpl ebtmain.o ebtinit.o ebtcohrt.o ebtrkf45.o \
   ebtutils.o ebtstop.o ebttmpl.o -lm
```

(The `-lm` switch is incorporated into this link statement to search also the mathematical library to resolve the calls to some of the routines used in the EBT–package). After carrying out this link step, the current directory contains an executable program, called **ebttmpl**, that is to say if no errors were encountered in the programmed source files. How to use this executable program for the time integration of the dynamics exhibited by the implemented structured population model is the topic of the next chapter.

Chapter 5

Program execution

The previous chapter described how an executable program could be created from the library files in the EBT–package and the model dimension and definition file, implemented by the user. This chapter will describe how the executable program has to be started (see section 5.1) and discuss the various in– and output files of the program. Apart from command–line arguments that the user might want to use in the routine `UserInit()` (see section 3.5.4) to define specific variables in the model definition file, in– and output of the program only takes place by means of files. More specifically, the program needs to read 2 input files, which are described in section 5.2.1 and 5.2.2 below, and generates up to 5 output files that are discussed in section 5.3.1–5.3.5.

5.1 Starting an integration

Starting the integration program can be done by simply typing the name of the executable file on the command–line of the operating system. One command–line argument is mandatory, indicating the name of the integration run that has to be carried out. Therefore, an integration run can be started with a statement of the form:

<program> <run>

For example, if the generated executable file is called **ebttmpl**, as was the case in the example discussed in the previous chapter, and the integration run to be carried out is called **tmpl**, the command to start the integration run is:

```
ebttmpl tmpl
```

The importance of the name of the run is that it equals the root of the name of all the in– and output files. All the files read or written by the program have this same basename and only differ from each other by a unique extension. As input the integration program reads a *control variable file* (discussed in section 5.2.1) and an *initial state file* (discussed in section 5.2.2). The program assumes that these two files bear names that are constructed by adding a **.cvf** and **.isf** extension, respectively, to the common root of the filename, *i.e.* the name of the run. Similarly, the program constructs the names of the output files by adding a unique extension to the name of the run (see the description of the output files in the sections 5.3.1–5.3.5 below).

As part of the EBT–package (see the contents description in section 1.4.2 on page 6) two files are supplied, called **tmpl.cvf** and **tmpl.isf**, that serve as templates for the *control variable* *file* and the *initial state file*, respectively. Therefore, the example of the command–line given above starts the program **ebttmpl** from the previous chapter to carry out the run **tmpl**, the characteristics of which are determined by these two template files.

Obviously, to start the integration program successfully the executable file has to be found by the operating system. It should therefore be in the current directory or in a directory that is part of the path that is searched by the operating system for executable files. Alternatively the full name of the file, including the directory in which it is located, can be used to start the program. In the same way the name of the run should give sufficient information on the location of the input files. Since the names of the input files are constructed by only adding an extension to the name of the run, the latter can contain a directory specification as well. If the program does not succeed in localizing one of the input files, it exits with an appropriate error message.

5.2 Input files

As mentioned, the program reads a *control variable file* and an *initial state file* as input. The latter file has to contain all the information on the initial state of the environment and the statistics (number of individuals, mean *i–state* variables, plus the value of the *i–state* constants) for all the cohorts in the various structured populations in the model. The *control variable file* has to contain all the necessary variables that control the operation of the program. Examples are the maximum time until which the integration has to be continued, various tolerance levels determining the required accuracy of the integration, but also the values of the parameters for the current integration run. Both these files, which are plain text files containing only characters, digits, tabs, and white spaces, should have a specific format, otherwise the integration program cannot successfully read them. The necessary layout of both files is discussed in the next two sections.

5.2.1 The control variable file (CVF–file)

The EBT–package contains templates of the *control variable file* (CVF–file) for the $2nd$–order version of the *Escalator Boxcar Train*. This files is called **tmpl.cvf**. Here I will discuss the layout of this file, which is the appropriate CVF–file for the example model **ebttmpl** that has been used as an illustration throughout all previous chapters.

