Max-Planck-Institut für demografische Forschung
Max Planck Institute for Demographic Research
Konrad-Zuse-Strasse 1 · D-18057 Rostock · GERMANY
Tel +49 (0) 3 81 20 81 - 0; Fax +49 (0) 3 81 20 81 - 202;
http://www.demogr.mpg.de

MPIDR WORKING PAPER WP 2008-010
MARCH 2008

# MAXIM

## A system for simulation of demographic processes in populations of related individuals

**Version 2.3**
**User and programmer manual**

Arseniy S. Karkach

# MAXIM

A system for simulation of demographic processes
in populations of related individuals

Version 2.3

User and programmer manual

Arseniy S. Karkach, 2005-2007

July 18, 2005
March, 2007

# Introduction

MAXIM belongs to the class of "microsimulation" population projection programs [ref. to Wachter] and is based on the ideas, and further develops the code of SOCSIM created by K. Wachter and collaborates [references about SOCSIM]. The MAXIM software development project was started by J. Oeppen and A.S. Karkach in January, 2005.

The name MAXIM commemorates Max-Planck Institute for Demographic Research where it was developed.

MAXIM is open-source modelling software ready for extension and modification. The program is written in C++ programming language, should compile and run under any operation system as a command line application with no or slight modifications.

MAXIM is a complete rewrite of SOCSIM with many enhancements. Still most of the syntax and algorithms are the same. Hence the user is expected to familiarize himself with the SOCSIM documentation first [………]. The difference between the two programs and new features of MAXIM are described in this document.

**MAXIM is not "back compatible" to SOCSIM**. Files prepared for simulations under SOCSIM **will not run on MAXIM**. Still they will usually require only slight modifications.

This manual describes the main differences between SOCSIM and MAXIM and how to set-up MAXIM for different model set-ups. The reader is assumed to be acquainted with the SOCSIM documentation beforehand.

## Basic ideas and areas of application

SOCSIM, which MAXIM bases upon, was originally developed for human populations but it allows simulating a much broader range of system.

MAXIM can be used to simulate 1- or 2-sex populations. In 1 sex set-up all individuals are treated as females.

Reproduction may be sexual and a-sexual (in 1-sex population) births can be legitimate or illegitimate.

Simulation involves estimating the time when all or some of the following events happen:

| | |
|---|---|
| **birth** | Creation of a new individual (happens to females). Changes information of mother and father |
| **marriage** | start of search of a partner of opposite sex and eventually marriage |
| **divorce** | split of a marriage |
| **cohabitation** | similar to marriage, with slightly different state change laws and possibly different rates |
| **split** | split of cohabitation |
| **death** | end of individual life |
| **transition** | migration. Realized as a change in "group" characteristic/ |

The meaning of these events may be re-interpreted, but new events can hardly be added.

Individuals can have arbitrary number of parameters which can eventually influence the rates with which events happen – for self or other individuals in the population (e.g. neighbours, kin). Genetic evolution, characteristics such as education influencing mating success, even infectious diseases that can spread in the population can be easily modelled with MAXIM.

Area of utilization of MAXIM covers and widens that of SOCSIM.

Modelling goes by discrete time steps.

The current state of each individual in the population can be fully characterized by its

**Basic variable:** age,

**Basic characteristics:** sex, marital status (which can be "single", "married", "divorced", "cohabiting", "widowed"), parity (total number of children born to a woman) and group number (integer value denoting belonging to one of the population groups),

**Predefined individual parameters:** fertility multiplier, etc.,

**User-defined individual parameters:** any number of constant or age-dependent characteristics which can be expressed as *double* values.

Above described events change the basic characteristics. For each individual of the population each time step occurrence of all possible events is tested based on the vital rates which may depend of base characteristics of ego, additional characteristics set by parameters, possibly current characteristics of other individuals and population-wide parameters.

Additional individual and population-wide parameters may change every time step.

Reproduction event can create only one offspring at a time. So simulation of systems with multiple offspring (e.g. birds and egg clutches) is so far not possible, but it is not too hard to implement it in MAXIM.

## Differences between MAXIM and SOCSIM

Most differences between the programs are given in the following table.

| | **SOCSIM** | **MAXIM** |
|---|---|---|
| Programming language and style | Initially written in Pascal, later ported to C. Functional | C++. Object-oriented |
| Event scheduling | Scheduling (evaluation of time) of next event. Event competition model | "Real time" testing for occurrence of event every time step |
| Modelling time step | 1 month | User-defined (integer months) |
| Population size | Limited by amount of RAM | Limited by amount of RAM |
| Simulation time period | Unlimited | Unlimited |
| Maximum age of individuals | 100 years, changeable | 100 years, changeable |
| Simulation | Proceeds by segments. Unlimited number of segments | Proceeds by segments. Unlimited number of segments |
| Life history rates | Can not change within a segment due to limitation of event-scheduling scheme | Can change within a segment since event occurrence depends upon the current rates |
| Definition of vital rates | Tables of age-specific monthly rates, Lee-Carter age- and time-specific rates. Ages denote **end** of the period when the rate acts | Age-specific tables of annual rates, ages denote **start** of the period when the rate acts. Rate-generator and altering plug-ins. "Rate patterns" allow to define similar rates (e.g. migra- |

| | | |
|---|---|---|
| | | tion rates for all groups) using asterisks (*). |
| Maximum parity | 16 | unlimited |
| Number of population groups (used as “families”, “tribes”, “countries” etc.) | Defined by a constant. 16 by default | unlimited |
| Additional parameters | Definition difficult. Only individual parameters | Simple definition in form of a rectangular table. Parameters defined by names. Individual and population-wide parameters. Can be easily set, initialized and updated during the simulation by the user-defined plug-ins |
| Support of comments in rate, population, marriage, additional parameters and command files | No | Yes (3 types) |
| Descriptive headers in population, marriage, additional parameter files | No | Yes, automatic |
| Restart ability | Yes | Yes |
| Execution | Each modelling segment is read, model executed, than the next segment is read | Commands and data for all segments is read into memory, then all segments are executed sequentially |
| Extendibility | Needs programming and good knowledge of the program | Needs programming. Plug-in framework:<br>4 types of user-defined plug-ins with standardized simple interfaces (3 types) and invocation:<br>plugins altering personal state. Invoked by life-history and simulation events (such as start of time period)<br>plugins generating vital rates (mx). Invoked when a specific rate is required<br>“statistical” plug-ins, collecting and dumping information about the population to files. Invoked periodically, dump information to files<br>Modifier plugins – functions changing output of rate generator plugins |

| | | |
|---|---|---|
| Robustness and user-friendliness | Original version provided by K. Wachter ran under Unix. After porting to MS Windows many annoying bugs that were previously masked or hard to find. Many memory-management errors were found. Error messages sparse and obscure. If a user provided wrong information is very hard to figure out what went wrong | Version developed under DEV C++ compiler under Windows contains enhances error codes and hints for the user. New code contains increased number of comments. Increase number of checks against errors in data, user input or OS issues (e.g. file accessibility). Support of directories, multiple modelling experiments |
| Suitability | Fast, suitable for slowly-changing segment-constant vital rates, simple modelling set-ups, modelling times much longer that individual life span | Slower, more precise, suitable for quickly changing vital rates, modelling of population density loop-backs and other "real time" changes that influence the individual life history between events, complex set-ups with many individual parameters (e.g. mutation-selection problems) etc. |

## Stochastic event model

In order to model which event happens to an individual and when SOCSIM uses model of event competition with scheduling of next events. In simple form it works as follows. Vital rates are unchanged within one modelling segment. So the possibility of events happening to any individual being in a certain state (age, marital status etc.) is defined by a known before hand set of rates corresponding to their age, sex, group, marital status, parity etc. till the next event. Ages of all possible events that can happen to the individual are evaluated and the closest event is selected. This is called an "event competition model". Usually an event changes the status of the individual (e.g. marital status, or parity), and, hence, the set of possible events for the individual and corresponding rates. Each time an event happens to an individual "event competition" is repeated and a new event in scheduled. Events are rescheduled and again the next closest event is selected. The process is repeated until the death of the individual.

Such system is computationally effective, since each month only individuals with events scheduled for that month are processed. These usually comprise only a small fraction of all population.

The drawback of such event scheduling model is inability to adjust the rates dynamically. For an example, consider a child born to a mother. At birth a next event is scheduled for the child based on vital rates acting at births, suppose this is the "marriage" event. If his mother dies soon, most probably, his mortality rate strongly increases. Although in SOCSIM there is no way to influence his life history before the next scheduled event, in our case, "marriage", happens. So the death of mother will never lead to higher mortality of children.

More important events that change mortality rates, such as epidemics, wars, can only be modelled by splitting the timeline into "segments" with different rates, in the modelling. Modelling of rates depending on kin, resources and such parameters changing unpredictably is impossible in SOCSIM.

In MAXIM all individuals alive in the population at current time step of simulation are tested against all possible events, based on rates of events depending on their current states (and possibly, states of other members of the population and parameters). This requires much more computation time, but the system become more “responsive” to changes.

## New features of MAXIM

### Improved usability

Files describing model and rates (**.sup** files), population (**.opop** files), marriage and cohabitation unions (**.omar** files) and additional parameters of individuals (**.opox** files) may contain comments. A comment in MAXIM is a line or part of the line starting from one of characters: **#**, **%**, or from //. Moreover, **.sup** file parser supports comments in any place of the file and at the end of the lines with commands. In **.opop** and **.opox** files comments may be only in the beginning of the file.

Result .opop, .omar and .opox files generated by the program contain headers which explain the file structure and contain additional information about the simulation.

Additional parameters defined for individuals in .opox files and for the population in .sup files are referenced by names.

Data checking and error reporting are enhanced in comparison to SOCSIM. All errors report names of functions in which they occur thus simplifying debugging.

### Support of complex models

Modelling of every life history process requires a different approach. For example, mortality may be described by age- or age- and time-specific mortality rates, which may be defined in the program by stepwise rates by means of a table, or in parametric from by formulas. Fertility is a process with totally different rules which may be described by age-specific or period-specific birth rates, birth intervals, ages of start of reproduction and age-specific sterility factors. Mortality of children and female fertility may be influenced by different factors such as available resources.

It is impossible to implement all possible models for different life history processes keeping the modelling ideas simple and the program robust. A modeller needs a flexible, yet simple, event-simulation framework which can be extended by new modules that take into account specifics of certain processes and implement certain models of them.

MAXIM offers such possibility since it can easily be extended by means of plug-ins.

MAXIM plug-ins are functions with a simple standardized interface which may be added, compiled into the program and invoked by simple commands in .sup files.

Plug-ins that come with MAXIM and how to create the new ones is explained in section “Extending MAXIM. Programming plug-ins” of this manual.

## How it works

When started MAXIM reads command file with extension **sup** which describes the model. It defines

- how long does the simulation do and what is the time step
- which files describing population use for input and output
- describes rates transitions between the states of individual by means of rate tables or rate generating plugins (compiled into the program beforehand). Command file can include other files (e.g. tables of age-specific rates). It also describe what statistics about the population should be collected and written to files
- Information about the starting population consisting of information about individuals, their unions (marriages, cohabitations) and values of individual parameters are read in
- The modelling processes by defined time step. Some error and diagnostic information is printed to the screen and sent to log file during the simulation. Estimated time till the end is displayed
- After the end of the simulation information about all individuals ever lived during the simulation, their kin relations, marriages and parameters is written to a file

MAXIM supports 5 marital statuses of individuals: **single**, **married**, **cohabiting**, **divorced** and **widowed**. Possible transitions between them are shown on Fig. 1.

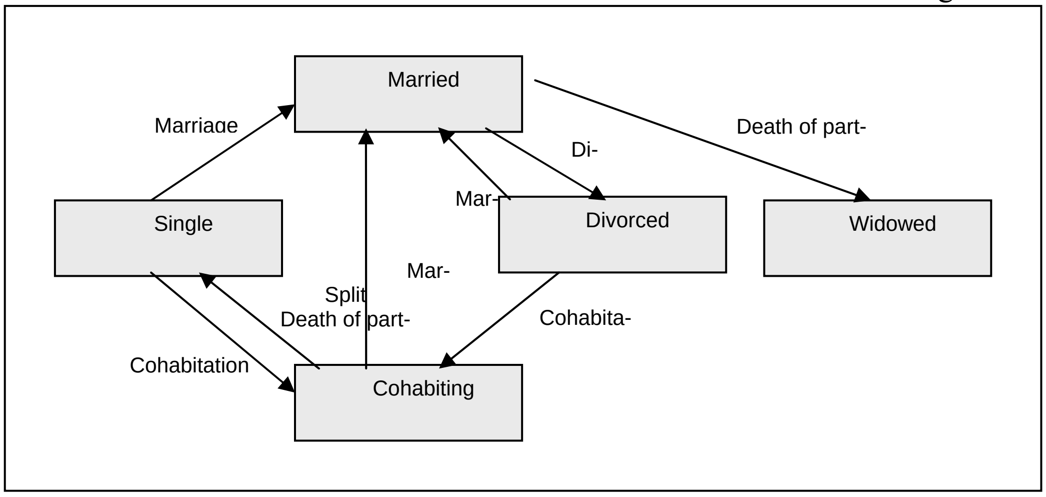


Figure 1. Marital statuses and possible transitions between them in MAXIM

## Compiling

MAXIM distribution contains the source code, precompiled binary for MS Windows system (file MAXIM.exe) and example model files. You can start experimenting with MAXIM under MS Windows immediately using the binary.

In order to run the program under other operation systems or if you decide to extend the program by adding plug-ins you will need to recompile it. This can be done from the source code using any C++ compiler. MAXIM is a command line program that does not use any system-specific functions and (theoretically) should compile and run under any operation system. Build it as a command prompt C++ program. Consult the manual of your compiler how to do it.

MAXIM makes heavy use of the Standard Template Library. Your compiler should support it.

## Running

MAXIM executable can be invoked from the command prompt

```
MAXIM sup_file rnd_number left_trunc right_censoring [/v]
```

| | |
|---|---|
| `sup_file` | Filename describing mode of work and rates of the life processes. These files should have a .sup extension. sup_file is the name without extension. This name is also used to generate names for log files. Name may be given relative to the directory of maxim executable. In this case all input and output files referenced from the .sup file are searched for relative to this directory |
| `random_number` | Arbitrary positive integer number – initial seed of the quazi-random number generator used in simulation |
| `left_trunc` | Integer denoting month. Used as the start of simulation. All individuals in the input population file not marked as dead are assumed to survive till this date |
| `right_censoring` | Integer denoting month. Not used |
| `/v` | Switch on verbatim mode, print abundant runtime information. Optional. |
| `/l` | |

Example

We want to run a simulation described by file `evolution.sup`. Input population contains information about the population up to time 1200 months (end of the previous simulation). All files are located in directory `simulation`. A command

```
maxim simulation/evolution 1 1201 2400 /v
```

run a simulation starting from month 1201. Length of the simulation and all parameters are defined by file `simulation/evolution.sup`. All results are written relative to directory simulation.

## The command (supervisory) (.sup) file

Command files have extension .sup and describe mode of work, parameters, rates and most information required for the modelling. These files were referred to as supervisoru (hence the name) in SOCSIM documentation.

The format of files is almost back compatible, so SOCSIM files may work in MAXIM with some modifications.

Unlike SOCSIM in MAXIM data for all segments of the multi-segment simulation are first read into memory at start and then the simulation starts.

### Overview of top-level structure of the file

Command files are easiest to understand as nested structures. The outermost part gives information that will characterize the entire simulation--the rate and population files used and the parameter settings--and the control commands. It's probably a good idea to set all parameters in the top level file and provide source information (as comments) for all lower-level files (which typically contain rate sets), just to make sure that the simulation can be reproduced.

Any number of *simulation segments* can be nested within this outermost structure. A "simulation segment" refers to a period of simulated time characterized by specific rate regime and parameters (hence the close association between segments and distinct rate files). For example, one might wish to model a population that experienced 100 years under a high fertility/high mortality regime and then underwent a demographic transition to low fertility/low mortality rates. Such a simulation may be set using two simulation segments. Modelling the kinship structure of a country with a census (and a new rate set) every 10 years could be done with as many 10-year segments, each with a distinct set of rates, as necessary.

