**This practical (TD)**

The goals of this practical are to learn:

- how to use the ebttool
- how the EBT program Works
- how to program the EBT (including some basic C code)
- to analyse output generated by the EBT
- to translate biological problems into (EBT) models

**The EBT and its code**

The general idea of the EBT method (for solving PSPMs) has been explained in the lectures and in the course handbook. One step further is to actually implement a PSPM as a computer program and to study it numerically. To make this step easy, we use the EBT program. It is a program, written in the programming language 'C', which allows you to study PSPMs without worrying too much about the technical problems (i.e., computer and programming related problems). Instead, you can focus on the modelling aspects, which are a lot more interesting (from a biological point of view anyway).

However, in order to do so, you will have to understand some basics of the programming of the EBT method, and of programming in C. That is the first goal of this practical. The second (and ultimate) is to know how to translate a biological problem (concerning a structured population) into an EBT model from scratch.

We start with the first goal, and we use an example.

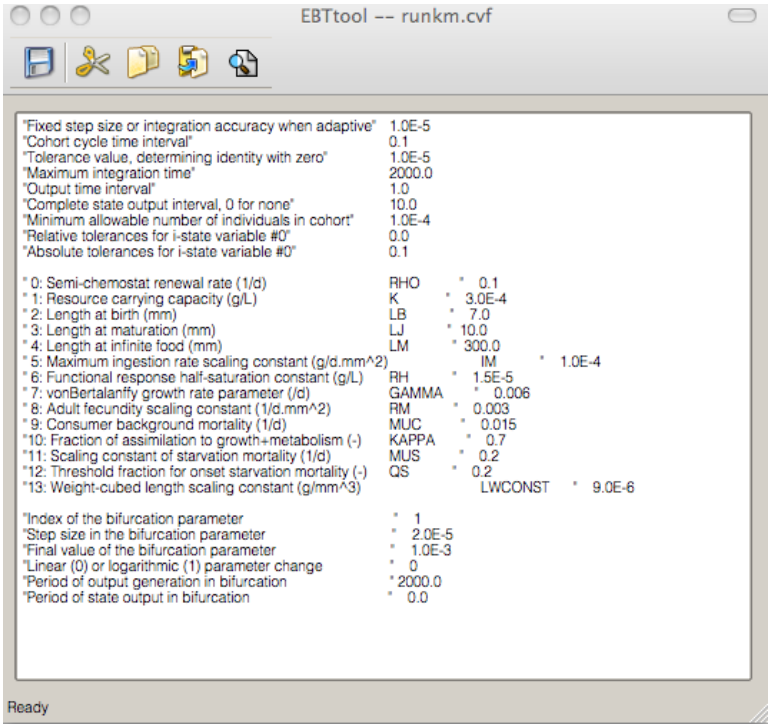
The EBT software consists of a set of routines written in C, which take care of:

- reading input necessary for a computation, usually from a file written by you
- doing all the computation of life history and population dynamics
- writing output to file.

The ebttool is graphic tool which gives you easy control over all of these aspects, and which allows you to see the output visually in the form of graphs. Most of the routines of the EBT software are written down in a collection of files hidden away from you (you can find them in places such as **/usr/local/escbox**). Together with two so-called 'problem files' (i.e., one .c file and one .h) these files can be compiled into an executable computer program. The problem .c file contains routines which specify the biological model, and the .h file (the 'header file') contains some global definitions of the model. These two files are where you implement your models; the goal of this practical. Once you have written the model, you can compile the whole program using the command 'ebt', or using the ebttool. Then in order to run the program, you need two more files. One to set the so-called 'control variables', which are general settings for the ebt program, such as the total time for your computation and the time interval between output. The other contains the initial conditions of the run.

Summarising, there are four files of interest:

- the .c file (e.g. model.c) with the model
- the .h file (e.g. model.h) with some global definitions
- the .cvf file (e.g. run.cvf) with settings and parameter values
- the .isf file (e.g. run.isf) with initial conditions




**Output files**

The EBT produces different types of output. First of all, the output as you define it yourself in the C-function DefineOutput(), specified in the .c file, can be found in the file **run.out**. The first column is the time, followed by column which contain output[0], output[1], etc. The interval between output is determined in the file run.cvf.