The file **tmpl.cvf** contains the following lines of plain text:

```
"Integration accuracy"                                    1.0E-8
"Cohort cycle time interval"                              0.5
"Tolerance value, determining identity with zero"         1.0E-5

"Maximum integration time"                              100.0
"Output time interval"                                    1.0
"Complete state output interval, 0 for none"             50.0

"Minimum allowable number of individuals in cohort"       0.0

"Relative tolerances for i-state variable #0"             0.0
"                         i-state variable #1"            0.0
"Absolute tolerances for i-state variable #0"             0.0
"                         i-state variable #1"            0.0

"Birth rate coefficient Alpha"                            0.75
"Individual death rate Delta"                             0.1
"Logistic growth rate R"                                  3.0
"Carrying capacity K"                                     8.3
```

The lines or records in this file all have the same structure: every record consists of a string that gives a verbal description of the quantity that is specified on the current line, followed by one or more real–valued numbers, specifying the value(s) to be used in the current integration run for the particular control quantity. If the particular control quantity is specific for one structured population in the model only, there have to be exactly `POPULATION_NR` real–valued numbers specified after the description string, one pertaining to each population. This holds for the minimum cohort size and the relative and absolute tolerances for the *i–state* variables. The description string is contained within double quotes to indicate its start and end, although this is not absolutely necessary. If a record does not start with a double quote the program assumes that all the characters up to the very first digit encountered make up the description string (Note that this would cause problems if the description string contains a digit as a character, as is the case in the **tmpl.cvf** file).

The empty lines that occur in the file are irrelevant. They are inserted for better readability and the program skips these empty lines while processing the file. The program reads and stores the description strings simply to repeat them afterwards in the *report file* (see section 5.3.1), it does *not* interpret them at all. The entire set of control variables is read in a predetermined way. Therefore, the order of the different records (lines) in the CVF–file, not its absolute position, is of utmost importance and should be maintained meticulously. In the next sections the different records in the file will be explained in more detail.

Integration accuracy

This first record in the CVF–file specifies the required accuracy with which individual integration steps have to be carried out. The EBT–package employs the $4th/5th$–order Runge–Kutta–Fehlberg method with adaptive step size to carry out the time integration of all ODEs (see (9)). This method performs every integration step twice but uses both times a different method: the first time a $4th$–order Runge–Kutta method and subsequently a $5th$–order Runge–Kutta method. The difference between the results obtained with the two methods is measured relative to the current value of the variable that is integrated, yielding an estimate of the relative error made during the integration of the variable. Subsequently, the maximum relative error is determined over the set of all variables of which the dynamics are described by ODEs. If the maximum relative error thusly obtained is less than the integration accuracy specified in the first record in the CVF–file, the integration step is accepted as successful, otherwise it is rejected and the step size is decreased. Appropriate values for this integration accuracy are of the order 10-*7*–10-*9*, smaller values imply a more accurate integration using smaller steps and taking longer to complete the run.

Cohort cycle time interval

The second record in the CVF–file specifies the length of the cohort interval, *i.e.*, the interval between the equidistant time values at which the boundary cohorts are closed off, transformed into internal cohorts and replaced by a new set of boundary cohorts (See section 3.5.8 for a complete discussion of the times at which the boundary cohorts are transformed and replaced). The length of this cohort interval is entirely dependent on the structured population model itself. However, smaller values lead to a better approximation of the analytical model dynamics, but imply that more often, and hence more, new internal cohorts are created. As a result the number of ODEs to be integrated is larger and the program will run slower. For the example model **ebttmpl** a value of 0.5 turns out to be appropriate. It is advisable when studying the dynamics of a structured population model to study the influence of the cohort interval length. It should have such a value that halving or doubling its value does not influence the resulting dynamics significantly.