In addition to changing the demographic rates, each segment may also be governed by a distinct set of options and variables which govern the behaviour of the simulated population during that period. For example the average inter-birth interval, the ratio of male/female births, or the level of heterogeneity of fertility may change from simulation segment to segment. These should be set in the highest level file, as this localizes required changes and makes it easier to keep consistent.

```
include filename
```

commands allow to include other text files. These have the effect of splicing the named file into the input stream at that point. This works for any files but is most useful for rate tables, which tend to be quite long, full of numbers, relatively static once they are created. (The top level command file could simply have contained all included files directly but that would have made it somewhat harder to use - it also would be much longer and much more difficult to understand.)

Filenames may include directories.

The example below sets up a two-segment simulation: segment 1 is 480 months long and is governed by rates stored in file "RATES/rates.1840.1879" and parameters **bint** and **hetfert** set in the toplevel file. Segment 2 is 360 months long and is governed by rates stored in "RATES/rates.1880.1909" and parameters **bint** and **hetfert**set in the toplevel file.

## Comments

In MAXIM command files comments may be added using #, % or // (C-style) characters. Any line or part of line starting with any of these characters will be treated as a comment and not parsed.

*Note: the character* ***** *which was used to denote comments in SOCSIM should not be used any more as it now denotes wildcard ("any") in rate patterns.*

## Example of a .sup file

```
segments 2
input_file test
output_file test.out
duration 480
bint 12
hetfert 0
include RATES/rates.1840.1879
run
duration 360
bint 24
hetfert 0
include RATES/rates.1880.1909
run
```

This is an annotated file version of the file shown above and would be processed identically. The rate-set provenance comments would be appropriate even in an otherwise uncommented file:

```
# number of segments in the simulation
segments 2            # segment command is obsolete in MAXIM
                      # but still can be used

# input file prefix. Read files test.opop, test.omar,
# in the current directory
input_file test

# output file prefix. Write files test.out.opop, test.out.omar
# in directory */tmp.
output_file /tmp/test.out

# set up segment 1
# duration of segment 1
# model to approximate 1840-end of 1879
duration 480

# birth interval setting for segment 1
bint 12         # 12 months

# hetergeneous fertility setting for segment 1
hetfert 0

# file containing the birth, death, marriage and divorce rates
# for segment 1 is rates.1840.1879
# in the directory RATES, which is one below this one
include RATES/rates.1840.1879

# save information for the first segment of the simulation
# and continue reading this file
run

# Set up segment 2
# duration of segment 2
# model to approximate 1880-end of 1909
duration 360

# birth interval setting for segment 2
bint 24

# hetergeneous fertility setting for segment 2
hetfert 0

# file containing the birth, death, marriage and divorce rates
# for segment 2
# is rates.1880.1909 in the directory RATES,
# which is one below this one
include RATES/rates.1880.1909

# Ends defining the second segment of the simulation.
# Segment 2 is the last segment so the modelling will start now
run
```

SOCSIM .sup files may work in MAXIM with some modifications. Most variable names and table structure of SOCSIM should work in MAXIM (see note about the rate table structure)

MAXIM parser is more robust than that of SOCSIM. It allows for any level of recursion in the .sup files, tables can be "cut" between files in any place (see limitations below). More comment delimiters are supported.

MAXIM .sup file parser works in 2 steps.

Step 1 unfolds the inclusion of .sup files and presents all the information as one piece of text. During this process all empty lines and comments, beginning with symbols **#, %** and // are stripped. This allows to use comments starting at the end of lines. E.g. lines

```
death 1 F single      # death rate for single females in group 1
0   1     0.010000    # rate acts in ages 0 and 1 complete month
```

will be first stripped by preparser to

```
death 1 F single
0   1     0.010000
```

and then parsed.

Step 2 parses the information from the .sup files and loads it into **variables** and **rate tables** for one or several **segments**.

In SOCSIM each **segment** contains the information about parameter values and rate tables fixed and acting on certain time interval of simulation. This idea remains in MAXIM although the rates have possibility to vary within segment (see further). In SOCSIM the number of segments is defined by command **segments**, information for each segment is terminated by a **run** command. Thus a .sup file

```
…..
run

…..
run
```

defines 2 segments.

MAXIM parser ignores the **segments** commands used to define number of segments in SOCSIM and uses just **run** commands to delimit the segments. Each **run** command defines marks the end of the definition of the current segment. Number of segments corresponds to the number of encountered **run** commands. Hence the .sup file defining only one time segment must have **run** command at the end.

If for a simulation more than one segment was defined, the parses acts as following:

The information from the first block (parameters, rate tables) until the run command is read and saved for the second segment.

## “Inheritance” of control information between the segments

The rules of inheritance are different in MAXIM. They are rather simple.

Everything - the values of parameters (built-in, population-wide), defined rate patterns, rate tables, link of rate patterns to rate table and plugins, … acting in the segment - are inherited by the next segment. So if you desire to change only one or several parameters you don’t need to worry about all the others.

Information read for the second segment updates one set for the first segment. This allows changing only values of some parameters or replacing only some of rate tables. Each following piece of data between **run** commands updates information of the previous segment, and the updated information is saved into the new segment.

Of course you may as well define all parameters and rates for a new segment. An example may clarify the subject. Consider a code fragment:

```
…
bint 9
hetfert 0
…
run

…
bint 20
…
run

…
bint 30
hetfert 1
…
run
```

This defines 3 segments. In the first **bint=9**, **hetfert=0**, in the second **bint=20** and **hetfert** remains **0**, in the third both parameters get new values: **bint=30, hetfert=1**.

## Structure of rate tables

Age-specific (and other???) rate tables are stored in MAXIM in a more natural format different from somewhat awkward and error-prone format of SOCSIM. The MAXIM format implies that the given **rate starts at the defined month**. The **last rate may be given for any age**, it is assumed to be acting further on.

The rate definition should start at age 0 otherwise an error message is issued.

```
death 1 F single
0   0     0.5
0   2     0.10000
1   0     0.08948
2   0     0.03314
3   0     0.01105
4   0     0.00601
…..
```

```
100 0     1.0
```

The rate that the table defines is coded in its header as “event group sex marstatus” and “parity” for birth rates.

The first number in every line with numbers denotes year, the second – month of the **first date when the current rate starts**. The third number defines the **monthly rate**. In this example monthly mortality rate 0.01 will act during life months 0 and 2 till age 11 months, then rate 0.008948 will start at age 1 year etc.

Such form is comfortable since in the rate tables typically found in the literature time defines the **moment starting with which the rate acts**.

## Keywords and basic model parameters

| Keyword | Interpretation | Examples | Notes |
|---|---|---|---|
| **General control** | | | |
| **input_file** | input files prefix - actual files will be name `prefix.opop, prefix.omar` | `input_file test1` `input_file testruns/test1` **Explanation: in the first case, the starting population will be in file test1.opop and starting marriage file will be in file test1.omar (both in the current directory). In the second, the files have the same names but are in directory testruns (a subdirectory of the current directory).** | Analogous keyword: **output_file** |
| **segments** | number of segments expected in the simulation. There isn't any limit on the number, but segment information must be provided for each segment. | `segments 1` **`segments 78`** | *Obsolete in MAXIM. End of segments (and hence, their number) is defined by the “run” command* |
| **duration** | Assign the integer following to the integer variable duration_of_segment, which is the duration of the current segment in months. | `duration 10` `duration 1200` **default value: 0 (this triggers an error message)** | |
| **birthtarget** | interpretation: Modify rates to get a number of births close to the value | `birthtarget 2 100` `birthtarget 1 1500` | caveat: it's not always possible to to achieve too |

| | | | |
|---|---|---|---|
| | in target in the given segment. When targets are specified a crude calculation of the expected number of births in that segment (based on the rate structure and the parity and age distributions) is used to scale the birth rates by a fixed factor for all individuals in a particular group. The rates themselves are unchanged and can be used in later segments. Note that some feedback is possible while the simulation is running--the target can be read from a file created using the **execute** command. | | high a target by increasing birth rates alone--the age/sex structure of the population may not be favorable. *Note: is not implemented in MAXIM. May be implemented by a user plugin.* |
| **include** | Read the file that follows as though it's contents were spliced into the current file from that point on. | examples: `include rates.segment1` **`include rates/mortality.seg4`** | **`Directory names may be used in the names`** |
| **execute** | interpretation: interpret the rest of the line as a string to be passed to the UNIX "system command." Control is transferred to the shell and reading doesn't continue until the command is executed and returns. Commands that produce screen output will place their results at this point in the output stream. Commands that generate ratefiles will generate the file before reading the next line (which can be a command to include that file, as above) | example: `execute ls` `execute generate_rates 1 5 0 >mortality.seg4` `execute sleep 10` `execute date` `execute echo` <br><br> `this section isn't very clear` <br> explanation: ls and date are Unix commands and the output will appear on the screen. sleep 10 will pause for 10 seconds, and the echo command will print out the disclaimer. If the line including generate_rates is a correct command line (and generates | **`Not yet implemented in MAXIM`** |

| | | | |
|---|---|---|---|
| | | rates in the proper form for socsim), the results of the command will be in file mmortality.seg4 and eligible for inclusion by the next line. | |
| **run** | interpretation: run the next program segment. This command should only be in the "highest" level file, i.e., in the file that is named on the command line. This top-level file is bookmarked at the next reading position (so reading can resume at the end of the segment), the rate file is closed, and the segment processing begins. | `run` | **`End definition of data for the current segment`** |
| **Rate patterns** | | | |
| * | Used in definition of rate patterns. Replaces any possible value in its position | **`death 1 * married`** – defines death rate for married individuals of group 1 and both sexes | |
| **M** | Used in definition of rate patterns. Defined rate pattern applies to males. | **`death 1 M married`** **`marriage 1 M single`** | **`Analogous keyword:`** `F` |
| **single** | Used in definition of rate patterns. Defined rates apply to marital status "single" | `birth 1 F single 0` `death 1 M single` **`marriage 1 M single`** | Analogous keywords: **divorced married widowed [cohabiting]** |
| **death** | The rest of the line will represent the group, sex, and marital status of a set of agents in the population. The rate set or output of plugin that follows will apply to them and to all agents whose rate sets default to this set.<br>Within the simulation the rates may be duration-specific. For example, | `death 1 M married` `death 1 F divorced` | analogous keywords: **marriage divorce**<br>*Note: in MAXIM an extra parameter in the line may denote name of the rate plig-in. It replaces the definition of the rate by a rate table* |

| | | | |
|---|---|---|---|
| | divorce rates are based on the "age" (duration) of the union. | | |
| **birth** | The rest of the line will represent the group, sex, marital status, and parity (which starts at 0) of a set of female agents in the population. The rate set or plugin that follows will apply to them and to all agents whose rate sets default to this set.<br>If rate is defined by a table the next lines will be interpreted as rate table. | `birth 1 F married 0`<br>birth 1 F married 14 | |
| **transit** | The line will represents rate pattern of a “transition” event which changes “group” of the individual. Follow: the starting group, sex, marital status, and destination group of a set of female agents in the population.<br>The rate may be defined by a rate generating plugin (following position – its name) or a rate table in the next lines. There is no need for a complete set of migration rates, allowing this sort of rate to model general status transitions. The maximum number of groups is unlimited in MAXIM | `transit 1 F married 2`<br>`transit 3 M single 2`<br><br>The first example gives the migration rates for group 1 married women who are migrating to group 2; while the second applies to group 3 single men who migrate to group 2. | |
| | | | |

## Inheritance of group

A newborn individual inherits its population group from one of its parents. This is controlled by a keyword

**`child_inherits_group`**

with possible values

**`MOTHER, FATHER, SAME_SEX_PARENT, OPPOSITE_SEX_PARENT`**

**Example:**

if .sup file defines for the current segment
`child_inherits_group OPPOSITE_SEX_PARENT`
a newborn girl will inherit her group from her father if he is known. Else she inherits the group from her mother.

### Inheritance of additional parameters

A newborn individual inherits the set and values of additional parameters from one of its parents. This simplifies programming of models with additional parameters. The values of parameters may be further adjusted by plugins hooked to the **`birth`** event which occurs just after the creation of a new individual.
The inheritance behaviour is controlled by keyword
**`child_inherits_parameters`**
with possible values
**`MOTHER, FATHER, SAME_SEX_PARENT, OPPOSITE_SEX_PARENT`**

**Example:**
if .sup file defines for the current segment
**`child_inherits_parameters FATHER`**
a newborn boy will inherit values of all individual-level additional parameters defined for his father if he is known. Else he inherits them from mother.

### Inheritance of heterogeneous fertility

It's useful to have a mechanism that allows the daughters of large families to be more likely to have large families themselves. Alphas and betas are used in pairs (.1/1.05, .2/1.10,.4/1.15, etc). The values default to those that are there, effectively, when there is only the heterogeneous fertility setting: alpha = 0, beta = 1.

**alpha**

interpretation: proportion of an individual daughter's fertility multiplier--when heterogeneous fertility holds sway--that is inherited from her mother. The rest come from a call to a funtion that generates random numbers with a beta distribution. Or as in the code:

```
   daughter->fmult = alpha * mother->fmult + (1 - alpha) * fert-
mult();
```

example:

```
alpha .1
```

default value: 0 (reassigned at start of segment)

**beta**

interpretation: once the child's fertility multiplier (daughter->fmult) is established, it can be modified using the code

```
daughter->fmult = 2.5 * exp(beta*log(daughter->fmult/2.5));
```

example:

```
beta 1.05
```

default value: 1 (reassigned at start of segment)

**factor**

interpretation: modify a particular set of rates (a single event, group, sex, marital status index into the rate set) by the given value. In the case of birth rate sets, all parities are modified by the value for the given group and maritial status.

example:

```
factor birth 1 single 1.1
factor death 1 M single .9
factor transit1 2 M single .9
```

default value: 1 (reassigned at start of segment)

*Note: not implemented in MAXIM*

**Commands setting values of predefined parameters**

| **Key-word** | **Interpreta-tion** | **Assigns value of parameter** | **Possible values, de-fault** | **Examples** | **Notes** |
|---|---|---|---|---|---|
| **sex_ratio** | Percentage of births that are male. | **`prop_males`** | **`0..1. 0.5112 (reas-signed at the start of a new segment)`** | `sex_ratio 0.5112` `sex_ratio 0.3333` | |
| **het_fert** | It's reassigned at the start of the simulation and each segment (as are the analogous keywords which follow). Note that turning the flag on and off is not a good idea. | assign a value to het-fert, | 0(FALSE)/1(TRUE) (flag) default: 1 | `hetfert 1` `hetfert 0` | analogous keywords: hhmigration (migrants emi-grate with their entire house-hold, default 0) endogamy (marriage within groups, only, default 0) |
| **bint** | A double pre-cision value rep-resenting the minimum inter-val between births (in months). Birth rates are then adjusted (in-flated) to keep the intended value for the monthly prob-ability of giving birth, however, when the bint is too large it's pos-sible that some births will be lost | **`bint,`** | | `bint 9` **`bint 24`** | Acts only if birth rate is given by age-specific table of rates. Does not act when rate is given by a rate generating plugin |
| **timestep** | Modelling time step in months | model-ling_time_step | 1 | | |

## Support of the Lee-Carter mortality model (not implemented in MAXIM)

Lee-Carter mortality uses several parameters to create on-the-fly rate sets based on the age of the individual at the time of the event competition and the "year" in the simulation the competition occurs. The format for parameters **ax** and **bx** is similar to ordinary rate sets, while the **kt** format is completely different. In particular, the **kt** can be specified using arbitrary start and stop points (so there must be support for delimiters), or by initial values and a formula (which requires specification keywords). All parameters (including one whole set for some version of the **kt** values) must be specified before a set is "complete." It's also necessary to inform SOCSIM of the correspondence between simulation month and "year" (e.g., 1990). The formula to use when extrapolating beyond the values provided is computed from the values provided, unless all values are specified by formula from an initial value supplied in a **kc_k_val** line. A simulation may run until year "2030," but SOCSIM has to have 100 years of event horizon at all times: the "year" parameters must be available for years up to 2129 for use in the last running year of the simulation. Lee-Carter rates are broken up by sex and group, only. The starting year is specified by an parameter to the lc_init keyword. This means that the entire set of Lee-Carter parameters and specification can be kept in one file and read -- via the "include" keyword in the top level command file-- during each segment, with a different start year (corresponding to the point in the simulation) specified in a higher level file (that is, higher with respect to nested reads).