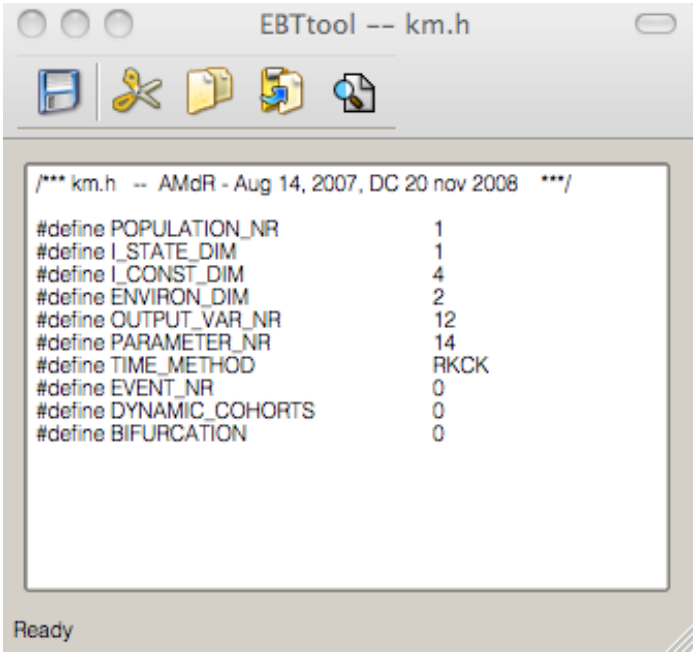
Second, the complete state of the system at the end of the run is saved in the file **run.esf**. This file contains information on all environmental variables, and all population variables, at the last moment of the run. (So you can rename this file into run2.isf, for example, to start a new run from this point). The first line lists env[0], env[1], etc. The sec- ond line is blank. The third line lists the data for the first cohort. The first column has pop[0][i][number], followed by the i-states, in turn followed by the i-consts (popIDcards). The next lines contain the data of the other cohorts. Throughout the run, at intervals specified in run.cvf, the program writes such 'complete state' output to the file **run.csb**. This file is binary but can be translated into normal text with the command csb2txt run.csb at the command line, or via the ebttool directly. If you write csb2txt run.csb > run.cso then the complete state output is saved in the file **run.cso**.

The file run.cso now contains, for each moment of output as defined in run.cvf, information on all environmental variables, and all population variables. The format is the same as for the run.esf file.

Finally, the file **run.rep** contains a report of the run which can be used to reproduce the same run at a later time, for example.

**Explanation of the EBT code**

***The header file km.h***




This file contains the settings of the various constants that are necessary to tailor the Escalator Boxcar Train program to the problem under study. The more important entries in the file are explained below.

```
#define POPULATION_NR 1
```

...defines the number of structured populations in the problem.

```
#define I_STATE_DIM 1
#define I_CONST_DIM 4
```

...defines the dimension of the i-state and the number of constant variables that characterize a cohort. These dimensions should be the same for all the populations.

```
#define ENVIRON_DIM 2
```

...defines the number of variables characterizing the environment. Here it is 2: time and the resource density.

```
#define OUTPUTVAR_NR 12
```

...defines the number of quantities that have to be written to the output file each time that output should be generated.

```
#define PARAMETER_NR 14
```

...defines the number of free parameters in the problem. These parameters can be changed between various runs without compilation of the program. Parameters that are fixed in the problem can be defined as constants in the problem-specific program file written by the user. Changing of these constants requires a new compilation of the program before use.

***The problem file model.c***

This file contains the user-defined routines of the EBT integration program. So this is where you specify your model. A number of functions are explained here:

**LABELING ENVIRONMENT AND I-STATE VARIABLES**

For convenience it is possible to label the environment and i-state variables with a more meaningful name by defining, for instance, see:

```
#define time            env[0]
#define resource        env[1]
#define length          i_state(0)
#define IDage           i_const(0)
```

...which define the variables (E-state) and (i-state). Note the zero-based array indexing in the C-language.

**DEFINING AND LABELING CONSTANTS AND PARAMETERS**

Define the constant names and values used in the user-specified routines. Most parameters in the problem can be treated as constants for this file. This seems the easiest way to specify the parameters. The parameters that are free to change will be in a vector called "parameter[]", defined elsewhere. As with the i-state variables it is possible to label these parameters with a more meaningful name, e.g.:

```
#define RHO       parameter[ 0]
#define K         parameter[ 1]
#define LB        parameter[ 2]
```

**SPECIFICATION OF THE NUMBER AND VALUES OF BOUNDARY POINTS**

SetBpointNo() and SetBpoints()

Newborn individuals enter the population at the so-called ‘‘boundary points’’.

Specify here the number of boundary cohorts that should be created at the start of the next cohort integration cycle. Fill the array "bpoint no[]" with the appropriate integer values. The length of the array is "POPULATION_NR". The state of the environment and the population can be used to adapt the number of the fixed points on the boundary to the current state.

In the function SetBpoints you specify the state of newborn individuals. In the example km.c:

```
bpoint no[0]=1;
```

...so there is only one boundary point (all individuals are born with the same state), and

```
bpoints[0][0][length]=LB;
```

...which means that all individuals are born with length LB.