Tolerance for identity to zero

The third record in the CVF–file determines the tolerance level when quantities in the program are compared to 0. This control variable is especially important in switching problems, that is when the dynamic cohort closure option is used. As described in section 3.5.8 on page 30, in switching problems the program attempts to locate the exact moment at which the returned value of the routine `Eventlocation()` equals 0. In reality the program assumes it has obtained this exact moment when the returned result is smaller in absolute value than the tolerance specified in this third record in the CVF–file. Appropriate values are in the order of 10-*3*–10-*5*. If smaller values are taken the program will try to determine the moment at which the returned value of the routine `Eventlocation()` equals 0, more accurately. This is, however, very time consuming and should be used with great care.

In non–switching problems the tolerance level in this record in the CVF–file is only used to determine whether it is time to write the output variables or the entire state of the model to the corresponding file. Here the value of the tolerance level is less critical and can easily be taken equal to 10-*5*, as is done in the **tmpl.cvf** file for the **ebttmpl** model.

Maximum integration time

The fourth record in the CVF–file is almost self–explanatory, as it specifies the maximum time value until which the integration has to be continued. On reaching this maximum integration time, the program will write the output variables to the output file, save the final state of all the variables in the model, write the report file, and exit.

Output time interval

The fifth record in the CVF–file specifies the time interval between successive output moments. The output variables defined in the routine `DefineOutput()` (see section 3.5.11) are written to the *output file* at the start of the integration run and at equidistant times afterwards, the length of the interval in between being specified in the current record of the CVF–file. The *output file* is more fully described in section 5.3.2. It is *not* possible to specify a zero value for this time interval.

Complete state output interval

Apart from writing the values of the output variables, defined by the user, to the *output file*, the program incorporates the option to write the entire state of the environment and all the structured populations to a separate file, called the *complete state output file* (CSO–file; see section 5.3.4 on page 45 for a description of the form in which the entire state is saved in this file). The sixth non–empty record in the CVF–file specifies the time interval between the moments at which the entire state is written to this CSO–file. It should be noted that the entire state of the system constitutes a considerable amount of data and specifying too small a value at the current record will lead to a very big CSO–file. In contrast to the regular output time interval, it is possible to specify a zero value for this time interval, which has the meaning that the entire state of the model is never saved in the CSO–file. Consequently, if the value specified in this sixth record equals 0, the CSO–file is not created at all.

Minimum cohort size

As described in section 2.2 on page 8, the integration program screens all cohorts in all structured populations at the end of a cohort cycle interval. Cohorts that have become too small are subsequently removed from the population. The quantity in the seventh record of the CVF-file determines what exactly is meant by "too small". If the number of individuals in a cohort is less than the minimum cohort size specified in the current record, it is considered negligible and removed from the population. If this quantity is set to 0, cohorts are never considered to be too small and are hence never removed in this way from the structured population. Obviously, the minimum cohort size can be used to limit the number of cohorts in a structured population by removing cohorts of which the influence is negligible. If no other mechanism keeps the total number of cohorts within bounds, this minimum cohort size should absolutely be set to a non–zero value. Otherwise the total number of cohorts will simply grow unlimitedly and the integration program will slowly grind to a halt.

Smaller values for the minimum cohort size would postpone deleting the cohorts from a population and hence will lead to a better approximation of the analytical model dynamics. This better approximation is obtained at the expense of a larger system of ODEs that has to be integrated over time and hence an integration program that is considerably slowed down. The appropriate value of the minimum cohort size is strongly model–dependent. For the example model **ebttmpl** it is set equal to 0 in the **tmpl.cvf** file. The example model implements a maximum age after which all the individuals die instantaneously (see section 3.5.10 on page 32). In the cohorts that reach this maximum age the number of individuals is set to 0 and hence they are removed irrespective of the minimum cohort size. The presence of this mechanism to limit the number of cohorts in the structured population makes it unnecessary to specify a non–zero value for the minimum cohort size.

The minimum cohort size can be different for the different structured populations in the model. Therefore, if more than one structured population is present in the model, the minimum cohort size has to be given for them separately. This seventh non–empty record in the CVF–file should always specify as many values after the description string as there are structured populations present in the model, meaning that the number of values given for the minimum cohort size should equal the value of the constant `POPULATION_NR` (see section 3.3.1 on page 15). If not enough values are given, the program will exit with an appropriate error message. Note, however, that the description and all the values should still be given on the same line, the subsequent values separated from each other by white spaces.