**lc_init**

interpretation: the mortality model for this population class will be Lee-Carter. This command sets a flag that triggers Lee-Carter completeness checks (and doesn't doesn't complain about the absence of ordinary death rates) and allocates space for the new rate set components. If there is an extra argument, interpret this as the starting year if the kt values are read in. (Otherwise, assume that the values will be generated using a formula). In practice the init command can be at the top level while the rest of the LC specification can be in a separate file. The start year option allows the same file--with hundreds of kt values--to be included, unaltered, for each of several segments. In that case, the init commands should precede the inclusion commands. The command can appear in the included file, too, without ill effect and without the need for a start year ( only the first start year is recognized, and one must be specified before the list of kt values can be processed).

example:

```
      lc_init 1 F 1938
      lc_init 1 M 2030
```

**lc_ax**

interpretation: the ax values follow: these are specified in the manner of other rates, even though they're parameters

example:

```
      lc_ax 1 F single
```

analogous keyword: **lc_bx**

**lc_k_val**

interpretation: Initial value to use when the **kt** values are computed using the mean, standard deviation, and a starting value, base on the iterative formula:

```
  kt_value[i+1] = kt_value[i] + lc->mean + lc->std_dev * normal();
```

The linear continuation after the last value is read is similar, without the standard deviation term.

example:

```
  lc_k_val 1 F -3.5123
```

**lc_k_mean**

interpretation: the lc->mean, as above. Mean difference between successive **kt** values.

example:

```
lc_k_mean 1 F -.31
```

**lc_k_std_dev**

interpretation: the lc->std_dev, as above. Standard deviation between successive **kt** values.

example:

```
lc_k_std_dev 1 F 0
```

**lc_k_start_list**

interpretation: Begin reading **kt** values. These values have a different format which would be incorrect in any other context (while typically correct ones are wrong) so there is a need to inform the error checking mechanisms.

example:

```
lc_k_start_list 1 M
```

**done**

interpretation: end of the list of supplied kt values. Additional values will have to come from extrapolation (SOCSIM uses a linear continuation).

example:

```
done
```

## Rate patterns

Simulation of demographic processes even in simple cases will require defining many rates of transitions between various states in which individuals can be.

The modeller should define ALL necessary rates in order for the model to work. For example if births are allowed in the population one should define birth rates for all parities. Similarly death rates for all possible states should be defined.

The model will not run until one defines rates for all possible transitions.

Experimenter may choose to define some rates as similar. In SOCSIM maximum parity and number of groups are predefined and so a set of possible states of the organism is always limited (and equal to #events * #marital_statuses * #sexes * #parities * #groups). Thus is was possible to define only some "major" rates and the other missing rates were duplicated from them before the simulation. This was done by a clever, but complicated and fixed algorithm.

In MAXIM the number of parities and groups and, hence, states, is unlimited. Thus one can not create all possible rates. Moreover, we wanted to give the user more freedom in defining the rates.

A new idea of "rate patterns" is implemented in MAXIM. A rate pattern is a description of one or more rates of transitions between states done with the use of wildcards.

A rate is defined by a combination of event, sex, marital status, group and parity. A wildcard (* symbol) can replace any value in a given position.

All rate patterns have the same syntax:

**`[event] [group] [sex] [marital status] [parity]`**

where

**`event`** is one of words {**`birth, death, marriage, divorce, cohabiting, split, transition`**}

**`group`** is integer **`1, 2, …`**

**`sex`** is **`"M"`** or **`"F"`**

**`marital status`** is one of letters {**`"s", "m", "d", "c", "w"`**} standing for single, married, divorced, cohabiting, widowed correspondingly.

**`parity`** is integer **`0, 1, …`** (parity is the number of ever born children, so first child is born at parity 0)

A wildcard (*) can be used in any or all positions of the rate pattern and will denote ANY value in the corresponding position.

Example: rate pattern

```
death * F * 0
```

will denote mortality rate for females of any group, any marital status with parity 0.

Example:

```
* * * * *
0 0 0
```

defines zero rates of all events that may happen. Given such rates only the individual will age with their states unchanged and die when they reach maximum age allowed in MAXIM (100 years).

## Choosing relevant rate pattern

Patterns may define intersecting ranges of rates. For example birth rates can be defined for all parities

```
birth * F * *
………
```

and specifically for parities 1, 2 and 3:

```
birth * F * 1
………
birth * F * 2
………
birth * F * 3
………
```

When a program needs a specific rate it looks for all patterns relevant to this rate and selects the most specific pattern – the pattern with least wildcards (*) or first one from several patterns with the same number of wildcards.

If in our example MAXIM will need a rate for parity 2 if will find 2 relevant patterns: **`birth * F * *`** and **`birth * F * 2.`** Of them pattern **`birth * F * 2`** will be chosen since it has less wildcards.

## Included rate tables

It's obviously easier to separate sets of rates by segment and keep all the commands in a single "command file" (even though they could all be jointed together in single large file). The rates can then be put into arbitrarily many sets of other files. This is made possible by the **include** command. The command has a single argument--the name of the file--and treats the contents of the named file as though they were spliced into the current file at that point. The command can nest--the included file can include others in turn. For example, all individual segment rate files can include a one file which contains one set of Lee-Carter parameters.

*Note: Lee-Carter rates are not implemented in the current version of MAXIM*

For example, this can be the file named cn the command line when the simulation is run:

```
# segments 3
input_file test
output_file test.out

# segment 1
duration 120
```

```
bint 12
hetfert 1
include RATES/rates.1950
run

# segment 2
duration 120
bint 12
hetfert 1
include RATES/rates.1960
run

# segment 3
duration 120
bint 12
hetfert 1
include RATES/rates.1970
run
```

In addition, each of the rate files can contain lines to include other files. One reason there is a specified start year for the Lee-Carter mortality model is so that the parameter file doesn't need to be modified when it is used for several different segments--the same file can be read unchanged using the **include** command but the point at which the **kt** values become relevant can be specified. Accordingly, in file *RATES/rates.1950* expect to find lines like:

```
lc_init 1 F 1950
lc_init 1 M  1950
include lee_carter_us
```

and in *RATES/rates.1960*:

```
lc_init 1 F 1960
lc_init 1 M  1960
include lee_carter_us
```

and finally, in *RATES/rates.1970*

```
lc_init 1 F 1970
lc_init 1 M  1970
include lee_carter_us
```

The same initialization can be done at an even higher level, in the top-level .sup file:

```
lc_init 1 F 1950
lc_init 1 M  1950
include RATES/rates.1960
run
...more lines
lc_init 1 F 1960
lc_init 1 M  1960
include RATES/rates.1960
run
...more lines
lc_init 1 F 1970
lc_init 1 M  1970
include RATES/rates.1970
run
```

## Lee-Carter Rate Files (not implemented in the current version)

Lee-Carter **kt** values can be read in via a file (with eventual linear continuation, as necessary) or specified using a formula. The program checks to determine whether a complete set--in either form--is specified. In all cases it is necessary to set up Lee-Carter mortality structures for each group and sex modeled using a separate rate set using the **lc_init** command, analogous to what is shown in the previous section. The year to use as the index into the **kt** array must be specified here:

```
lc_init 1 F 1990
lc_init 1 M  1990
```

SOCSIM must find one of the following sets before the next run command (it can find one form for some of the rates, and the other for the others). The **ax** values have to appear before the corresponding **bx**. It's not an error if the start year is after the last specified **kt** value--it will be within the range of the linear continuation.

### Case 1: The kt values are given by a formula

```
lc_ax 1 F
.. the values
 lc_bx 1 F
.. the values
lc_k_val 1 F  -12.17
lc_k_mean 1 F -.496
lc_k_std_dev 1 F 0.651
```

### Case 2: The kt values are read via the rate file

```
lc_ax 1 F
.. the values
 lc_bx 1 F
.. the values
lc_k_start_list 1 M
1900    18.3796
1901    17.8665
1902    17.0566
1903    17.0231
...many years
1988    -10.1178
1989    -10.7226
everything before 1990 will be ignored
1990    -11.10
1991    -11.47
1992    -11.85
..many more years
2061    -37.71
2062    -38.08
2063    -38.46
2064    -38.83
2065    -39.21
done
```

## *Defining vital rates and loading data tables*

Several possible model applications require using data in table form. For example transitions of individuals between different life states may be described by life tables

– age-specific tables giving correspondence between age and lx, qx or mx values for a certain process.

MAXIM can use life table data given in form of lx, qx or mx (annual rate) values. The user should specify what type of data is supplied.

Transitions between states may also be described by a certain model (usually parameteric). Such a model may be implemented inside a function written by user in form of a MAXIM generator/alternator plugin. The plugin will be used to generate transition rates for certain transitions and is expected to return  annual rates for the process (mx values).

Sometimes in a model lx, qx or mx values obtained from a table or mx values returned by a plugin need to be modified based on some laws and possibly values of population-wide or individual parameters – e.g. scaled or shifted. This can be most easily implemented using alternator plugins.

A generator plugin returning mx values for some rate pattern or alternator plugin changing parameters may contain a complicated algorithm inside and require some age- or period-specific data. It is convenient to load such data from the .sup file. General data tables containing arbitrary data in the time-value format (year month value triples) can be loaded from .sup files, stored in the segment and further referenced from plugins by names.

The following describes all possible ways of loading data into MAXIM and possible applications.

## Notation

We denote any possible rate pattern by `* * * * *`. This does not necessary mean the "any rate" which is defined by 5 wildcards, but can be any specific pattern like **`birth 1 F single 2.`**

**`{…}`** denotes arbitrary commands that may be absent

**`[…|…|…]`** denotes a command that should be present in one of variants

```
0   0     0
:   :     :
0   0     0
```

denotes age- or time-specific data in form of **`year month value`** triples. These should not be necessary zeroes. Limitations for specific types of tables are given in the end.

Other words are keywords – they should be typed exactly as shown.

## Rates defined by a life table (qx, mx, or lx values)

```
* * * * *
{type [qx|mx|lx|}
0   0     0
:   :     :
0   0     0
```

The command orders MAXIM to describe the processes corresponding to the rate pattern by data from the given table. Data is treated as lx, qx or mx life table. If type command is unspecified data is treated as values of annual mx rates.

## Rates defined by a plugin

```
* * * * * plugin pligin_name
```

The command orders MAXIM to describe the processes corresponding to the rate pattern by mx annual rate generated by the plugin_name plugin.

## Arbitrary data defined by tables

table name
```
0   0     0
:   :     :
0   0     0
```

The command defines a table “name”, stores it in the data for current segment. Data is treated as **`year month value`** triples. Values can have any meaning, not limited to rules for lx, qx or mx rules. Time specifications should be increasing by rows. Data is treated as a step-wise value(time) function.

A plugin (state alternator, rate generator, rate modified, statistical) can get a value from a loaded table using function
**`double table_lookup(string table_nema, int age_months)`**.
Note: if data is given as time-specific, addressing still goes by months.
E.g. for a table defined as
```
table table1980
1980 0 0.01
1980 1 0.02
1981 0 0.03
```
one will obtain the following results:
```
table_lookup(“table1980”, 1980*12) == 0.01
table_lookup(“table1980”, 1980*12+5) == 0.02 etc.
```

## Modification of rates given by life tables

```
* * * * * mod modplugin_name
{type [qx|mx|lx}
0   0     0
:   :     :
0   0     0
```
Here a plugin modplugin_name will process all values requested by the table for given rate pattern.

## Modification of rates defined by a plugin

```
* * * * plugin pligin_name mod modplugin_name
```
or
```
* * * * * mod modplugin_name plugin pligin_name
```

The values returned by a plugin_name will be passed through the modplugin_name and then treated as mx annual rate to calculate rates of transition given by the rate pattern.

This can be summarized to the following general definition

## Transition rates – general definition

```
* * * * * {mod modplugin_name} [ plugin plugin_name |

{type [lx|qx|mx|}
0   0     0
:   :     :
0   0     0   ]
```

## Comments

1. All tables and, generally, any text of a .sup file, can be loaded from nested files using **`include`** command. E.g. the following construction:

file data1.txt containing

```
1980 0 0.01
1980 1 0.02
1981 0 0.03
```

together with a command in a .sup file

```
table table1980
include data1.txt
```

is equivalent to expression

```
table table1980
1980 0 0.01
1980 1 0.02
1981 0 0.03
```

2. No modification plugins are allowed for general data tables given by table command, so expressions like

```
table table1980 mod modplugin_name
1980 0 0.01
1980 1 0.02
1981 0 0.03
```

are illegal.

3. Models when life table data is obtained from tables, transformed and then used for a rate pattern can be programmed in 2 ways:

a) by a rate pattern – mod plugin – table combination

```
* * * * * mod modplugin_name
{type [lx|qx|mx|}
0   0     0
:   :     :
0   0     0
```

or

b) by a rate – generator plugin combination and a **`table`** expression loading table, which is used by the plugin:

```
* * * * * plugin plugin_name

table table_name
0   0     0
:   :     :
0   0     0
```

where plugin **`plugin_name`** uses data from table **`table_name`**.

## Limitations for tables

These conditions are checked directly after load of table from a .sup file.

All tables are defined by rows of **`year month value`** triples. Year and month jointly define an age or time calculated as **`year*12+month`** [months].

Values are treated as step-wise, acting from the given age till the last month before defined by the next rate line. The values defined for the last line act unlimited age/time onwards.

In all tables ages/times defined by year and month should be increasing.

Tables of values used in conjunction with rate patterns define age-specific values. Table should begin with age 0, i.e. the first line should be

```
0   0     …….
```

Tables of values used in conjunction with rate patterns and treated as life tables containing lx, qx or mx values should conform to the following rules:

**lx tables**

Define proportion of population surviving at certain age.

Values should start with 1 at age 0, i.e. first row be **`0    0      1.0`**

Values should be non-negative, non-increasing with time

**qx tables**

Define proportion of population alive at a certain time point that will die by the beginning of next time point.

Values should belong to [0, 1]

Last values should be 0

**mx tables**

Define annual incidence rates for events.

Values should be non-negative

## General syntax of sup file expressions

`Typewriter font` denotes keywords that should be typed exactly as written. "[|]" means "one of". Braces {} denote parts that can be omitted.

Finalising the current segment data

```
run
```

File inclusion

| | |
|---|---|
| `input_file` | *filename* |
| `output_file` | *filename* |

Global parameters

| | |
|---|---|
| `segments` | *integer* |
| `duration` | *integer* |
| `bint` | *integer* |
| `hetfert` | *number* |
| `sex_ratio` | *real* |
| `timestep` | *integer* |
| `read_xtra` | *[0 \| 1]* |

Event hooking

| | |
|---|---|
| `birth` | *plugin name* |
| `death` | *plugin name* |
| `marriage` | *plugin name* |
| `divorce` | *plugin name* |
| `cohabitation` | *plugin name* |
| `split` | *plugin name* |
| `timestep_once` | *plugin name* |
| `timestep_each` | *plugin name* |
| `newborn` | *plugin name* |

Group inheritance

```
child_inherits_group        [MOTHER | FATHER]
child_inherits_parameters   [MOTHER | FATHER]
```

Collection of statistics (stat plugins)

| | | |
|---|---|---|
| `stat` | *stat plugin name* | *integer (period in months)* |

Custom model parameters

| | | |
|---|---|---|
| `param` | *parameter name* | *real* |

Vital rate definitions

1. Rates defined by a rate table

| | | | | |
|---|---|---|---|---|
| *event* | *group* | *sex* | *marital status* | *parity* |
| [birth\|death\| marriage\| divorce\| cohabitation\| split\|*] | [*number*\|*] | [M\|F\|*] | [s\|m\|v\|w\|*] | [*number*\|*] |
| type | [lx\|qx\|mx] | *time* [age\|tlc\|tlm\| tlv\|tlw] | | |
| *year* | *month* | *rate* | | |
| *integer* | *integer* | *real* | | |
| … | … | … | | |
| *integer* | *integer* | *real* | | |

2. Rates defined by a rate table modified by a plugin

| | | | | | | |
|---|---|---|---|---|---|---|
| *event* | *group* | *sex* | *marital status* | *parity* | mod | *plugin name* |
| [birth\|death\| marriage\| divorce\| cohabitation\| split\|*] | [*number*\|*] | [M\|F\|*] | [s\|m\|v\|w\|*] | [*number*\|*] | | |
| type | [lx\|qx\|mx] | *time* [age\|tlc\|tlm\| tlv\|tlw] | | | | |
| *year* | *month* | *rate* | | | | |
| *integer* | *integer* | *real* | | | | |
| … | … | … | | | | |
| *integer* | *integer* | *real* | | | | |

3. Rates defined by a plugin

| | | | | | | |
|---|---|---|---|---|---|---|
| *event* | *group* | *sex* | *marital status* | *parity* | plugin | *plugin name* |
| [birth\|death\| marriage\| divorce\| cohabitation\| split\|*] | [*number*\|*] | [M\|F\|*] | [s\|m\|v\|w\|*] | [*number*\|*] | | |

Definition of a named data table

| | | |
|---|---|---|
| table | *table name* | |
| type | [lx\|qx\|mx] | *time* [age\|tlc\|tlm\|tlv\|tlw] |
| *integer* | *integer* | *real* |
| … | … | … |
| *integer* | *integer* | *real* |

## Files describing population

Modelled population is described by means of 3 jointly used files having the same name and different extensions

.opop – individuals (population)

.omar – marriages and cohabitation unions

.opox – additional (extra) parameters of the individuals

These files are read in the start of modelling and written at the end. MAXIM can use the files it had generated for input, so experiments may be chained.

