LIAM 2 User Guide

Release 0.4.1

G. Bryon, G. Dekkers, G. de Menten

December 02, 2011

CONTENTS

CHAPTER

ONE

INTRODUCTION

LIAM 2 is a tool to develop (different kinds of) microsimulation models.

1.1 About LIAM2

The goal of the project is to let modellers concentrate on what is strictly specific to their model without having to worry about the technical details. This is achieved by providing a generic microsimulation toolbox which is not tied to a particular model. By making it available for free, our hope is to greatly reduce the development costs (in terms of both time and money) of microsimulation models.

The toolbox is made as generic as possible so that it can be used to develop almost any microsimulation model as long as it use cross-sectional ageing, ie all individuals are simulated at the same time for one period, then for the next period, etc.

You can find the latest version of LIAM2 and this documentation at: http://liam2.plan.be

1.2 About this guide

This guide will help you develop dynamic microsimulation models using LIAM 2. Please note that it describes version 0.4 of LIAM 2, but both the software package and this manual are very much work-in-progress, and are therefore subject to change, including in the syntax described in this manual for defining models.

1.3 Microsimulation

Microsimulation is (as defined by the International Microsimulation Association), a modelling technique that operates at the level of individual units such as persons, households, vehicles or firms. Each unit has a set of associated attributes – e.g. each person in the model has an associated age, sex, marital and employment status. At each time step, a set of rules (intended to represent individual preferences and tendencies) are applied to these units leading to simulated changes in state and possibly behaviour. These rules may be deterministic (probability = 1), such as ageing, or stochastic (probability < 1), such as the chance of dying, marrying, giving birth or moving within a given time period.

The aim of such simulations is to give insight about both the overall aggregate change of some characteristics and (importantly) the way these changes are distributed in the population that is being modelled.

1.4 Credits

LIAM2 is being developed at the Federal Planning Bureau (Belgium), with funding and testing by CEPS/INSTEAD (Luxembourg) and IGSS (Luxembourg), and funding from the European Community. It is the spiritual successor of LIAM 1, developed by Cathal O'Donoghue.

More formally, it is part of the MiDaL project, supported by the European Community Programme for Employment and Social Solidarity - PROGRESS (2007-2013), under the Grant VS/2009/0569 Abstract - Project PROGRESS MiDaL deliverable Work Package A.

CHAPTER

TWO

ENVIRONMENT

2.1 LIAM 2 bundle

The bundle includes:

- The executable.
- A text editor (Notepad++), pre-configured to work with LIAM2 models.
  - Notepad++ is a free (and open source) text editor that is available at http://sourceforge.net/projects/notepad-plus/.
  - We pre-configured it so that you can import csv files and run your models directly from within the editor by simply pressing F5 or F6 respectively. See below for more information.
- The documentation in html, pdf and chm (windows help) format.
- A demonstration model with a synthetic data set of 20,200 persons grouped in 14,700 households.

2.2 Getting Started

- Copy the contents of the bundle in a directory on your disk (let's call it *\localpath*).
- Run the "Notepad++Portable.exe" from the *\localpath\Liam2Suite\editor* directory.
- Open a model (eg. \localpath\Liam2Suite\examples\demo01.yml)
- Press F6 to run the model. A console window will open within the editor and display the status of the simulation. After the simulation completes, the console window becomes interactive.
- Use this console to explore the results.

2.3 Using your own data

- Prepare your data as CSV files. The first row should contain the name of the fields. You need at least two integer columns: "id" and "period" (though they do not necessarily need to be named like that in the csv file).
- Create an import file, as described in the *Importing data* section. You can use \localpath\Liam2Suite\examples\demo_import.yml as an example.
- Press F5 to convert your CSV files to hdf5.
- Use the newly created data file with your model.

CHAPTER

THREE

MODEL DEFINITION

To define the model, we have to describe the different *entities*, their *fields*, the way they interact (*links*) and how they behave over time (*processes*). This is done in one file. We use the YAML-markup language. This format uses the level of indentation to specify objects and sub objects.

In a LIAM 2 model file, all text following a # is considered to be comments, and is therefore ignored.

A LIAM 2 model has the following structure:

```
globals:
    ...

entities:
    ...

simulation:
    ...
```

3.1 globals

The *globals* are variables (aka. parameters) that do not relate to any particular *entity* defined in the model. They can be used in expressions across all entities.

Periodic globals can have a different value for each period. For example, the retirement age for women in Belgium has been gradually increasing from 61 in 1997 via 63 from 2003 onward, up to 65 in 2009. A global variable WEMRA has therefore been included.:

```
globals:
    periodic:
        - WEMRA: float
```

Periodic globals can be used in any process. They can be used in two ways: like a normal variable, they will evaluate to their value for the period being simulated, for example

```
workstate: if(age >= WEMRA, 9, workstate)
```

This changes the workstate of the individual to retired (9) if the age is higher than the required retirement age in that year.