Relative and absolute tolerances for i–state *variables*

Simultaneously with screening all the structured populations at the end of a cohort cycle for cohorts that have become too small, the populations are also screened to identify cohorts that have become identical (see section 2.2 on page 8). If cohorts are found to be identical, they are joined together. The resulting new cohort has a total number of individuals that equals the sum of the number of individuals in the two original cohorts. The values of the mean *i–state* variables and *i–state* constants in the new cohort is set to the weighted mean of these statistics in the original cohorts.

Joining identical cohorts is another way to keep the number of cohorts in the structured populations within bounds, just like the removal of cohorts that have become too small. The relative and absolute tolerances that have to be specified in the CVF–file determine whether two cohorts have become identical. Two cohorts are considered *identical* with respect to a specific *i–state* variable if *either* the difference between the values of this *i–state* variable in the two original cohorts is less than the *absolute* tolerance specified in the CVF–file, *or* the quotient of this difference and the mean of the *i–state* variables in the two cohorts is less than the *relative* tolerance (Note that only one of the relative and absolute tolerance has to be met to consider two cohorts identical with respect to the specific *i–state* variable). Two cohorts are now considered completely identical, if they are identical in all their *i–state* variables. Only then the cohorts are joined.

The number of records that specify these tolerances in the CVF–file varies between models, as it always has to equal the number of *i–state* variables (Remember that the number of *i–state* variables was necessarily equal in all structured populations in a model). The tolerance values for all the *i–state* variables are specified on separate lines in the CVF-file, first all relative, subsequently all absolute tolerances. Therefore, after the seventh record specifying the minimum cohort sizes, first `I_STATE_DIM` records should be given that specify the relative tolerance for each of the *i–state* variables, numbered $0 \ldots \texttt{I\_STATE\_DIM} - \texttt{1}$. Following these relative tolerances, the absolute tolerances should be specified for each of these `I_STATE_DIM` *i–state* variables. Self–evidently, the order in which the tolerances are specified should match the order of the *i–state* variables, as used in the model definition file.

As was the case for the minimum cohort size, also the tolerance values can differ for different structured populations. If more than one structured population is present in the model, every single tolerance value has to be given for them separately. Therefore, each tolerance record in the CVF–file should always specify as many values after the description string as there are structured populations present in the model (equal to `POPULATION_NR`, see section 3.3.1 on page 15). If not enough values are given, the program will exit with an appropriate error message. Note, however, that the description and all the values should still be given on the same line, the successive values separated from each other by white spaces.

In the template file **tmpl.cvf** all relative and absolute tolerances are set to 0. This implies that no screening of the cohorts will take place at all, which will speed up the integration program considerably. Lumping identical cohorts in the example model **ebttmpl** is unnecessary since the number of cohorts is limited anyway, because of the maximum age that individuals can attain. It should be noted that setting both the relative *and* the absolute tolerance to 0 for only one *i–state* variable, is sufficient to inhibit lumping of cohorts in all cases. Identity of cohorts is then impossible.

Parameter values

The last records in the CVF–file have to specify the values of the free parameters in the model (see the sections 3.2, 3.3.6 and 3.5.3). Every free parameter is specified on a separate line, hence there should be a total of `PARAMETER_NR` records altogether (If `PARAMETER_NR` equals 0, no records are required). Self–evidently, every record should contain only a description string plus one single real–valued number and the order of the records should match the order that is used in the model definition file (see section 3.5.3 on page 25).

In the example model that is implemented in the file **ebttmpl.c** the quantities α, δ, r_m and K (see the model description in section 2.3) are taken as free parameters and are assumed to be the first, second, third and fourth element of the array `parameter[]`, respectively. Therefore, there appear 4 records in the CVF–file **tmpl.cvf** that successively specify the values of these parameters.