SOCSIM and MAXIM files are incompatible. Although the population has the same structure the files have different number of fields (columns) and different coding.

SOCSIM files can be recoded into MAXIM files using the tables described further.

This document is a reworked edition of SOCSIM help file.

### *Files describing individuals (.opop)*

The input and output files, describing the structure of the population (.opop files) in SOCSIM contain information about all individuals of the population, which lived before or are still alive at the beginning of simulation. Files are in ASCII form and contain one line of text per individual. Each line contains 14 numbers separated by spaces.

**Table 1. Format of the SOCSIM input and output population file**

| Position | Meaning | Coding | Comments |
|---|---|---|---|
| 1 | Person id | | Positive integer number. Should be unique among all population members |
| 2 | Sex | 0 – Male<br>1 - Female | |
| 3 | Group | | |
| 4 | Next Scheduled Event | 3 - Death, if already dead | Seems to be useless and is discarded (new events are scheduled after the population is read). The moment of next event is not stored anyway |
| 5 | Date of birth | | In months, starting from some arbitrary starting point |
| 6 | Mother's person-id | | Same type as pos. 1. 0 if from unrelated starting population or a child born outside of marriage or cohabitation (father unknown) |
| 7 | Father's person-id | | Same as pos. 6 |
| 8 | Person id of eldest sibling via mother | | 0 if from unrelated starting population |
| 9 | Person id of eldest sibling via father | | 0 if from unrelated starting population |
| 10 | Person id of | | |

|  |  |  |  |
|---|---|---|---|
|  | last-born child |  |  |
| 11 | Marriage id of last union |  | Positive integer number. Should be unique among all marriages |
| 12 | Marital status | 1 Single<br>2 Divorced<br>3 Widowed<br>4 Married<br>5 Cohabiting | . Partners in broken cohabitations (or in cases where the cohabiting partner dies) revert back to their last marital status before the start of the cohabitation. |
| 13 | Date of death | 0 if alive, otherwise month of death |  |
| 14 | Fertility multiplier | For females, fertility multipliers in model: round(fmult*1000000)<br>For females, otherwise: 0<br>For males:0 | Seems, that in reality they are stored as float numbers, so as plain fmult instead |

Comment: if the “illegitimate” birth rate (for single women) is set to non zero, births can occur to single mothers. In this case the father of the child is set to “unknown” (id=0) and SOCSIM/MAXIM stop to be a “closed simulation system” in a strict sense of a system in which the full kinship may be traced till the starting population. If the birth rate outside marriages is set to zero both parents are always known for the child and SOCSIM/MAXIM behave as closed simulation systems.

To avoid errors when interpreting integer codes of sex and marital statuses and make the files more human readable MAXIM deals internally with one-character representations of sex and marital status (Table 2).

**Table 2. Coding of sexes and marital statuses in SOCSIM and MAXIM**

| **Meaning** | **In SOCSIM file** | **Internally, in SOCSIM** | **In MAXIM I\O .opop files and in the program: string** |
|---|---|---|---|
| Male | 0 | MALE = 0 | “m” |
| Female | 1 | FEMALE = 1 | “f” |
|  |  |  |  |
| Unknown | 0 | MS_NULL = 0 | Not used. Error-triggering value “-“ |
| Single | 1 | SINGLE = 1 | “s” |
| Divorced | 2 | DIVORCED = 2 | “v” |
| Widowed | 3 | WIDOWED = 3 | “w” |
| Married | 4 | MARRIED = 4 | “m” |
| Cohabiting | 5 | COHABITING = 5 | “c” |

Since each member of the population has references to his/her mother and (possibly also) father except for the members of the initial population, references to the **Person id of eldest sibling via mother, Person id of eldest sibling via father,** and **Person id of last-born child** are superfluous. One can always collect offspring of any individual searching through the table (though there may be certain complications when the starting population is a “census” of all living at that moment and children, that died by the moment of the census, will not be included – and the number of individuals and, hence, parity would be incorrect. These issues will be discussed further, when we discuss problems concerning reading the initial population).

Analogously, every marriage record includes **id**’s of both partners, So it is not necessary to store the marriage information in person records.

As was noted before, it is makes little sense to keep the next scheduled event in SOCSIM, and it is absolutely useless in MAXIM due to a different stochastic event model.

All this results in a different, simplified format of the MAXIM input/output population files, that have only 9 parameters, and sex and marital status are coded as symbols.

**Table 3. Format of the MAXIM input and output population file**

| **Position** | **Meaning** | **Coding** | **Comments** |
|---|---|---|---|
| **1** | Person id | | Positive integer number. Should be unique among all population members |
| **2** | Sex | **m** – Male<br>**f** - Female | |
| **3** | Group | | |
| **4** | Date of birth | | In months, starting from some arbitrary starting point |
| **5** | Mother's person-id | | Same type as pos. 1. **0** if from unrelated starting population or a child born outside of marriage or cohabitation (father unknown) |
| **6** | Father's person-id | | |
| **7** | Marital status | **s –** Single<br>**v** – Divorced<br>**w** – Widowed<br>**m** – Married<br>**c** - Cohabiting | . Partners in broken cohabitations (or in cases where the cohabiting partner dies) revert back to their last marital status before the start of the cohabitation. |
| **8** | Date of death | **0** if alive, otherwise month of death | |
| **9** | Fertility multiplier | For females, fertility multipliers in model: **fmult or**: **0**<br>For males: **0** | |

Similar modifications were made to the input/output marriage (.omar) file format [TO BE DESRIBED].

## Modifications to the .opop file

--- here describe problems with reconstructing start/end date and type of population data (censor, initial, mixed)

→ introduce new header to the .opop files

Several problems arise when reading in the starting population and starting a new simulation in SOCSIM. The first is related to the starting month of the simulation. What month shall we start from? The input population contains dates of death for dead – so the starting month should be at least max(date of death)+1. But the survivors might have survived past that date actually till some later date, at which the population was dumped. We need to have information about the last month of the previous simulation, or the month at which the starting population (or census) was dumped. In other words, we need to know the right censoring date. In MAXIM the right censoring date is a command line parameter. Moreover, we might want to store some comments about the starting population in the .opop file – e.g. the date of dump, or census date. To support this MAXIM understands comments in the beginning of the .opop file also (see more about the comments further.

## Population file fields

- 1 Person id
- 2 Sex
  - **0** Male
  - **1** Female
- 3 Group
- 5 Date of birth
- 6 Mother's person-id (**0** if from unrelated starting population)
- 7 Father's person-id (**0** if from unrelated starting population or if unknown)
- 12 Marital status. Partners in broken cohabitations (or in cases where the cohabiting partner dies) revert back to their last marital status before the start of the cohabitation.
  - **1** Single
  - **2** Divorced
  - **3** Widowed
  - **4** Married
  - **5** Cohabiting
- 13 Date of death (0 if alive)
- 14 Fertility multipler
  - For females, fertility multipliers in model: **round(fmult*1000000)**
  - For females, otherwise:**0**
  - For males:**0**

### Population file format

PERSON_ID
SEX
GROUP
NEXT_EVENT
DATE_OF_BIRTH
MOTHER_ID
FATHER_ID
LBORN
MARSTATUS
DATE_OF_DEATH
FMULT

## *Marriage file (.omar)*

A single system applies to marriages and to more general unions.

1 Union id

2 Wife's person-id

3 Husband's person-id

4 Union start date

5 Union end date (**0** = none)

6 Reason (current) union ended--shown by code:

**2** Divorce, marriages only

**3** Death, marriages only

**16** Null--marriage hasn't ended

**4** Break off cohabitation

**5** Cohabitation followed by marriage (which has its own union structure)

**6** Cohabiting partner dies

### Marriage file format

```
MARR_ID

WIFE_ID

HUSBAND_ID

DATE_START

DATE_END

REASON_END
```

## *File defining additional parameters of individuals (.opox)*

Extra file is an ASCII file. Extra file start with zero or more comment lines. A comment line should start from # or % sign. The first non-comment line is treated as table header. Each part of it separated by space or tab sysmols is treated as the column name. These should be the names of the individual parameters.

Then the program expects to read lines of text with the number of real values separated by spaces or tabs. The number of parameter lines should correspond to the number of the individuals in the .opop file. Note: there is no ID fiels in the MAXIM .opox file. The parameters are loaded into the xtra structures of the corresponding individuals.

## Example

Suppose, the .opop file gives information about 5 individuals. Then the corresponding .opox file may look like

```
a1  a2   a3   b1   Siler_par1
.1  .1   1    1    .01
.2  .1   1    1    .017
.1  .1   3    1    .01
0   .1   4    1    .012
.1  .1   5    1    .01
```

## Modifications to the .opox file

New format of the .opox file supported by MAXIM includes header, describing what individual parameters are loaded. Elsewhere in the program the blocks will address these parameters by name. E.g. .opox file beginning with

**alpha beta age_sterile**
**0.1 1 50.0**
**0.11 2 47.0**

Defines 3 parameters with names **alpha**, **beta** and **age_sterile** correspondingly for individuals denoted by 1st and 2nd lines of the .opop file

These parameters are loaded into the **aux_param[]** map of the corresponding individual as [string index – double value] pairs:

**aux_param["alpha"] = 0.1;**
**aux_param["beta"] = 1;**
**aux_param["age_sterile"] = 50.0;**

The functions that need any of individual parameters, can get values from the **aux_param[]** map (using **double get_indiv_param(string name)** function. The function produces an error if some parameter is quiered, but missing.

Value of individual parameter may be set by function **void set_indiv_param(string name, double value)**.

These functions are members of class PERSON

## Notes

Population-wide parameters (defined in .sup file) override individual-level parameters (define in .opox file)

Population-wide parameters are segment-specific since new values of population parameters can be defined for each segment.

The output .opox file contains values for all additional parameters defined in the models – population-wide and individual, resorted in alphabetical order. Since population-wide parameters have the same value for all individuals their value will be same for all individuals written in this file write (read section "Memory management" about the procedure of periodic saving of the population).

Since population-wide parameters have higher priority you can always define a parameter with a name existing in the input .opox file as population-wide to override it.

***Hint***

To replace values of some individual parameter to a new value define a population-wide parameter and run a model with duration 0. The input population will be written to the output population unprocessed and the parameter values will be replaced.

***Advise***

Although you may use long names for parameters it is best to make them no longer than 8 symbols.

## Known problems

The .opop, .opox and .omar files are single for all segments. The header is written for the first segment. So we can not change set (number and names) of population-wide parameters – the .opox file will have different number and order or parameters in different rows (for individuals written during different segments). Although in practice models and pop. parameters may be totally different for different segments…

# Demographical processes in MAXIM

## Main ideas. Rate patterns

In MAXIM things happen to an individual, which may be in (characterised by) some **state**, more specifically, a certain subset of state characteristics. They are:

group: 1, 2, …
sex: M/F
marital status:
parity: 0, 1, …

The **event** which happened to an individual may be:

event: birth (*of a child*), death, marriage, divorce, cohabitation, split, transition (not implemented).

So should define the rates, or probabilities (or the rules to evaluate them) for **all possible events** for individuals in **all possible states** which may be encountered during the modeling. This is done in .sup file(s).

We have to define rates for many possible combinations of event-state, and this is simplified by the using of **wildcards**. A star symbol (*) used in a position for and rate of state characteristics represents any value of it.

## Rate patterns lookup. Specific-to-general priority rule

We may like to define distinct rates for certain cases and one rate for all the others. This can be easily done if we know how the rates are looked-up and used.

MAXIM implements an event competition model. This means that on each time step (1 month) each individual is tested across all possible events that may happen to him/her. E.g. a single female may bet married, start cohabitation, give birth to a child, migrate or die. Thus the program needs to know probabilities of certain events happening to a person being in a certain state. It searches among the set of rates using the **specific-to-general priority rule**:

for each state (e.g. parity=2) rate(s) defined specifically for this state (parity=2) get priority over rate(s) defined for a general case (parity=*).

**Example:** if we define rates as following:

```
birth * * * 0
{rate definition A}
birth * * * *
{rate definition B}
birth * * * 3
{rate definition C}
```

rate definition A will be used for parity 0, C – for parity 3 and B for all the others. The order in which these definitions go in .sup file is not important.

**Ways to define a rate**

MAXIM provides flexibility in defining the vital rates. They can be defined as age-specific or duration-specific using tables (time-rate), tables modified by "modifier plugins" (mod plugin) or by functions (plugins). The following patterns may be:

#### 1. Age- or period- specific rates defined by a rate table

```
{event} {group} {sex} {m_status} {parity}
type {lx|qx|mx| {age|tlc|tlm|tlv|tlw}
xxx  xxx   xxx
….
xxx  xxx   xxx
```

The first line is the event-state combination previously desribed. The secong line tells which type of rate is it (lx, qx or mx rates) and are the age- or duration-specific.

The difference between age- or period-specific rates is basically how we define the "clock", the time. For age-specific rates time is age, for duration-specific it is duration – usually time since last event of a certain kind.

The 3-letter combinations define the duration:

| Value | Duration | Type of rate |
|---|---|---|
| age | age | age-specific |
| tlc | time since last child | duration-specific |
| tlm | time since last marriage | duration-specific |
| tlv | time since last divorce | duration-specific |
| tlw | time since last widowhood | duration-specific |

The 3rd and later lines define a rate table of the form

year – month – value

**2. Age- or period- specific rates defined by a rate table modified by a modifier plugin**

```
{event} {group} {sex} {m_status} {parity} mod {plugin name}
type {lx|qx|mx| {age|tlc|tlm|tlv|tlw}
xxx  xxx   xxx
….
xxx  xxx   xxx
```

Same as previous, but the rate is requested from a "modifier plugin" which gets the table for it's input. The user should program the plugin and recompile the program (see the "Programming plug-ins" section)

**3. Age- or period- specific rates defined by a rate table modified by a plugin**

```
{event} {group} {sex} {m_status} {parity} plugin {plugin name}
type {lx|qx|mx| {age|tlc|tlm|tlv|tlw}
```

The plugin is used to return the rate. No table is loaded into memory. The plugin is a generator plugin programmed by the user. It's values are treated as lx, qx or mx rates with a certain time (`age|tlc|tlm|tlv|tlw`) given to the plugin.

The `type {lx|qx|mx| {age|tlc|tlm|tlv|tlw}` specification should always be present.

## Defining complex vital rates without programming

Example 1. Define birth rate which is a combination of age-specific for the first child and period-specific for the subsequent children.

To do this write in .sup file something like this:

```
birth * * * 0
type qx age
{table 1 of age-specific rates}
birth * * * *
type qx tlc
{table 2 of duration-specific rates}
```

In this case the age-specific rate (defined by table 1) will be used for the risk of first birth (parity 0) and period-specific (table 2) for the risk of next births.

You can define rates differently depending on marital status.

Example 1A: same as Ex. 1 but rates different for married, cohabiting and unmarried

```
birth * * m 0
type qx age
{table 1_married of age-specific rates}
```

```
birth * * c 0
type qx age
{table 1_cohabiting of age-specific rates}
birth * * * 0
type qx age
{table 1_default of age-specific rates}
birth * * m *
type qx tlc
{table 2_married of duration-specific rates}
birth * * c *
type qx tlc
{table 2_cohabiting of duration-specific rates}
birth * * * *
type qx tlc
{table 2_default of duration-specific rates}
```

## Comments

Certain combinations of rates, states and clock make sence, others don't. Here is a table of what you can do

| Event | Clock | Type of rate |
|---|---|---|
| birth (Child birth) | age | age-specific birth rate |
| | tlc (time since last child) | duration-specific birth rate |
| Marriage | age | age-specific marriage rate |
| | tlv (time since last divorce) | duration-specific marriage rate |
| | tlw (time since last widowed) | duration-specific marriage rate |
| Divorce | tlm (time since last marriage) | duration-specific divorce rate |

Note that not all clocks are always defined:

| Clock | When defined |
|---|---|
| age | always |
| tlc | for parities > 0 |
| tlm | for married |
| tlv | for divorced |
| tlw | for widowed |

If the clock can not be defined in order to evaluate the rate the program will stop and complain. Change the way you define the rates. Example:

The following definition:

```
birth * * * *
type qx tlc
…
```

will produce an error when testing for event "birth" for a childless woman since tlc is not defined for her. The correct way to define the births rate in this case would be:

```
birth * * * 0
type qx age
…..
birth * * * *
type qx tlc
…..
```

In this case the age-specific rate will be used for the child-less (parity=0) women.