**SPECIFICATION OF DERIVATIVES**

```
Gradient()
```

Here the derivatives of the i-states, E-states and cohort abundances are specified. Define the derivatives of the various environment variables, which have to be returned to the main program in the array "envgrad[]". Define also the derivatives of the various population variables, which have to be returned to the main program in the matrix of cohort variables "popgrad[][][]" for each population. Finally define the derivatives of the various offspring variables, which have to be returned to the main program in the matrix of cohort variables "ofsgrad[][][]" for each population. NB :

The integer array "cohort no[]" is globally available and denotes the number of internal cohorts present in each structured population.

The integer array "bpoint no[]" is globally available and denotes the number of boundary cohorts present in each structured population during the current cohort cycle.

The offspring number can be zero. If used in a division, check for this equality to zero!!

So, to conclude, the goal of this function is to define the derivatives of the state variables of the model. In the km.c model, these derivatives are:

```
popgrad[0][i][number] = -mort*pop[0][i][number];

envgrad[0] = 1;
envgrad[1] = RHO*(K- resource) - IM*Fr*grazing/LAKEVOLUME;
```

Note the use of the ID cards, which also referred to as i-constants. They are not state variables themselves, but they usually are functions of the state variables. They can come in handy to remember something, such as the age-specific death rate in this case.

**SPECIFICATION OF BETWEEN COHORT CYCLE DYNAMICS**

```
InstantDynamics()
```

This routine is called at the end of each cohort cycle. The routine can be used to implement any type of instantaneous dynamics, occurring between two subsequent cohort cycles. It can, for instance, be used for a pulsed, instantaneous reproduction process or to set the number of individuals in cohorts that have reached their maximum lifespan to 0.

(Note that the transformation of boundary cohorts into internal cohorts has already been performed and that the boundary cohorts are hence characterized by the number of individuals and their transformed moments. In an instantaneous reproduction process the i-state of the offspring can therefore be simply specified in terms of the mean i state!)

**SPECIFICATION OF OUTPUT VARIABLES**

```
DefineOutput()
```

Define here the values of the output variables in terms of the population and environment statistics. These values have to be returned to the main program in the array "output[]".

**Some hints about C code**

Here is a very limited list of some useful things to know about C in case you are not familiar with this programming language. (Only things that are relevant to this practical!).

- In general, lines should end with a semicolon (;)
- Everything in between /* and */ is ignored: here you can write your comments about your code for later reference, etc.
- There are local and global variables. Local variables are only valid inside the function where they are defined (see below), but global variables are always valid (ie, globally). In the EBT, cohort no[0] is an example of a global variable.
- C is based on 'functions', which are routines or procedures in which you can make the computer do something (compute, read, write, sing, etc.). Functions can be called from within functions.
- Functions come in different types and have different types of arguments. For example the function Gradient:

```
void    Gradient(argument1, argument1, etc.)

{

       double               a, b;   /* declaration of doubles */
       int                  c, d;          /* …of integers */
       register int         i;             /* …of frequently used integers */


(... lots of code here ...)

       return;

}
```

  - The actual contents of the function is in between the { and }
  - The names and types of variables (except global variables) that will be used is the function have to be declared first (see 'double a, b' etc).
  - At the statement "return;" we return to where the function was called from.

- For loops

```
for(i=0; i<cohort no[0]; i++)

      popgrad[0][i][age] = 1.0;
```

  - This is a loop starting from i=0
  - For each value of i, the code in between { and } is executed,
  - after which i is increased by 1.
  - This is repeated as long as i<cohort no[0]

- i++ is the same as i=i+1, see also:

```
      eggs += pop[0][i][number]*E*pr; which means
      eggs = eggs + pop[0][i][number]*E*pr;
```

- floor((pop[0][i][age]+SMALL)/YEAR);
  - The function floor() rounds of to the nearest integer below the number in between parenthesis.

- If … then … else …
  For example the code:

```
if ((len > LJ) && (len <= lmax))
    newborns += Fr*len*len*pop[0][i][number];
else if (len > LJ)
   newborns += (Fr - KAPPA*len/LM)*len*len*pop[0][i][number]/(1-KAPPA);
```

is translated as:
Individuals with length between LJ and lmax are mature (and not starving) and hence they reproduce at rate `Fr*len*len*pop[0][i][number];`
This rate is cumulated for all cohorts i.
For individuals that are mature (len>LJ) but are starving (Len>lmax), the fecundity is `(Fr - KAPPA*len/LM)*len*len*pop[0][i][number]/(1-KAPPA);`
This rate, too, is added to the total population-level rate of newborn production (=newborns)

So in general: If the condition inside (...) is true, then the command is ex- ecuted, otherwise, we skip to the next command (the 'else' in this case). Everything behind 'else' is only executed if the condition inside (...) was false.

- note the difference between a == 1.0 (which checks the equality and is therefore true or false) and pr = P1 (which assigns P1 to pr)

- exp(x) means pow(a,b) means

- If you're interested in more, see the book 'Practical C' which should be present at the practical.