5.2.2 The initial state file (ISF–file)

The *initial state file* (ISF–file) has to define the initial state of the environment and all the structured populations in the model. The program reads this file after reading the CVF–file, creates the appropriate number of internal cohorts for every population and stores the values that it reads from this file. As the integration program reads the file in a specific order, the layout of the ISF–file is again of overriding importance. If this layout is not meticulously adopted, the initial state will not be read correctly. It is possible that an incorrectly specified initial state will generate no error message and the program will nonetheless start the integration run.

In contrast to the CVF–file, empty lines in the ISF–file are very important. The presence of an empty line is interpreted by the integration program as the end of the current section in the ISF–file. Every such a section contains the initial state of either the environmental data or of one of the structured populations in the model. The use of any type of comment in the ISF–file is also discouraged, it will simply be skipped by the program and can cause problems while reading the file (Only the presence of real–valued numbers counts, which means that a line is considered empty even if it is full of non–digit characters).

The first line in the ISF–file should contain the initial values for the environment variables in the model. The first value should always equal the initial time value, as explained in section 3.3.4. The following values should be the initial data for the other environmental statistics. In total, the number of values specified in this section should equal the value of the constant `ENVIRON_DIM`. (These values can best be given on one single line, although the program is flexible at this point: the integration program simply reads the first `ENVIRON_DIM` values in the ISF–file as initial values for the environment variables.)

After the initial state of the environment one or more empty lines should be inserted in the ISF–file to identify the start of the next section. The second section in the ISF–file contains the initial data for the structured population with index 0 in the current model. Every line in this second section has to specify the initial values for exactly one single cohort. The program is very strict about this: if there are not enough data on one line to define a cohort completely, it will discard the entire data line, ushering a warning about an incomplete cohort specification. Similarly, if there are more numbers present on a single line than is necessary to define a single cohort, the superfluous numbers are discarded. The program will not warn about the latter situation.

The order of the numbers on every single line is also significant. The first number read on a line is assumed to be the total number of individuals in the cohort. The following quantities are taken as the initial values for the mean *i–state* variables in the order that is adopted by the user throughout the model definition file (see section 3.5). Whatever the structured population model, every line has to contain at least this total number of individuals and a total number of `I_STATE_DIM` values, representing the means of the *i–state* variables. The program then continues with reading the initial values of the *i–state* constants. These should be given as the last set of numbers on the line, their number should obviously equal the value of the `I_CONST_DIM` constant (see section 3.3.3) and the order in which they are specified should match the order that is adopted by the user throughout the model definition file. If the constant `I_CONST_DIM` is set to 0 in the model dimension file, the program naturally skips this part of the initialization of a cohort.

After the successful processing of a single line and initialization of the corresponding cohort, the program starts reading the subsequent line. If this line also contains a set of numbers, it is assumed to define the initial state of another cohort belonging to the same structured population as the previous one. The process is repeated until the program encounters one or more empty lines, which are interpreted as the end of the initial state of the structured population that is currently read. If the thusly completed structured population wasn't the last one in the model, the program continues to search for the initial state of the next structured population. This cycle is repeated until a valid initial state for all the structured populations in the model is read from the ISF–file.

Summarizing, it is important to remember that sections in the ISF–file are separated from each other by empty lines, containing no numbers at all. The sections specify the initial state of the environment and the successive structured populations, while the initial state of a single cohort should always occupy exactly one line of data. Failure to adhere to the described layout can cause warning messages about insufficient cohort specifications or structured populations that do not contain any cohort at all. In case of the incomplete cohort warning the program will continue the integration if possible, in case of a population without any cohort the program will exit with an error message.

The EBT–package contains a file **tmpl.isf** that is an appropriate *initial state file* for the example model **ebttmpl**. This file, which can serve as a template for any structured population model implemented using the $2nd$–order version of the *Escalator Boxcar Train*, contains the following lines of data:

```
0   2.5

10.0   20.0   0.5
```

The data define the initial time at the start of the integration run (`env[0]`) to equal 0 and the current food density, which was the second environmental variable in the model (`env[1]`) to equal 2.5. At the start of the integration, the only structured population in the model is made up of one single cohort containing 10 individual consumers with an age of 20 time units and a length equal to 0.5 (Remember that these quantities were scaled in the example model and hence dimensionless, see section 2.3).