Another way to use them is to specify explicitly for which period you want them to be evaluated. This is done by using GLOBALNAME[period_expr]. periodexpr can be any expression yielding a valid period value. Here are a few artificial examples:

```
workstate: if(age >= WEMRA[2010], 9, workstate)
workstate: if(age >= WEMRA[period - 1], 9, workstate)
workstate: if(age >= WEMRA[year_of_birth + 60], 9, workstate)
```

3.2 entities

Each entity has a unique identifier and a set of attributes (**fields**). You can use different entities in one model. You can define the interaction between members of the same entity (eg. between partners) or among different entities (eg. a person and its household) using *links*.

The **processes** section describe how the entities behave. The order in which they are declared is not important. In the **simulation** block you define if and when they have to be executed, this allows to simulate processes of different entities in the order you want.

LIAM 2 declares the entities as follows:

```
entities:
    entity-name1:
        fields:
            fields definition

        links:
            links definition

        macros:
            macros definition

        processes:
            processes definition

    entity-name2:
        ...
```

As we use YAML as the description language, indentation and the use of ":" are important.

3.2.1 fields

The fields hold the information of each member in the entity. That information is global in a run of the model. Every process defined in that entity can use and change the value.

LIAM 2 handles three types of fields:

- bool: boolean (True or False)
- int: integer
- float: real number

There are two implicit fields that do not have to be defined:

- id: the unique identifier of the item
- period: the current period in the run of the program

example

```
entities:
    person:
        fields:
            # period and id are implicit
            - age:        int
            - dead:       bool
            - gender:     bool
            # 1: single, 2: married, 3: cohabitant, 4: divorced, 5: widowed
            - civilstate: int
            - partner_id: int
            - earnings:   float
```

This example defines the entity person. Each person has an age, gender, is dead or not, has a civil state, possibly a partner. We use the field civilstate to store the marital status as a switch of values.

By default, all declared fields are supposed to be present in the input file (because they are *observed* or computed elsewhere and their value can be found in the supplied data set). The value for all declared fields will also be stored for each period in the output file.

However, in practice, there are often some fields which are not present in the input file. They will need to be calculated later by the model, and you need to tell LIAM2 that the field is missing, by using "initialdata: false" in the definition for that field (see the *agegroup* variable in the example below).

example

```
entities:
    person:
        fields:
            - age:      int
            - agegroup: {type: int, initialdata: false}
```

Field names must be unique per entity (i.e. several entities may have a field with the same name).

3.2.2 links

Entities can be linked with each other or with other entities, for example, individuals belong to households, and mothers are linked to their children, while partners are interlinked as well.

A typical link has the following form:

```
name: {type: <type>, target: <entity>, field: <name of link field>}
```

LIAM 2 uses integer fields to establish links between entities. Those integer fields contain the id-number of the linked individual.

LIAM 2 allows two types of links: many2one and one2many.

More detail, see *Links*.

3.2.3 macros

Macros are a way to make the code easier to read and maintain. They are defined on the entity level. Macros are re-evaluated wherever they appear. Use *capital* letters to define macros.

example

```
entities:
    person:
        fields:
            - age: int

        macros:
            ISCHILD: age < 18

        processes:
            test_macros:
                - ischild: age < 18
                - before1: if(ischild, 1, 2)
                - before2: if(ISCHILD, 1, 2)   # before1 == before2
                - age: age + 1
                - after1: if(ischild, 1, 2)
                - after2: if(ISCHILD, 1, 2)    # after1 != after2

simulation:
```

```
    processes:
        - person: [test_macros]
```

The above example does

- ischild: creates a temporary variable *ischild* and sets it to *True* if the age of the person is under 18 and to *False* if not
- before1: creates a temporary variable *before1* and sets it to 1 if the value of the temporary variable *ischild* is *True* and to 2 if not.
- before2: creates a temporary variable *before2* and sets it to 1 if the value age < 18 is *True* and to 2 if not
- age: the age is changed
- after1: creates a temporary variable *after1* and sets it to 1 if the value of the temporary variable *ischild* is *True* and to 2 is not.
- after2: creates a temporary variable *after2* and sets it to 1 if the value age < 18 is *True* and to 2 if not.

It is clear that after1 != after2 since the age has been changed and *ischild* has not been updated since.

3.2.4 processes

Here you define the processes you will need in the model.

More detail, see *Processes*.

3.3 simulation

The *simulation* block includes the location of the datasets (**input**, **output**), the number of periods and the start period. It sets what processes defined in the **entities** block are simulated (since some can be omitted), and the order in which this is done.