To run a model rates for all possible transitions should be defined. At least the following rates should be defined:

During simulation other possible transitions may be encountered and MAXIM will require other rates. SOCSIM creates the complete set of all possible rates by creating missing rates from the defined one according to the following rules:

**Figure … Rules for creation of missing rate tables in SOCSIM**

**1.** Ensure that the essential rates are defined:

| **Process rate** | **Sex** | **Group** | **Mar. Status** | **Parity** |
|---|---|---|---|---|
| birth | female | 1 | single | 0 |
| birth | female | 1 | married | 0 |
| death | female and male | 1 | single | - |
| marriage | female and male | 1 | single | - |
| divorce | female and male | 1 | married | - |

**2.** Create tables for all marital statuses, group 1, parity 0 by copying rate tables for other marital statuses as follows:

**"s"->"v"->"w"->"m"->"c"**

E.g. if only table for state "s" (single) is present, all others will be created from it. If "w" is present, "m" and "c" are made from it.

**3.** For all marital statuses other than "**s**", group 1, parity 1 copy data from previous parity.

**4** For groups numbers greater than, all marital statuses, copy data from previous group

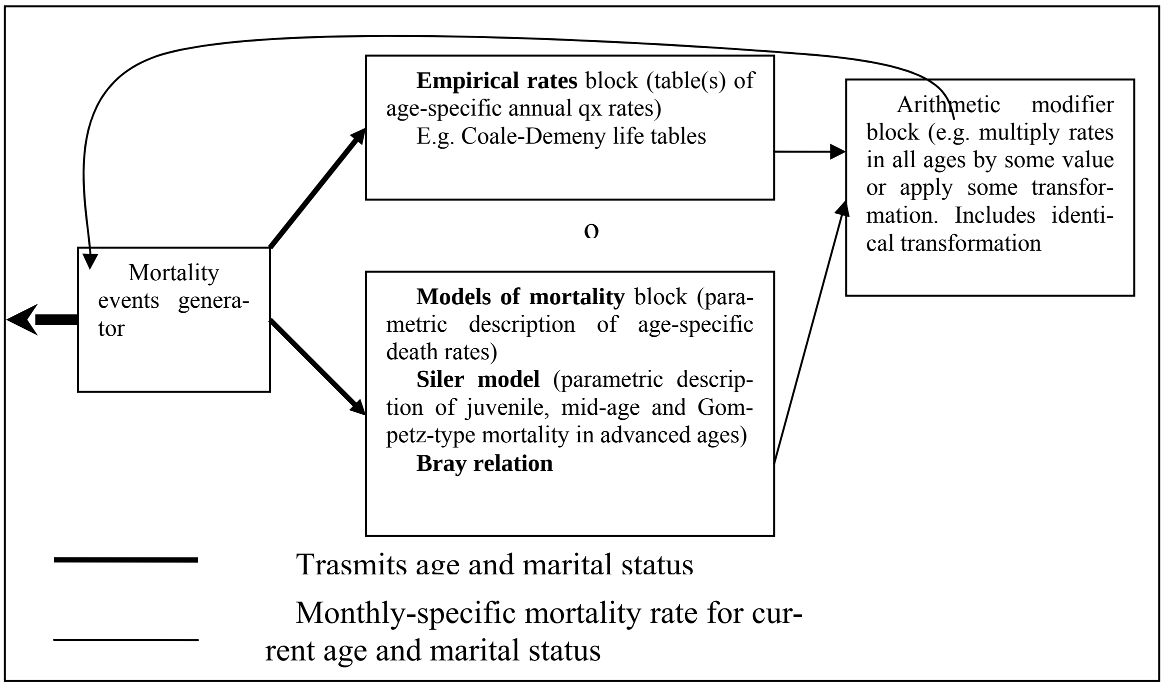
**5.** Similarly for the non-birth rates start with group 1 and default back by marital status.

The number of parities is limited by a compile-time constant (10, hence parities are possible 0..9). Rate tables for all parities were created.

The number of groups is limited by a constant (16). Rate tables for all parities are created.

MAXIM does not create "all possible rates" since their set may be unlimited. It requires that all needed rates be defined but allows to define multiple rates in one expression. This method is called "rate patters" and is explained in detail in the "Rate patterns" section of this manual.

## Mortality



Empirical rates block (table(s) of age-specific annual qx rates)
E.g. Coale-Demeny life tables
Arithmetic modifier block (e.g. multiply rates in all ages by some value or apply some transformation. Includes identical transformation
o
Mortality events generator
Models of mortality block (parametric description of age-specific death rates)
Siler model (parametric description of juvenile, mid-age and Gompetz-type mortality in advanced ages)
Bray relation
Trasmits age and marital status
Monthly-specific mortality rate for current age and marital status

**Fig. 1. Flow chart of generating rates describing death (mortality rates) in MAXIM**

MAXIM comes with a Siler parametric model of mortality implemented as a plugin. It is described in the section “Extending MAXIM. Programming plug-ins”.

You can implement your own parametric models of mortality. Consult the above mentioned section and study the plugin implementing the Siler model to get a feeling of how a MAXIM plugin works.

### Rates given by rate tables

If no mortality_rate function is specified (invoked from the .sup file), the mortality rates are taken from the tables. In this case the common requirements and rules common for SOCSIM and MAXIM act (Table …)

### Rates specified by functions

If a mortality_rate function is specified, it should be able to handle any combination of person parameters, or any **age, sex, marstatus, group, parity**.

## Marriage

*Awaiting for description*

Models, describing distribution of time of entry into marriage:

Coale-McNeill, Hermes (Henry?) (not implemented)

## Fertility

Rates can be given as **age-specific** and **parity-specific** yearly probabilities of birth for females for different marital status and parity. This is all coded by the rate table header.

The user can also supply a birth interval using parameter with interval in months:

**bint 5**

In this case the randomly generated birth events within **bint** from the previous birth, are dropped, and the internal (crude) rate is "inflated" to achieve the same output, net rate.

*Awaiting for description*

Other models of fertility: (Not implemented)
PPRs
Sterility model (Pittenger)
Partial fertility models (Bath, Zaba)
Birth intervals

Birth-spacing models – Bongaarts, Hilary Poge, Pollerd, Brant Keeling

Remarrige/Re-cohabitition delay after widowhood/Divorce.

## *Rate look-up and conversion*

Here we describe how the rates are looked-up in the rate tables and converted from the supported formats (lx, qx, mx) to the internal format (annual qx).

The vital rate tables have a general form
type [lx|qx|mx] *duration*
y1 m1 r1
y2 m2 r2
.........

Every row is year, month and value. Rewrite time as t=12y+m and consider 2 rows
t1 r1
t2 r2
Suppose we look up for a value at time t. t1<=t<t2 (else we look up in other rows)
The internal format used for the event-competition model in MAXIM is annual qx rates, or risk of event in a year time (denote it qx_12). There can be 3 cases:

### Input rate given as qx

Rate qx corresponds to the interval t1..t2. To convert it to annual probability we assume that the risk of this event per unit time is constant. In this case the risk per year is calculated as follows. Suppose the period length for which the input rate is defined is T=t2-t1. Then
px_12 = px_T^(12/T) where px = 1-qx. So
**qx_12 = 1-(1-qx_T)^(12/T)**.

### Input rate given as lx

Lx is survival over the period. S(x1->x2)=exp(-\int(_x1^x2  h(q) dq)
where h(q) is the hazard rate over the interval x1..x2. Assume that it is constant over the interval. Then we have
lx_T = exp(-hT)
px_12 = exp(-12h)
So log px_12 = log lx_T*(12/T) and
**qx_12 = 1-exp(log lx_T * 12/T) = 1-lx_T^(12/T)**

## Input rate given as mx

***To be implemented***

# Customizing MAXIM

## Maximum age in model

The maximum age that an individual may attain is limited in MAXIM. When a person reaches this age a “death” event is generated automatically. Individuals will die at this age even if the mortality rate is set to zero.

The maximum age **in years** is defined by the constant `MAXYEARS` in file `common_declarations.h`. Its default value is 100. Change it if your model requires a different maximum age. Be sure to rebuild the program for the change to take effect.

## Generating a start population

MAXIM may generate a simple test starting population of given size and sex rate. Call the program with parameters….

# NOT IMPLEMENTED

## Example set-ups for different models

### Modelling on a different time scale

The "base" (minimum) modelling time step in the program is expressed as "1 month". Rate tables and rate generators are expected to return "annual" rates of events. But the meaning of this "month" is abstract; this can be any time step. The base time step can be arbitrary and the rates should be output per 12 of such time steps.

E.g. suppose rate tables and rate plug-ins assume and return daily rates of events. Then the "base" modelling time step will be 2 hours. In all input and output words "months" should be treated as these 2-hour periods.

### One sex population with births and deaths

- create a starting population file **`population.opop`** where all individuals are females (column 2: "f"), single (column 7: "s") and have non-zero fertility multipliers (column 9)
- create files describing age-specific mortality and fertility rates **`mort.txt`** and **`fert.txt`**
- write a .sup file including the following important lines:

```
  input_file population
  output_file population_final            # any name
  duration 120                      # [months]. Any duration

  sex_ratio 0.0         # prop. of males → all newborns are females
  timestep 1                  # time step: 1 month
  read_xtra 0           # no additional individual parameters read

  * * * *               # any, i.e. default rate
  0   0   0             # is zero


  death 1 F *                 # mortality in group 1, females, any
status
  include mort.txt      # described by a rate table from this file

  birth 1 F single *
  include fert.txt

  run                   # end of information for this segment
```

This file denotes a simulation running 120 months, progressing by time step equal 1 month, starting with population population. Zero rates are defined for all events except birth to single females and death of single females.

### Marriage success depending on education

- take or create a starting population file **`population.opop`**
- create a file of extra parameters **`population.opox`** including parameter `education`. Set it to value 0 for individuals of the starting population.
- create a plug-in called `plugin_education` in `plugins.cpp` file (how to do it was described earlier). This plugin will be called every time step for each individual. The

plugin should model how the education of the individual changes over time. The current level of education should be stored as value of the `education` parameter.

- create a plug-in called plugin_marriage_education. It will be called whenever it is tested whether the current individual should start marital search. It should return the annual rate of marriage depending on the value of `education` parameter and basic parameters of the individual.
- recompile the program
- write a `.sup` file including the following important lines:

```
  input_file population
  output_file population_out  # any name
  duration 1200               # any duration

  sex_ratio 0.5112            #  typical  ratio  proportion  of  newborn
males
  timestep 12           # any time step
  read_xtra 1           # read additional individual parameters

  monthly plugin_education
  …definitions of vital rates except marriage # define
                        # rates for other events

  marriage * * * plugin_marriage_education

  run
```

This file denotes a simulation running 100 years, progressing by time step equal 12 months, starting with population `population`. Zero rates are defined for all events except birth to single females and death of single females.

## Using wildcards to define vital rates

To define constant rate for all ages In the .sup file write string like

```
  death * * *
  0 0 0.005
```

This defines mortality rate for all groups, sexes and parities as 0.005/year starting with age 0 years o months.

Similarly

```
  * * * * *
  0 0 0.0
```

Defines zero rates for all processes.

*Note 1: the rate definition should always start with age 0 years 0 months.*

Specific definitions override “wildcard” ones.

```
  birth * * * *
  0 0 0.0

  birth * * * 1
  0 0 0.0
  15 0 0.8
  50 0 0

  birth * * * 2
  0 0 0.0
  15 0 0.5
  50 0 0
```

This defines birth rate 0.8/year for parity 1, 0.5/year for parity 2 both acting from age 15 to 50 years, rates for other parities and ages would be zero.

Rate generation plugins and rate tables may be mixed. For example we could write

```
birth * * * *
0 0 0.0

birth * * * 1  plugin_birth_rate_parity_1

birth * * * 2  plugin_birth_rate_parity_2
```

Here birth rate for parities 1 and 2 are defined by 2 separate plugins. The default rate acting for other parities is zero.

***Note: you can NOT use both plugin AND rate table definitions for the same rate pattern, i.e. write like this:***

```
birth * * * 1  plugin_birth_rate_parity_1
0 0 0.0
15 0 0.8
50 0 0
```

## Running speed and optimization

Depends on many factors such as size of the population, how the transition rates are calculated at each time step, how many user-plugins have been hook to the program.

Some operations are repeated many times in cycle and so are most critical for the speed. Such operations are operations performed every modelling time step for every individual – e.g. testing for events. Some events are possible (and, hence, tested for) for only a part of the population (e.g. divorce or birth), some are tested for everyone (e.g. death). So e.g. if a mortality rate is calculated by a plugin, it's processing speed will have a big influence.

Without plugins processed every time step for every individual and vital rates given by tables the processing speed is 2-6*T mln person-years/h on a 4 GHz Intel PC under MS Windows (given modelling time step equal T months). E.g. simulation of a 100 thousand population during 1 hour allows for 240-720 time steps. If 1 time step equals 1 year this will give us 240-720 years of "model" time.

To increase the modelling speed consider the following important points.

- the program and all data should fit into operation memory without swapping. Swapping may decrease the processing speed 100 times or more. Swapping begins when the memory size requested by the program exceeds total amount of installed physical memory minus minimum amount necessary for OS and other programs. In MAXIM most memory is consumed by the population map which stores all individuals – alive and dead and so grows all the time throughout the simulation. Rate of increase depends on rate of birth of new individuals and greatly – on the number of additional parameters of the individuals. To avoid swapping close all unnecessary programs and avoid unnecessary individual parameters.

  Example:

  in a model where individuals have additional parameters each individual record consumes 500-800 bytes of memory. On a computer with 512 mb of physical memory about 490 mb of it may be allocated without swapping, which allows to create a population map of about 600 thousand records. This may be a 10 thousand human population modeled for 60*generation_time ~ 1500 years.
- optimize most critical plugins (such as generators of rates of death and other processes experienced by many members of the population)
- disable unnecessary plugins
- when possible invoke plugins with a bigger time period.

  Example:

  You may design plugins that collect some information about the population periodically (e.g. size, age structure, total wealth) and store it in population-wide parameters which are then used by other plugins to calculate vital rates. Collecting information about the whole population is costly, so you may consider doing it every n-th model time step.

  **(YET NOT INPLEMENTED TO INVOKE managing PLUGINS LESS OFTEN THAN every time step. But note stat plugins!!!!)**

## Memory requirements and management in MAXIM

MAXIM stores information about individuals, their unions and parameters in operative memory and actively uses it during simulation. When amount of memory required for MAXIM increases above the amount of operative memory that the operation can offer without swapping (this amount equals roughly amount of installed RAM minus 50 mb for MS Windows) swapping begins and the processing speed may drop thousand times or so. Swapping can be avoided by installing more RAM and economical memory management.

Most of memory required for MAXIM is used to store information about individuals (alive and dead) (including variable number of additional parameters) and information about the marriage unions (in program these are maps **`popmap`** and **`marmap`**).

The number of individuals and unions generated depends on rates of fertility, marriage and cohabitation, these are model-dependant.

The complete historic record of all individuals from a long term or big-population simulation may contain millions of records. All this information is need to track the kin structure and evolution of the population, but it is impractical to keep it all in RAM.

To use the operation memory more economically MAXIM stores in RAM information only about individuals that were born and unions that started more recent than 2*maximum life span (2*MAXYEARS years). Information about more ancient individuals and unions is periodically written to output .opop, .opox and .omar files and these individuals and unions are erased from the maps.

Time lag 2*MAXYEARS years ensures that parents and most probably, grand parents of living individuals are kept in the working map at each time. Still more ancient ancestors may be already absent.