5.3 Output files

After the extensive discussion of the input files, the output files will require considerably less explanation. Up to 5 output files will be created by the integration program, the exact number depends on whether complete state output is required (see section 5.2.1) and whether the implemented model uses the dynamic cohort closure option (see section 3.5.8 on page 30). The 3 files that will always be generated are the *report* (see section 5.3.1), the *output* (see section 5.3.2) and the *end state file* (see section 5.3.3). If the time interval for complete state output is set to a non–zero value (see section 5.2.1), the program will also regularly write the entire state of the model to the *complete state output file* (see section 5.3.4). All generated output files are plain text files, containing simple characters, digits, tabs and white spaces.

5.3.1 The report file (REP–file)

The *report file* (REP–file) is one of the 3 files that is always generated by the integration program. It reports the name of the executable file that generated the report, the name of the run that was carried out and the values of the control variables used during the integration. In addition to the name of the program and the run it hence simply contains a copy of the contents of the CVF–file (see section 5.2.1). Both the description strings and the specified values from the CVF–file are written to the report file. Its contents are therefore self–explanatory. The name of the report file is constructed by adding the extension **.rep** to the common root of the filename, *i.e.*, to the name of the run.

5.3.2 The output variable file (OUT–file)

The *output file* contains the values of the output variables, as defined by the user in the routine `DefineOutput()` (see section 3.5.11 on page 33). Its name is constructed by adding the extension **.out** to the common root of the filename (name of the run). The values of the output variables are saved for the first time at the start of an integration run. Subsequently, their values are appended to the file at equidistant moments in time, the interval in between two output events being equal to the value of the "output time interval", as described in section 5.2.1.

The output file contains lines of data, in which the individual numbers are arranged in columns, separated from each other by tabs and white spaces. The first number in every data line always gives the time at which the variables were determined, the following numbers are the actual values of the output variables at that particular time. The order of these output variables is determined by the user in the routine `DefineOutput()`. The total number of values on a data line hence is one more (the time value!) than the value of the `OUTPUT_VAR_NR` constant (see section 3.3.5 on page 17).

5.3.3 The end state file (ESF–file)

At the end of an integration run, whether the maximum integration time has been reached or the integration is stopped for another reason, the program tries to save the entire state of the model in a file, the *end state file* (ESF–file). The name of the ESF–file is constructed by adding the extension **.esf** to the common root of the filename (name of the run). Saving the final state of the model might not always be successful or possible, for instance, when there is no more room on the file system to store new information. On the other hand, the program might come to a halt because one of the populations has gone extinct, while all cohorts died out. In the latter case the ESF–file does not contain very much valuable information. However, when the program stops because the maximum integration time has been reached, the ESF–file usually does contain the entire final state of the model. The file then has exactly the same layout as the ISF–file. It can therefore be copied to a new file with a **.isf** extension and used as the initial state file for a next run. In this way it is possible to continue an integration run at the point where it stopped.

5.3.4 The complete state output file (CSO–file)

The *complete state output file* (CSO–file) is only created by the program if the value of the "time interval between complete state output" is set to a non–zero value in the CVF–file (see section 5.2.1). It is again a plain text file, the name of which is generated by adding the extension **.cso** to the common root of the filename (name of the run). If the time interval between complete state output is non–zero, the specified value equals the interval between times that the entire state of the model is written to the CSO–file, where the start of the integration run is the first time the entire state of the model is saved. The format that is used to save the entire state of the model is identical to the format that is described for the ISF–file, and is also used in the ESF–file. Hence, the complete state output starts with a line containing the values of all the environmental data and continues with writing all the individual cohort statistics for each structured population in the model. Each cohort takes up one line of data and an empty line marks the end of the current section, being the environment or a structured population. Every line contains the total number of individuals in a cohort, their mean *i–state* values and the values of the *i–state* constants (see section 5.2.2 on page 43). At subsequent times of complete state output the entire state of the model is simply appended to the CSO–file.

5.3.5 The complete state binary output file (CSB–file)

This is a binary version of the .CSO file that can be read by the Ebttool.

Chapter 6

Bourne–shell scripts

In addition to the library and template files that constitute the core of the EBT–package, it also contains a number of Bourne–shell scripts to be used within a UNIX–environment only. These scripts are not essential but facilitate certain tasks that frequently recur while using the EBT–package. They have been created and tested on a SUN–IPX workstation running SUN–OS version 4.1.2. For a proper functioning of these shell scripts a number of environment variables should be defined (refer to the descriptions below) and the `PATH` variable, which specifies the directories that the operating system searches for executable files, might have to be changed. Otherwise, the scripts should not require any adaptation for the use on different systems.

The **ebt**–script, discussed in section 6.1, is a helpful tool that automates the entire process of building and maintaining the executable file from the library files and the model dimension and definition file. The **ebtclean**–script (section 6.2) removes on request either the compiled object and executable files of a specified model or the output files of a specific run. In the following sections the use of these shell scripts will be explained one by one.

6.1 Compilation using the EBT–script

Syntax

`ebt` [switches] <*model*>

Description

This script helps in building and maintaining the object and executable files of a structured population model that has been implemented using the EBT–package. The argument <*model*> should be the name of a structured population model and hence a model dimension and definition file have to be present under that name with a **.h** and a **.c** extension added to it, respectively. For example, to use the script for building the **ebttmpl** model discussed in previous chapters both the **ebttmpl.h** and **ebttmpl.c** file should be present. If these files are not found, the script asks the user to confirm the creation of the files from the templates in the EBT–package.

The script will build all object files and the executable file if they are not present yet. The compiled object files that are generated from the EBT library files are linked together into one library module, the name of which is constructed by adding a **.lib.o** extension to the name of the model itself (*cf.* **ebttmpl.lib.o**). This library module contains all model–independent routines in compiled form and only depends on the model dimensions. The model definition file is compiled and stored under a name that equals the name of the model with a **.o** extension added to it (*cf.* **ebttmpl.o**). The last step in the building process is linking together the compiled library module (name of the model plus **.lib.o** extension) and the compiled model definition file (name of the model plus **.o** extension) into an executable file that is saved under the basename of the model (*cf.* **ebttmpl**). When invoked again the script checks all dependencies and will rebuild the object modules and the executable file, as far as necessary. If the model dimension file has changed, both the compiled library module and the compiled model definition file have to be created anew. If the model definition file is changed, only this file is compiled anew and linked together with the still valid compiled library module.

The **ebt**–script implements the appropriate commands to compile and link all the files, including the command–line arguments that are specifically required for the compilation of the files in the EBT–package (discussed in chapter 4 on page 34). Hence, it defines the `PROBLEMFILE` macro in accordance with the name of the model that is specified as the argument to the script. In addition, the script can also be used for running a line checking program, such as "`lint`", on the model definition and/or the library files to check the consistency of the C–coding in these files. These operations are controlled by command–line switches as well (see below).

The operations of the **ebt**–script are controlled by a number of environment variables that determine the name and desired command–line switches of the C–compiler used for the compilation, the name and desired command–line switches of the line checking program, and the location of all EBT–files. These environment variables are described below and should be set appropriately for the computer system on which the script is used.

Switches

-l Run the line checking program (*cf.* `lint`) on all source files, including the model definition file and all EBT library files.

-t Run the line checking program (*cf.* `lint`) on the model definition file only, skip the EBT library files.

-h Display a help message, listing all the switches and the required syntax to invoke this script.

Environment variables

EBTPATH

This environment variable is read by the shell script to determine the directory in which the EBT–package is installed. The script carries out a restricted test whether all library and template files are really present in the specified directory. If not, it exits with an appropriate error message.
Default value: **/usr/local/escbox**.

CC
: This environment variable should contain the command with which to invoke the C–compiler to be used for the compilation of the source files. The command should either have a full pathname preceding it, or the directory in which the compiler resides should be part of the `PATH` variable.