**Warning:**

*when writing plugins that access information about individuals that may be diseased and unions that started long time ago check their presence in the popmap and marmap (by unique IDs)!*

Clearing of the maps occurs periodically – each 100 model time steps, so it should not impair the modelling speed too much.

Amount of memory required for MAXIM may be estimated in the following way.

Suppose we have a population with constant birth rate (amount of births per unit time) b [indiv/year] (this may be a stationary population with a constant birth rate) and a constant marriage/cohabitation rate m [union/year]. Suppose at some point we have a stationary population of size N with number of marriage/cohabitation unions M. Suppose a model time step is T years. Let size_p define size of individual record and size_m – union record in bytes. Then RAM will contain about **`N + N*b*2*MAXYEARS`** individual and **`M + M*b*2*MAXYEARS`** union records *just after* the clearing of maps and **`N + N*b*(2*MAXYEARS + 100*T)`** and **`M + M*m*(2*MAXYEARS + 100*T)`** after 100 timesteps, *just before* the next clearing, correspondingly. The memory requirement of MAXIM would be changing over time, having a saw profile with peaks

**`size_p N(1+b(2MAXYEARS+100T))+`**
**`size_m M(1+m(2MAXYEARS+100T))`** bytes approx.

The size of one individual record (without additional parameters) is about 300 bytes, and 200 bytes for a union.

**Real life example**

A stationary human population of about 5000 individuals with typical rates is modelled. Individuals have 25 additional parameters, no unions are generated. Model time step is 1 year. In this set-up the long term peak memory requirement of MAXIM is about 460 mb.

## Logging, debugging and troubleshooting

Program writes much information to the screen and creates a file with program log (`basename.log`) and a file with population pyramid and base information about the population measured in the beginning and end of each modelling segment (`beasename.pyr`). Here `basename` is the name of your `.sup` file. These files are created in the same directory where the model .sup file is located.

The command line parameter `/v` forces the program to produce more verbatim output.

Error, notification and debug messages of MAXIM contain name of the function in which the message was raised. This helps to track the problem.

Many functions have an `int debug` parameter which controls output of debugging information by that function. If you suspect a problem in certain function(s) set this parameter to 1 (in the code) and rebuild the program.

In case of problems it may be helpful to run the program with /v switch and reroute the program printout to a file. This can be done with a `>` command, e.g.

```
maxim evolution 1 1201 2400 /v > output.txt
```

Consult sections “Programming: functions” and “call structure” of the manual.

A very detailed report about events that happen to the population can be obtained by setting debug=1 in process_time_step() function in population.cpp.

### *Frequent errors*

Errors in MAXIM may occur from wrong input data (rate tables and population data), wrong .sup files, faulty user plugins or errors of the program itself. Here we describe typical errors and effective ways to trace and fix them.

#### get_xtra_param()

**Screen error text:**

```
Error: get_xtra_param(): could not find parameter "xxx" for person id=xxx neither in current segment, not in person xtra map!
xxx segment parameters:
    xxx = xxx
      ……
    xxx = xxx
xxx personal parameters:
    xxx = xxx
      ……
    xxx = xxx
Hint: get_xtra_param(): do you have inheritance of individual parameters (hooked to event birth)?
```

**Reason:**

function get_xtra_param() was called from a plugin to get a value of individual or population-wide parameter but did not find it.

**Solution:**

The list of existing parameters is given in the error message. Inspect it. Does it have the necessary parameter?

The error may occur on invocation of generator/modifier or stat plugin. They are called by plugin() and stat_plugin() functions respectively.

1. Find which plugin generates the error. For that set parameter debug to 1 in functions plugin() and stat_plugin in plugins.cpp and rebuild the program.
2. After having ran the program you will see as line

```
Debug: plugin(): called plugin 'plugin_xxxxxxxxxxxx'
```

or

```
Debug: stat_plugin(): called stat plugin 'plugin_xxxxxxxxxxxx'
```

just before the error message.

3. Find this plugin in the `plugins.cpp` file. Check whether this plugin is invoked at the right time and requests the right parameters. Are this parameters properly initialized at birth of a new organism?

`Symptom: unexpected operations are performed upon individuals`

**Debug:** find out what plugins are invoked. To see this turn on the invoke messages in the plugin "switchboards" (calling routines). To do this set internal constant **debug=1** in **plugin()**, **stat_plugin()** and **mod_plugin()** in **plugins.cpp**. Recompile.

Another useful measure is to check what is actually happening to the individuals: what events happen and what happens to their state. In order to do this go to **process_time_step()** in **population.cpp** and set **debug=1**. Recompile. This will print the diagnostics of the form:

```
  Debug: process_time_step: performing event b on individual (state
before process.): id862 sexf grp1 bd1406 dd0 moth116 fath0 msts
prmst- fmult1 marity0 parity1,chil{1521 }
  Debug: process_time_step: processing possible plugins for id=862
hooked to event b
  State after event processing:id862 sexf grp1 bd1406 dd0 moth116
fath0 msts prmst- fmult1 marity0 parity2,chil{1521 1598 }
```

This gives the event type (b), state before and the event (id, sex, grp=group, bd=birth date, dd=death date, moth, fath = parent ids, mst=marital status (s=single), prmst = previous marital status, chil=children ids).

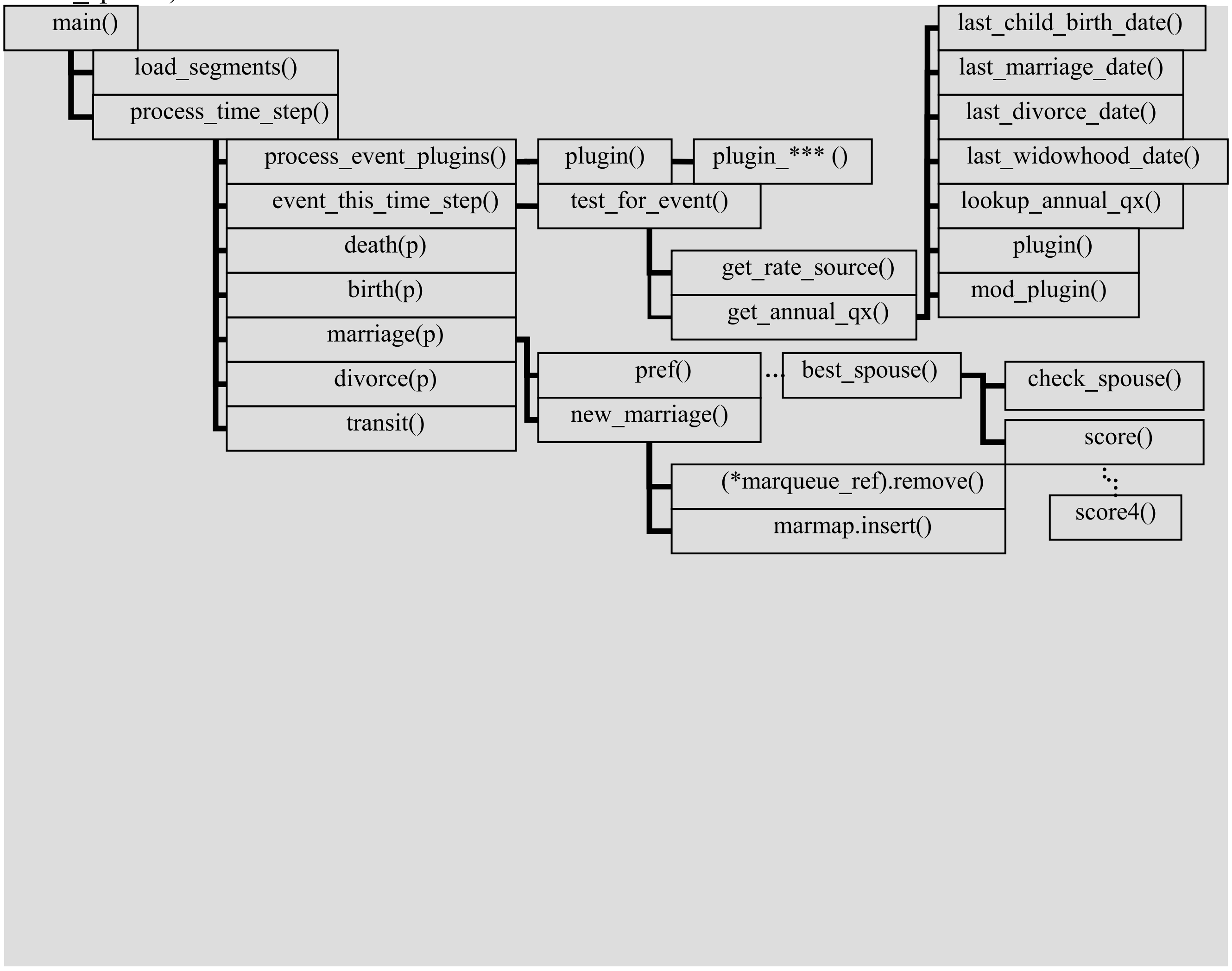
Note that many functions have a debug constant defined inside them. Normally it is set to 0. It you set it to 1 the function will prints some relevant diagnostics.

## Programmer guide

### *Hierarchy of function calls*

Note: functions may be located in different cpp files. Use search to locate them. Certain functions may be called conditionally, not always. Some functions may be called in another order (e.g. several times within one function, with different parameters. Here are displayed the links, not the order. Each function is shown only once.

Dashed lines – function calls, dotted lines – functional links (e.g. pref = &best_spouse)



To be continued…

## Modules and functions

This section tells about MAXIM functions and their meaning. Functions are collected by modules in which they are implemented.

### Main.cpp

| | |
|---|---|
| main() | |

### Segment.cpp

| | |
|---|---|
| int SEGMENT:: compose_table_name(string& table_name, string rate_type, string event, int group, string sex, string marstatus, int parity) | |
| int SEGMENT:: get_rate(double& rate, string rate_type, string event, int group, string sex, string marstatus, int parity, int time, int age) | |
| double SEGMENT:: table_lookup(int age, vector<RATE_TABLE_ROW> table) | |
| int SEGMENT:: rate_set(int group, string event, string sex, string mstatus, vector<RATE_TABLE_ROW>& table) | |
| int SEGMENT:: birth_rate_set(int group, string mstatus, int parity, vector<RATE_TABLE_ROW>& table) | |
| int SEGMENT:: cohab_probs(int group, string sex, vector<RATE_TABLE_ROW>& table) | |
| void SEGMENT:: print_rate_tables(ostream& f) | |
| void SEGMENT:: print_rate_table(ostream& f, vector<RATE_TABLE_ROW>& rate_table) | |
| int SEGMENT:: set_param(string parname, string parvalue) | |
| void SEGMENT:: print_params(ostream& f) | |
| void SEGMENT::report(POPULATION *ptr_pop, ofstream& fd) | |
| int SEGMENT:: fill_missing_LC_rate_tables() | |
| int SEGMENT:: fill_missing_rate_tables(POPULATION* ptr_pop) | |
| void SEGMENT::initialize_segment_vars() | |

### marriage.cpp

| | |
|---|---|
| void MARRIAGE::write(ofstream& fd) | |

### random.cpp

| | |
|---|---|
| int irandom() | |
| int irandom_ab(int a, int b) | |
| double rrandom() | |
| double normal() | |
| double fertmult() | |

### Population.cpp

| | |
|---|---|
| void POPULATION::birth(PERSON *p, double prop_males) | |
| void POPULATION:: death(PERSON *p) | |
| void POPULATION:: marriage(PERSON *p) | |
| void POPULATION::new_marriage(PERSON *p, PERSON *spouse) | |
| void POPULATION:: divorce(PERSON *p) | |
| int POPULATION::check_spouse(PERSON *p1, PERSON *p2) | |
| void POPULATION:: transit(PERSON *p, int hhmigration, int destgroup) | |
| void POPULATION::add_minor_children(PERSON *p, list<int> transit_list) | |
| void POPULATION::assemble_household(PERSON *p, list<int> transit_list, int hhmigration) | |
| int POPULATION::time_on_marqueue(string sex) | |
| void POPULATION::process_month() | |
| int POPULATION::count_current_population(string sex, int group) | |
| void POPULATION:: event_this_month(PERSON* p, string& event, int& new_group) | |
| void POPULATION:: pyramid(int current_month, ofstream& fd_pyr) | |
| MARRIAGE* POPULATION::plast_marriage(PERSON *p) | |

| | |
|---|---|
| PERSON* POPULATION::pspouse(PERSON *p) | |
| PERSON* POPULATION::pgrand_p(PERSON *p, string kincode) | |
| int POPULATION::get_expected_number_of_births(int group) | |
| int POPULATION::get_expected_number_of_transits(int from, int dest) | |
| void POPULATION::census(int current_month) | |
| void POPULATION::child_census(int current_month) | |
| int POPULATION:: read_permap_MAXIM(ifstream& f, int input_left_t, int input_right_c){ | |
| int POPULATION:: read_permap_SOCSIM(ifstream& f, int input_left_t, int input_right_c){ | |
| void POPULATION:: write_population_MAXIM(ofstream& f){ | |
| int POPULATION::read_marmap(ifstream& f){ | |
| void POPULATION::read_xtra(ifstream& fd) | |
| void POPULATION:: write_xtra(ofstream& fd) | |
| void POPULATION::cross_link_input(){ | |
| void POPULATION:: report(ofstream& f, string text) | |
| int POPULATION:: generate_initial_population(string type, int pop_size, double prop_male, string fname) | |
| void POPULATION:: marriage_tally(int current_month, ofstream& fd) | |

## Person.cpp

| | |
|---|---|
| void PERSON:: print_person_MAXIM(ostream& f) | |
| void PERSON::write_person_MAXIM(ofstream& fd) | |

## *Call structure*

| | |
|---|---|
| main() → | |
| ├load_segments()→ | Loading parameters and rates from the .sup file<br>Loads parameters of modeling – scalar parameters, rate tables, action hooks – segment- and population-wide parameters |
| ├preparse_sup_file() | Unwinds the recursion of included .sup files, puts all lines into an array |
| ├curr_seg.set_param() | If found a par_name par_value pair and par_name is not a hook name, try to set a segment-wide variable with that name |
| -- prints data defined for segments -- | |
| ├read_permap_MAXIM() | Loads population |
| ├read_marmap_MAXIM() | Loads marriage info |
| ├cross_link_input() | Builds links between individuals and marriages in the population |
| ├report() | Report on the current population |
| ├pyramid() | Pop. pyramid |
| └Main simulation loop: loop over segments; monthly loop: | model operations over the population performed every period of time defined by timestep (external) / modelling_time_step (internal) variable |
| ├process_time_step() | |
| ├event_this_time_step() | evaluates an event that will happen to the individual during current time step |
| ├test_for_event() | test if event of a certain kind happens |

| | |
|---|---|
| | during the current time step to a certain individual |
| └get_annual_rate() | Annual incidence rate for specified event |
| ├death()<br>├birth()<br>├marriage()<br>├divorce()<br>├transit() | Perform scheduled event actions upon individuals (in random order) |

## *MAXIM Classes*

All MAXIM classes are defined in MAXIM.h. Implementations are located in .cpp files with the names of classes.

### class TABLE_ROW

Describes a row of a table having 2 coulmns – integer and double.

### class TABLE

Describes tables as vectors TABLE_ROW rows, additional parameters of tables (name, type). This class stores life tables and 2-row data user-defined tables with arbitrary data. TABLE supports several types of data corresponding to table of mx, lx, qx or free “” type. Type of the table is stored as a public parameter **name**. The type defines limitations on the data and verification procedures that the function **check** performs.

TABLE supports two methods of table look-up.

**Public parameters**

| | |
|---|---|
| vector <TABLE_ROW> table | Table itself |
| vector <TABLE_ROW>:: iterator row_iter | Iterator |
| string name | Name of the table used to address it from the lookup plugins |
| string type | Type of the table. Influences checking and conversion. Possible values are "", "mx", "lx" or "qx" |

**Member functions**

| | |
|---|---|
| double lookup(int month); | General look-up of stepwise function value defined by table without value conversions |
| double lookup_mx(int month) | Looks-up the mx value corresponding to month. Conversion from the original format of table (lx or qx values for life tables) is performed based on table type |
| void clear() | Clears table |
| int check() | Checks consistency of a table according to its defined type |
| void print(ostream& f) | Returns the data dump of a table by the f reference |

### class SUP_OBJECT

Preparsing of .sup files and commands to work with its lines by tokens. On instantiating of the class .sup file is loaded, nesting files merged, comments and empty lines stripped. Member functions allow to get the contents of lines by tokens – parts separated by spaces or tabs.