Default value: `cc`.

CCFLAGS
: This environment variable should contain the command–line switches with which to invoke the C–compiler. Added to these switches are the switches that are necessary for the successful compilation of the EBT files (see chapter 4).
Default value: `-O`.

LINT
: This environment variable should contain the command with which to invoke the line checking program to be used. The command should either have a full pathname preceding it, or the directory in which the program resides should be part of the `PATH` variable.
Default value: `lint`.

LINTFLAGS
: This environment variable should contain the command–line switches with which to invoke the line checking program.
Default value: `-u -v`.

LDFLAGS
: This environment variable should contain the command–line switches for linking the compiled model definition file and the compiled library files into an executable program. Mainly used to specify which libraries have to be searched for unresolved function calls.
Default value: `-lm`.

6.2 Deleting files using the EBTCLEAN–script

Syntax

`ebtclean` [switches] <*name*>

Description

The **ebtclean**–script deletes on request files that are generated either by the **ebt**–script during compilation and program building or by an integration program itself while carrying out a specific run. The only argument <*name*> should hence be a valid name of a model or an integration run. The functioning of the script is very simple: it checks for the presence of the following files:

- an executable program, called <*name*>,
- a compiled object file, called <*name*>**.o**,

- a compiled library file, called <*name*>**.lib.o**,
- a *complete state output file*, called <*name*>**.cso**,
- a *complete state binary output file*, called <*name*>**.csb**.
- an *end state file*, called <*name*>**.esf**,
- an *output file*, called <*name*>**.out**, and
- a *report file*, called <*name*>**.rep**.

The full name of every one of these files that is indeed encountered by the script is passed on to the standard UNIX delete command (`rm`) together with the command–line switches that were specified when the script was invoked. The mentioned files are subsequently removed, either silently or after confirmation by the user.

Switches

-f	Standard command–line switch of the UNIX delete command (`rm`). Forces files to be removed without verification and without reporting error messages.
-i	Standard command–line switch of the UNIX delete command (`rm`). Asks the user to verify the deletion of each file.

6.3 References

[1] A. M. de Roos. Numerical methods for structured population models: The Escalator Boxcar Train. *Num. Meth. Part. Diff. Eqs.*, 4:173–195, 1988.

[2] A.M. de Roos. A gentle introduction to physiologically structured population models. Pp. 119-204 *in:* S. Tuljapurkar and H. Caswell, eds., *Structured-population models in marine, terrestrial, and freshwater systems*. Chapman and Hall. New York, 1997.

[3] J. A. J. Metz, A. M. de Roos, and F. van den Bosch. Population models incorporating physiological structure: A quick survey of the basic concepts and an application to size–structured population dynamics in waterfleas. In B. Ebenman and L. Persson, editors, *Size–structured populations*, pages 106–126. Springer–Verlag, Heidelberg, 1988.

[4] J. A. J. Metz and O. Diekmann. *The dynamics of physiologically structured populations*, volume 68 of *Lect. Notes in Biomath.* Springer–Verlag, Heidelberg, 1986.

[5] A. M. de Roos, O. Diekmann, and J. A. J. Metz. Studying the dynamics of structured population models: A versatile technique and its application to Daphnia. *Am. Nat.*, 139(1):123–147, 1992.

[6] A. M. de Roos, O. Diekmann, and J. A. J. Metz. The Escalator Boxcar Train: Basic theory and an application to Daphnia population dynamics. Technical Report AM–R8814, CWI, Amsterdam, The Netherlands, 1988.

[7] P. J. Plauger and J. Brodie. *Standard C: Programmer's quick reference guide.* Microsoft Press, Redmond, Washington, USA, 1989.

[8] A. M. de Roos, J. A. J. Metz, E. Evers, and A. Leipoldt. A size dependent predator-prey interaction: Who pursues whom? *J. Math. Biol.*, 28:609–643, 1990.

[9] J. Stoer and R. Bulirsch. *Introduction to numerical analysis.* Springer–Verlag, New York, USA, 1980.