**Member functions:**

int preparse_sup_file(vector <string>& command_file_lines, string dir, string file_name); - does the preparsing

string current_line();

int num_tokens(); - number of tokens in the current line

string token(int n); - token number n (n=1,2…)

int find_token(string text); - Find token with text (case sensitive)

void next_line(); - Move to next line

int file_ended(); - TRUE if passed the end of .sup file

### class STAT_PLUGIN

Instance contains information about a plugin calculating and dumping some stat info. On init opens an output file, on destroy - closes it.

### class RATE_SOURCE

Contains information how to evaluate a rate - from life table, a plugin, possibly modified by a mod plugin. This is described by 3 string parameters: table ID, plugin ID, modifier plugin ID. Rate patterns refer to instances of this class

**Member functions:**

string print() – prints values of parameters.

void clear() – clears data

### class PERSON

Contains all information about an individual of the population.

**Member functions:**

void write_person_MAXIM(ofstream& f) –person info as a line of .opop file

void print(ostream& f); - person info for screen print

double get_xtra_param(POPULATION* ptr_pop, string parname); - get additional individual/population parameter by name

int set_xtra_param(POPULATION* ptr_pop, string parname, double value); - set value of parameter

void update_maxid() – function used to update auto ID counter of the class

### class MARRIAGE

All information about a marriage or cohabitation union.

**Member functions:**

void write(ofstream& f);

void print(ostream& f);

void update_maxid()

See class PERSON for descriptions.

## class POPULATION

Root class. Contains all information about the simulation, population, marriages and data by model segments. Only one instance is created in the beginning of the simulation.

**Public variables and structures**

```
map <int, PERSON> permap; // map, stores all persons
map <int, PERSON>:: iterator piter;

map <int, MARRIAGE> marmap; // map, stores all marriages
map <int, MARRIAGE>:: iterator miter;
list<SEGMENT> segment_list;
list<SEGMENT>::iterator segiter;
int current_segment, num_segments; // id of current and number of segments
SEGMENT *ptr_seg; // pointer to the current segment

list<PERSON*> marqueue_m; // marriage queue, males
list<PERSON*> marqueue_f; // marriage queue, females
list<PERSON*> *marqueue_ref; // reference to a marriage list
list<PERSON*>:: iterator marqiter;

vector<string> marstatuses; // vector of 1-letter abbreviations of all
// marital statuses. For iterating through list
vector<string> sexes; // same for sexes
vector<string> events; // same for sexes

map <int, GROUP> groupmap; // map, stores data for all groups
map <int, GROUP>:: iterator giter;

// if POPULATION is input it has a Left-Truncation date
int input_left_trunc, input_right_censor, numgroups, firstyear, current_month;
static size_t living;

// filenames
string title, input_file, output_file;
string base_dir, rate_fname, log_fname, rate_fname_root;

// string pop_in_fname, mar_in_fname, xtra_in_fname; // input files
string pop_out_fname, mar_out_fname, xtra_out_fname; // output files
string pyr_out_fname, stat_out_fname, prefix_out_fname;
string rnd_fname;

string pop_out_fname_seg, mar_out_fname_seg, xtra_out_fname_seg;
string pyr_file_fname_seg, stat_file_fname_seg;

// files
ifstream fd_pop, fd_mar, fd_xtra;
ofstream fd_out_pop_seg, fd_out_mar_seg, fd_out_xtra_seg;
//ofstream fd_out_pop, fd_out_mar, fd_out_xtra;
ofstream fd_rn, fd_stat;
```

```
//int a_tally[NUMSEXES][A_NUMCAT];
map<string, map<int, int> > a_tally;
//int c_tally[NUMSEXES][C_NUMCAT];
map<string, map<int, int> > c_tally;



// vector<int> size_of_pop; // by groups
int write_output_each_seg;
int read_xtra_file;

//int time_waiting[NUMSEXES];     /*person months on mqueue*/
map<string, int> time_waiting;
```

**Member functions**

| | |
|---|---|
| map <int, PERSON>:: iterator find_person(int id) | |
| void birth(PERSON *p) | Performs all necessary operations when a birth event happens to parent **p** |
| void marriage(PERSON *p) | Performs marriage search start and/or marriage for partner **p** |
| void new_marriage(PERSON *p, PERSON *spouse) | |
| void death(PERSON *p) | Performs all operations for event death to person **p** |
| void divorce(PERSON *p) | Performs divorce initiated by person **p** |
| void transit(PERSON *p, int hhmigration, int destgroup) | Performs transit of person p from group p->group to destgroup.<br>NOT FULLY IMPLEMENTED |
| void assemble_household(PERSON *p, list<int> transit_list, int hhmigration) | Returns a list of ID of members of household headed by person p |
| void add_minor_children(PERSON *p, list<int> transit_list) | |
| int check_spouse(PERSON *p1, PERSON *p2) | Check whether people p1 and p2 are close relatives and hence can not be married |
| PERSON* pgrand_p(PERSON *p, string kincode) | Return pointer to grand-parent identified by string = mm, mf, ff, fm |
| MARRIAGE *plast_marriage(PERSON *p) | Return pointer to PERSON's last marriage |
| PERSON* pspouse(PERSON *p); | Return pointer to PERSON's spouse in last marriage |
| **PERSON and SEGMENT utilities** | |
| void event_this_time_step(PERSON* p, string& event, int& new_group) | Returns event that will happen this time step as string:<br>"b" - child birth to parent **p**<br>“d” – death<br>“m” – start of marriage search |

| | |
|---|---|
| | “d” – divorce initiated by **p**<br>“c” – start of cohabitation search<br>“t” – transit to a new group<br>Returns "" if no event happens<br>If event ="t" **new_group** contains number of the new group<br>MODE “T” NOT TESTED |
| int process_time_step() | Performs all operations to model the population processes during 1 model time step |
| int date_and_event_rt(PERSON *p) | NOT IMPLEMENTED |
| void modify_rates(PERSON *p, int event, struct age_block * first_block) | NOT IMPLEMENTED |
| int lc_datev(PERSON *p, int g, int s, int age, SEGMENT *ptr_seg) | NOT IMPLEMENTED |
| void create_working_mqueue(PERSON *p) | NOT IMPLEMENTED |
| int datev_rt(struct age_block * first_block, int age, int time_shift) | NOT IMPLEMENTED |
| **SEGMENT utilities** | |
| int get_expected_number_of_births(int group) | OBSOLETE |
| int get_expected_number_of_transits(int from, int dest) | OBSOLETE |
| **POPULATION utilities** | |
| int generate_initial_population(string type, int pop_size, double prop_male, string fname) | Not fully implemented. Generates initial population according to a certain built-in lifetable |
| int read_permap_MAXIM(int input_left_trunc, int input_right_c) | Load individual data from .opop file in MAXIM format |
| int read_permap_SOCSIM(int input_left_trunc, int input_right_c) | Load individual data from .opop file in MAXIM format (for compatibility) |
| void read_xtra_MAXIM() | Load values of individual parameters from .xtra file into individual parameter maps |
| int read_marmap_MAXIM() | Load marriage data from .omar file |
| void cross_link_input() | Cross link individuals and marriages |
| void save_pop_data(string fname, string msg, int max_birth_date, int mode) | |
| //void write_marriages(ofstream& f, string msg) | |
| //void write_xtra(ofstream& f, string msg) | |
| //void POPULATION:: prepare_output_files(int segnum) | |
| **POPULATION reporting** | |
| oid census1(int current_month) | |
| void census2(int input_right_censor) | |
| void child_census(int current_month) | |

| | |
|---|---|
| void marriage_tally(int current_month, ofstream& f) | |
| void report(ofstream& f, string text) | |
| void pyramid(int mode) | |
| int count_current_population(string sex, int group) | |
| int time_on_marqueue(string sex) | |
| //static size_t count() | OBSOLETE Returns number of living/ |
| **Group-related methods** | |
| void groups_print(ostream& f) | Print current info about groups to **f** |
| void groups_add_person(PERSON *p) | Add a person to the group defined by p->group |
| void groups_update(string event, PERSON *p, int new_group) | Group "housekeeping" for event **event**<br>Called for event = "birth", "death" or "transit". Updates group sizes.<br>Creates new and removes empty groups as needed.<br>In mode="transit" new_group is used, p->group should contain original group (group before transit).<br>Method does not change p->group, only GROUP internals. |
| double lookup_table(string tablename, int month); | Lookup in a table defined by **tablename**. For use in user-defined plugins that require data from tables. No data conversion, time-specific data in tables is treated as stepwise. Value is looked up at time **time**.<br>Data should be read into tables by means of **table** command in the controlling .sup file |

## class SEGMENT

Holds all data (rates, life and data tables, plugin references, standard parameters etc.) relevant to the current simulation segment.

**Member functions:**

| | |
|---|---|
| void initialize_segment_vars(); | |
| int fill_missing_LC_rate_tables(); | OBSOLETE |
| //int fill_missing_rate_tables(POPULATION* ptr_pop); | DEPRECATED |
| int fill_rate_gaps(POPULATION *ptr_pop); | |
| void dump_rates(POPULATION *ptr_pop); | |
| void report(POPULATION *ptr_pop, ostream& fd); | // Information: print data defined for the current segment |
| void print_params(ostream& f); | Prints parameters defined for the segment |
| void print_tables_rate_patterns(ostream& f); | Prints all data tables (typed |

| | |
|---|---|
| | and free-type) defined for the current segment to stream **f** |
| void print_event_plugin_hooks(ostream& f); | Prints hooks to event-driven plugins |
| void print_stat_plugin_hooks(ostream& f); | Prints hooks to time-driven statistical plugins |
| int stat_init(POPULATION* ptr_pop); | |
| int stat_dump(POPULATION* ptr_pop); | |
| int set_param(string parname, string parvalue); | Works with standard set of segment parameters (SOCSIM set) |
| **Work with vital rates stored in tables and generated by model plugins** | |
| int compose_table_name(string& table_name, string rate_type, SUP_OBJECT *sup_object); | Composes table name for look-up |
| RATE_SOURCE get_rate_source(string event, PERSON *p); | |
| int get_annual_rate(POPULATION *ptr_pop, PERSON *p, RATE_SOURCE rate_source, string event, int time, double& rate); | |
| //int birth_rate_set(int group, string mstatus, int parity, TABLE& table); | Alias, custom case of get_rate |
| int cohab_probs(int group, string sex, TABLE& table); | Alias, custom case of get_rate. Returns the whole rate table for cohabitation probs |
| //double table_lookup(int age, TABLE table); | Low level routine. Returns rate in action for given age |
| //int rate_set(int group, string event, string sex, string mstatus, TABLE& table) | Returns the whole rate table for use in POPULTION:: get_expected_number_of_births |
| void process_event_plugins(POPULATION *ptr_pop, PERSON *p, string event) | Plugin-support functions. Processes all plugins hooked to event **event** upon individual **p** |
| void print_rate_patterns_cache() | Prints current contents of the rate patterns cache |

# Extending MAXIM. Programming new and modifying existing modules

## How to…

### Change the spouse selection model and scoring function

The partner is selected for cohabitation or marriage from the marriage queue of the opposite sex based on the partner selection model and score.

The partner selection model (e.g. random, avoid close relatives or something more elaborate) is given by the symbol link to a function

`PERSON::pref` in the maxim.h. MAXIM is shipped with `pref= &best_spouse`

In this case the function best_spouse() (located in misc.cpp) performs the checks and selection of the partner.

In it first the unacceptable candidates (close relatives) are removed and the rest are ranked using the score function.

The type of the score fucntione is defined by

`PERSON::score` in the maxim.h. MAXIM is shipped with `score = &score4`

To change program the new spouse selection routine and/or score function, change the symbolic links and rebuild.

## Programming plug-ins

Functionality of MAXIM can be greatly extended by means of so called “plugins”. A plugin is a function in C++ language, which should be written by user and put into **`plugins.cpp`** file. After the program is recompiled this function is available to the program.

Functions have standardized interface (list of input parameters and result) and in principle have access to all information about the population. Hence plugin functions can collect and modify information about the individuals.

*Note: it is a good idea to understand how the MAXIM plugins work by studying built-in plugins before your start creating your own. Plugins are powerful, but you can cause yourself lots of trouble if you do things in a wrong way. Badly written plugins can easily cause to run time errors and such things.*

There are 4 types of plugins (with 3 different interfaces):

1. **Rate generators.** Called for a certain individual and return a double value.

Used to generate a value treated as annual rate mx for some event. This allows writing rate models of vital rates based on the current state of the individual and (possibly) other individuals and values of population-wide parameters.

2. **State alternators.** They are called for a certain individual and return a double value.

The output value is ignored, such plugins are used to modify the state of the individual (e.g. value of his parameters). This allows implementing inheritance models, models for additional parameters of the individuals such as education and similar.

3. **Rate modifier plugins**. These are used to modify the annual mx rate obtained a generator plugin or a life table based on individual or population-wide parameters – e.g. to easily scale rates or create a family of rates from one by some functional transpositions.

4. **Statistical plugins.** Called for the whole population and return a string.

Such plugins are referred to as “statistical” since they collect some information about the population which is then written to a text file. Of course they can collect not just statistical, but any kind of information about the population.

Both types of plugins have access to all members of the population, alive or dead, individual and population-wide parameters, can read and alter them.

Here we describe both types plugins, how to program and use them and plugins that are provided with MAXIM.

**Rate generators** and **state alternator plugins** have the same prototype.

## *Generator/alternator plugins*

### Compiling

Should be located in `plugins.cpp` file.

To implement a new plugin you should create a function with interface

```
double plugin_name(POPULATION *ptr_pop, PERSON *p) {
}
```

and modify function `plugin()` located in `plugins.cpp` by adding 2 lines:

```
else if (plugin_name == "plugin_name")
    return(plugin_name(ptr_pop, p));
```

in the long `if else if` … statement similarly to the statements for other plugins already present there. The `pligin()` function acts as a switch box, calling that or the other generator/modifier plugin by name.

Here `plugin_name` should be replaced by your name of the plugin. It should not repeat names of other plugins.

Theoretically you can give your plugin any name but to avoid possible names clashes with other functions of MAXIM we advise to call all generator/modifier plugins `plugin_*****`, where `*****` is any name you like.

In the plugin interface `ptr_pop` is the pointer to the instance (only one existent) of the class `POPULATION` that contains all the information about the population, modelling segments etc.

`p` is a pointer to the instance of class `PERSON` which contains information about the individual for which the plugin was called.

Both classes are defined in `maxim.h` file.

### Accessing data from plugins

Standard, built-in variables of the individual and his basic traits (such as sex, marital status, current age) can be accessed by expressions of the form

```
p->variable
```

where variable is any of individual parameters. See the complete list and description in `maxim.h`.

**Most widely used are:**

```
string sex # “f” or “m”
string mstatus # marital status
int group, birthdate, deathdate;
int mother, father
double fmult # fertility multiplier, used for females if hetfert=1
int marity, parity # number of marriages, children
```

**Others:**

```
int prev_group # previous group
int migration_date;
```

```
string prev_marital_status;
int factor;
int mqueue_month;
int birth_group;
```

For example we can obtain birth date of the individual as `p->birthdate`.

Similarly parameters defined in the `POPULATION` class can be accessed by using the `ptr_pop` pointer. For example we can get the current month of the simulation (the first month of the time step if it is longer than 1 month) as

```
ptr_pop->current_month
```

**Example:** to get the age of the individual we can use

```
int age = ptr_pop->current_month - p->birthdate;
```

In order to read and update additional (extra) individual and population-wide parameters MAXIM provides 2 functions:

```
double PERSON:: get_xtra_param(POPULATION* ptr_pop, string parname)
int PERSON:: set_xtra_param(POPULATION* ptr_pop, string parname, double value)
```

`get_xtra_param` searches for the parameter named `parname` first in the list of the population-wide and then individual parameters and return its value. Individual parameters override values of population-wide parameters with same names.

If parameter is not found, fatal error occurs and the program pauses and terminates.

Example:

Getting a value of parameter `"userpar"` from a plugin function:

```
double userpar;
userpar = p->get_xtra_param(ptr_pop, "userpar");
// do something
```

Function `set_xtra_param` updates the value of an existing population or individual parameter. If parameter does not exist, fatal error occurs and the program pauses and terminates.

The set of all additional parameters for an individual is created from data in .opox file for initial population and at birth for newborns. Users can not create additional individual parameters during runtime.

If parameter does not exist, fatal error occurs and the program pauses and terminates.

The user does not have to worry about whether a parameter is defined on the population or individual level. If their names are different they would be accessed correctly.

*Note: newborn individuals have no individual parameters defined. It is the duty of the user to supply an "inheritance" function which is hooked to newborn event which would create and initialize his individual parameters based on parents' values or some defaults.*

A bit more elaborated is accessing information about parents and other individuals. Generally the easiest way to get it is to get the pointer to the record of the other individual and then get all the required information.

**Example. Getting information about mother**

integer `id` of mother is stored in the `p-> mother`. Check that it is not zero. `ptr_pop->permap[p->mother]` is the instance of class `PERSON` that stores information for mother. So if we want to get the value of `par1` for mother we could write

```
double par1;
```

```
  if (p->mother !=0)
      par1   =   (ptr_pop->permap[p->mother]).get_xtra_param(ptr_pop,
"par1")
  else
      // define parameter if no mother exits
```

### Invoking plugins

The generator/modifier plugins actually comprise 2 types: generator and modifier plugins invoked in different ways.

### Invoking alternator plugins

Plugins whose return value is ignored and which are used to modify information of the individuals are **alternator** and should be “hooked” to events.

Events can happen to an individual when his is born, dies or his state changes, to all individuals in the beginning of the modelling time step or once in the beginning of the modelling time step. They are described in the table

| Hook name | Occurrence | Called for | Examples of application |
|---|---|---|---|
| **timestep_once** | In the beginning of each simulation time step before individuals are processed | No real individual. Called with a pointer to a dummy individual | Actions performed once for the whole population such as collection of statistics upon the population used later by plugins (e.g. counting), models of the environment (food, temperature, infection) |
| **timestep_each** | In the beginning of each simulation time step | For each living individual, before event testing | Model events that always happen to the individual such as hunting, consumption, internal model of disease for individual etc. |
| **newborn** | Directly after the new individual is born | This, newborn individual | Initialization of the individual variable of the newborn – e.g. inheritance |
| **birth** | After the offspring is born | Mother | Update the state of the mother, e.g. her resources |
| **death** | After death of the individual | Individual that has been just diseased | Decrease the size of the family, group |
| **marriage** | After the event | To both partners (???) | |
| **divorce** | After the event | To both partners (???) | |
| **cohabit** | After the event | To both partners (???) | |
| **split** | After the event | To both partners (???) | |
| **transit** | After the event | Transiting individual | |

The plugin is invoked by adding a line of the form **`hook plugin_name`** to the .sup file.

Example:

```
timestep_once plugin_pop_size
```

Calls **`plugin_pop_size`** for a dummy individual once in the beginning of each time step (counts population and stores total number in population-wide parameter **`"pop_size"`**)

```
newborn   plugin_inheritance
```

Calls plugin **`plugin_inheritance`** for the newborn just after it has been created (initializes his parameters based on those of parents, i.e. performs inheritance)

## Invoking generator plugins

Plugins that typically do not alter values of individual parameters but generate some output double value based on the current state of the individual, population and its variables are referred to as generators. Their value may be used as the annual rate for some event.

Examples of such plugins are various parametric models of mortality, fertility and other processes that return rates based on state of individual and values of some parameters. These can of course be more complex functions like table look-ups etc.

Such plugins are “hooked” to rate patterns as was described previously.

**Example:**

One of the standard MAXIM plugins is the Siler model of mortality [W. Siler A competing risk model for animal mortality Ecology 60(4), 1979, pp. 750-757]. It describes annual mortality rate basing on 5-parameter model:

mu(x) = a1*exp(-b1*x)+a2+a3*exp(b3*x),

where x is age in years.

- define population-wide parameters **`siler_a1, siler_b1, siler_a2, siler_a3, siler_b3`** in the .sup file or individual, in the **`.opox`** file
- already have the built-in plugin plugin_mortality_model_siler defined in the **`plugins.cpp`** file:

```
double plugin_mortality_model_siler(POPULATION *ptr_pop, PERSON *p)
{
     // Siler 5-parameter model for death rates
     // W. Siler A competing risk model for animal mortality
     // Ecology 60(4), 1979, pp. 750-757.

     // ANNUAL RATES
     // Model: m(age) = a1*exp(-b1*age)+a2+a3*exp(b3*age)
     // for Italy1931male_single_death.txt good estimation
     // a1=0.038 b1=1.105 a2=0.001 a3=0.0005 b3=0.074

    int age_months;
    double age;
    double a1, a2, a3, b1, b3; // parameters of the Siler model
    double rate=0;
    int debug=1;

    age_months = ptr_pop->current_month - p->birthdate;
    if (age_months<0 || p->deathdate!=0) {
        cerr<<"Error:  plugin_mortality_model_siler():  person  age
negative or already dead!\n";
        system("pause");
        exit(1);
    }
    // for these parameters we need age in years
    age = (double) age_months/(double)12;

    a1 = p->get_xtra_param(ptr_pop, "siler_a1");
    b1 = p->get_xtra_param(ptr_pop, "siler_b1");
    a2 = p->get_xtra_param(ptr_pop, "siler_a2");
    a3 = p->get_xtra_param(ptr_pop, "siler_a3");
    b3 = p->get_xtra_param(ptr_pop, "siler_b3");
    rate = a1*exp(-b1*age) + a2 + a3*exp(b3*age);
    return(rate);
}
```

- Hook this plugin to the mortality rate pattern, e,g, for all individuals, adding a line to the model .sup file:

```
death * * * plugin_mortality_model_siler
```

### Invoking modifier plugins

Plugins input a value treated as annual mx rate and output a function of it.

Conversion function may as simple as a constant (e.g. built-in plugin `mod_plugin_unity`) or depending on parameters of individual or population.

Examples of such plugins are scaling models.

Such plugins are “hooked” to rate patterns usind **mod** command.

**Example:**

```
death 1 f single * plugin plugin_mortlaity_female mod
mod_plugin_times2
```

Mortlaity rate (mx) for group 1, females, single, all parities is generated by a rate generator plugin `plugin_mortlaity_female` and passed through a modifier plugin `mod_plugin_times2` which increases the rate two times.

### *Statistical plugins*

These are invoked periodically, collect some information about the population and return a string with text result, usually a row of the file.

### Compiling

Should be located in `plugins.cpp` file.

To implement a new stat plugin you should create a function with interface

```
string stat_plugin_name(POPULATION *ptr_pop, int mode)
```

and modify function `stat_plugin()` located in `plugins.cpp` including 2 lines:

```
     else if (plugin_name == "stat_plugin_name")
          return(stat_plugin_name (ptr_pop, mode));
```

in the long if else if … statement similarly to the statements for other plugins already present there. The `stat_pligin()` function acts as a switch box, calling that or the other stat plugin by name.

Here `plugin_name` should be replaced by your name of the stat plugin. It should not repeat names of other plugins.

Theoretically you can call your plugin any name. But to avoid possible names clashes with other functions of MAXIM we advise to call all stat plugins `stat_plugin_*****`, where `*****` is any name you like.

In the stat plugin interface **`ptr_pop`** is the pointer to the instance (only one existent) of the class **`POPULATION`** that contains all the information about the population, modelling segments etc.

mode is a flag telling the plugin whether is should return a descriptive header of the file (typically containing column names) (value 0) or a table row with values (value 1).

Class **`POPULATION`** is defined in **`maxim.h`** file.

### Accessing data from plugins

Data can be accessed by the stat plugins in the same way as it is done by generator/modifier plugins.

Example: a simple stat plugin counting the population (included in MAXIM)

```
string stat_plugin_pop_size(POPULATION *ptr_pop, int mode)
{
    char s[255];
    if (mode==0) {
        sprintf(s, "%6s %7s %7s", "time", "males", "females");
    } else {
        sprintf(s, "%6d %7d %7d", ptr_pop->current_month,
                ptr_pop->count_current_population("m", 1),
                ptr_pop->count_current_population("f", 1));
    }
    return(s);
}
```

*Note how the* **`sprintf()`** *function is used to format rows so that names and numbers align in 3 columns.*

It is a good habit to include time as the first column in the plugins output since that obviously would output time-varying data

### Invoking statistical plugins

Invoking is done from the model .sup file with a line of the form

```
stat   stat_plugin_name   period  filename
```

where stat is a key word, stat_plugin_name is the name of your plugin, period is an integer value, describing how often a plugin should be called – in time steps. So 100 will mean “every 100th model time step”, filename is the name of the file where the output should go, e.g. “pop_size.txt”.

## *Plugins built into MAXIM*

| **Plugin name** | **Plugin type** | **Note** |
|---|---|---|
| plugin_template | generator/modifier | Template to create a user plugin |
| plugin_mortality_model_siler | generator | Return annual mortality rate calculated according to the Siler model [W. Siler A competing risk model for animal mortality Ecology 60(4), 1979, pp. 750-757]<br>Rate is evaluated by formula m(x)=a1*exp(-b1*x)+a2+a3*exp(b3*x), where x is age in years<br>Requires population-wide or individual parameters with names siler_a1, siler_b1, |

| | | |
|---|---|---|
| | | siler_a2, siler_a3, siler_b3<br>Good approximation for modern mortality (Italy, 1931, single males) is obtained with parameters a1=0.038 b1=1.105 a2=0.001 a3=0.0005 b3=0.074 |
| plugin_pop_size | modifier | Counts the population and stores the total population size in variable pop_size<br>For correct work a population-wide parameter with this name should be defined in .sup file |
| stat_plugin_template | stat | Template for user statistical plugins |
| stat_plugin_pop_map_size | stat | Size of the population map. Outputs data for time and pop_map_size |
| stat_plugin_pop_size | stat | Count number of males and females in all groups |
| stat_plugin_pop_pyr | stat | Calculate population size and proportions of individual in age groups (“population pyramid”) separately by sexes, for all groups |

### Hints and warnings

Plugins are powerful and potentially dangerous to the stability of the program. Before creating your own plugins study templates and those provided with MAXIM. Copy and modify them as required.

Every generator/modifier plugin is invoked for a specific individual. When a plugin is invoked the whole information about this individual is available via **p** pointer. Study the PERSON class to understand what variables you can use. DO NOT CHANGE VALUES OF THE MAIN PARAMETERS OF THE INDIVIDUAL unless you understand what you do. Read them, do not alter them!

Parameters of the current segment, such as mostly used, current_month, can be obtained using the pointer to the population. The construction **ptr_pop->current_month** get the current month. The same rule applies: read, use the value, but do not alter it!

Write robust programs! Check the input data and result codes and generate error messages when needed! In MAXIM error result 0 generally means “OK” and non zero codes mean “Error”.

## *Practical examples of plugins*

### Parametric models of vital rates

A simple way to define a vital rate is to write a parametric function (similar to p.d.f.) defining how the rate depends on parameters. This requires passing one or several parameters to the function. In a heterogeneous population the parameter values typically vary from person to person, but they also can be same for all individuals (such parameters are referred to as population-wide here).

Additional individual parameters, i.e. those beyond standard parameters, describing the state of the living individual (**birthdate, sex, motherid, fatherid, marstatus, group, marity, parity)** are loaded from the .opox file and stored in the **map<string, double> aux_param** structure. They can be obtained by get_xtra_param function.

## Inheritance of individual parameters

All additional personal parameters used in the model in addition to the base set need to be initialized when a new individual is born. This may be described as inheritance of parameters. The set of individual parameters of the offspring may be fixed, or depend on the values or mother and father, maybe in a stochastic way.

To simplify things, the program initializes base parameters and then generates a **`newborn`** event to which user can hook a modifier plugin which will perform the parameter initialization.

The plugin will be called with the pointer to the newborn, using the pointer to the population ptr_pop it can access information from the records of father and mother (chack existence!) and then create individual parameters of the newborn with required values.

Use function **`create_xtra_param`** to create a new individual parameter for the newborn.

## Big example

The following example demonstrates how to setup a model describing a hunter-gatherer population in which people hunt and share resources among the whole tribe.

Idea:

Tribe has a common pool of food. Hunters hunt and add food to this pool every time step. All individuals get food from the food pool.

Food units are kilocalories

**.opox** file:

```
# Individuals have 1 parameter: individual reserve.
# People get food from the population food pool and add to the
# personal food reserve. They spend reserve according to
# their requirements.
# Personal amount of reserve (“fat”) influences individual
# mortality and fertility

# Define some starting nonzero amount enough for 1-2 months
# 2000*30 = 60000

personal_reserve
60000
60000
60000

….
60000
```

**.sup** file:

```
# Males hunt and get certain age-specific amounts of food monthly
# Gathered food is added to the population food pool
# Every month each individual gets a certain amount of food
# from the pool
# Amount of food influences fertility and mortality
…… # load or generate an initial population
param population_pool 6000000 # Population-wide parameter.
                        # Some nonzero initial value
                        # value to prevent people
                        # from dying on the first month

param pop_size 100      # Set to the initial population size

# It is slow to calculate the population size on every time step
```

```
  # Alternative was is to update is: increment upon birth and
  # decrement upon death

  # Time-driven plugins are called for each living individual
  # every time period
  monthly plugin_hunt     # defines age- and sex- specific produc-
tion.
                          # Adds hunted food to population_pool

  month plugin_consume    # Take a share from the population_pool
                          # and add it to the personal_reserve.
                          # Consume some from personal reserve
                          # may use some population “statistics” –
e.g.
                          # stored in pop_size variable

  birth plugin_birth_rate_modifier # Calculate birth rate using
                          # information about the amount of
                          # personal reserve

  death plugin_death_rate_modifier # Similar to the birth rate block
  birth plugin_increment_popsize   #
  death plugin_decrement_popsize   # Event-driven plugin.
                                   # Updates the pop_size parameter
  birth plugin_inheritance         # Init. parameters of newborn
                                   # Called after a child is created
                    # Set the child’s xtra parameters
                    #  based  on  those  of  his  mother  and  father  (if
any)

  birth 1 F single *   plugin_birth_energy  # model evaluating
                           # birth rate depending from enrgy level

  death 1 F single plugin_death_energy
  …. table ….
  death 1 M single
  …. table ….
  marriage 1 * single
  ……
  divorce 1 * married
  ………
  run
```

## *Algorithm diagrams*

These diagrams show schematically the flow of most important stages of program work so that a programmer could be able to understand and modify the program easier.

The diagrams are simplified in comparison to the original algorithms.

Brackets () denote names of MAXIM functions

### Testing for occurrence of specific event to an individual

Testing for event X, individual in state {group, sex, mar_status, parity} $\rightarrow$ test_for_event() $\rightarrow$

compose rate pattern from event and status: birth 1 f s 2 $\rightarrow$ “b_1_f_s_2” $\rightarrow$

search for pattern in rate pattern cache.

**Found?**

```
    yes: get rate_source from cache
no: match pattern and find appropriate rate_source using get_rate_source()
    Have rate_source. Get annual mx rate by get_annual_rate() →
            is rate_source.table_ID defined? Y: rate = lookup_mx()
                    Get data from table by name and convert data for given time to mx
                format based on table type
            else is rate_source.genplugin_ID defined? Y: rate = plugin()
                    Invoke built-in or user plugin programmed in plugins.cpp by name
            - get rate from table or plugin
            Is rate_source.modplugin_ID defined? Y: rate = mod_plugin() – modify rate
        by a modifier plugin
                    Invoke built-in or user modifier plugin programmed in plugins.cpp
                by name
    Evaluate probability of event to happen within the timestep.
    Perform stochastic test for event with this probability
    Return 1 if test successful
```

## Parsing .sup file – description of rates

```
Line not numeric, found 5 or more tokens →
token1 in dictionary of events?
Y: convert tokens 1..5 to rate pattern by compose_table_name()
N: error
Conversion OK?
Y: make new rate_source instance
        Search line for "plugin" keyword. Found in position n?
        Y: get plugin name: rate.source.genplugin_ID = token(n+1)
        N: No plugin defined. So are expecting table. Set context="lifetable"
        Search line for "mod" keyword. Found in position n?
        Y: get mod plugin name: rate.source.modplugin_ID = token(n+1)
        N: No plugin defined. Expecting to find a life table next. Set con-
    text="lifetable"
```

Non-numeric lines mean end of reading table (if context is not “”) except for expression type **** that concretises the type of the life table.

```
and table ****
2 token line:
        token(1)=="type"?
                Y: context=="lifetable"?
                        Y: set type of the current table to token(2)
                        N: error: "type command outside of table context"
        token(1)=="table"?
                Y: create a new TABLE instance, set name, set context = "datatable"
```