Suppose that we have a model that starts in 2002 and has to simulate for 10 periods. Furthermore, suppose that we have two object or entities: individuals and households. The model starts by some initial processes (grouped under the header *init*) that precede the actual prospective simulation of the model, and that only apply to the observed dataset in 2002. These initial simulations can pertain to the level of the individual or the household. Use the *init* block to calculate variables for the starting period.

The prospective part of the model starts by a number of sub-processes setting the household size and composition. Next, two processes apply on the level of the individual, changing the age and agegroup. Finally, mortality and fertility are simulated. Seeing that this changes the numbers of individuals in households, the process establishing the household size and composition is again used.

example

```
simulation:
    init:
        - household: [household_composition]
        - person: [agegroup]

    processes:
        - household: [household_composition]
        - person: [
                age, agegroup,
                dead_procedure, birth
           ]
        - household: [household_composition]

    input:
        path: liam2
        file: base.h5
```

```
    output:
        path: liam2
        file: simulation.h5
    start_period: 2002
    periods: 10
    random_seed: 5235       # optional
```

3.3.1 processes

This block defines which processes are executed and in what order. They will be executed for each period starting from *start_period* for *periods* times. Since processes are defined on a specific entities (they change the values of items of that entity), you have to specify the entity before each list of process. Note that you can execute the same process more than once during a simulation and that you can alternate between entities in the simulation of a period.

In the example you see that after dead_procedure and birth, the household_composition procedure is re-executed.

3.3.2 init

Every process specified here is only executed in the last period before *start period* (start_period - 1). You can use it to calculate (initialise) variables derived from observed data. This section is optional (it can be entirely omitted).

3.3.3 input

The initial (observed) data is read from the file specified in the *input* entry.

Specifying the *path* is optional. If it is omitted, it defaults to the directory where the simulation file is located.

The hdf5-file format can be browsed with *vitables* (http://vitables.berlios.de/) or another hdf5-browser available on the net.

3.3.4 output

The simulation result is stored in the file specified in the *output* entry. Only the variables defined at the *entity* level are stored. Temporary (local) variables are not saved. The output file contains values for each period and each field and each item.

Specifying the *path* is optional. If it is omitted, it defaults to the directory where the simulation file is located.

3.3.5 start_period

Defines the first period (integer) to be simulated.

3.3.6 periods

Defines the number of periods (integer) to be simulated.

3.3.7 random_seed

Defines the starting point (integer) of the pseudo-random generator. This section is optional. This can be useful if you want to have several runs of a simulation use the same random numbers.

3.3.8 skip_shows

If set to True, makes all show() functions do nothing. This can speed up simulations which include many shows (usually for debugging).

CHAPTER

FOUR

PROCESSES

The processes are the core of a model. LIAM2 supports two kinds of processes: *assignments*, which change the value of a variable (predictor) using an expression, and *actions* which don't (but have other effects).

For each entity (for example, "household" and "person"), the block of processes starts with the header "processes:". Each process then starts at a new line with an indentation of four spaces.

4.1 Assignments

Assignments have the following general format:

```
processes:
    variable1_name: expression1
    variable2_name: expression2
    ...
```

The variable_name will usually be one of the variables defined in the **fields** block of the entity but, as we will see later, it is not always necessary.

In this case, the name of the process equals the name of the *endogenous variable*. *Process names* have to be **unique** for each entity. See the section about procedures if you need to have several processes which modify the same variable.

To run the processes, they have to be specified in the "processes" section of the simulation block of the file. This explains why the *process names* have to be unique for each entity.

example

```
entities:
    person:
        fields:
            - age: int
        processes:
            age: age + 1
simulation:
    processes:
        - person: [age]
    ...
```

4.2 Temporary variables

All fields declared in the "fields" section of the entity are stored in the output file. Often you need a variable only to store an intermediate result during the computation of another variable.

In LIAM2, you can create a temporary variable at any point in the simulation by simply having an assignment to an undeclared variable. Their value will be discarded at the end of the period.

example

```
person:
    fields:
        # period and id are implicit
        - age:      int
        - agegroup: int

processes:
    age: age + 1
    agediv10: trunc(age / 10)
    agegroup: agediv10 * 10
    agegroup2: agediv10 * 5
```

In this example, *agediv10* and *agegroup2* are temporary variables. In this particular case, we could have bypassed the temporary variable, but when a long expression occurs several times, it is often cleaner and more efficient to express it (and compute it) only once by using a temporary variable.

4.3 Actions

Since actions don't return any value, they do not need a variable to store that result, and they only ever need the condensed form:

```
processes:
    process_name: action expression
    ...
```

example

```
processes:
    remove_deads: remove(dead)
```

4.4 Procedures

A process can consist of sub-processes, in that case we call it a *procedure*. Processes within a procedure are executed in the order they are declared.

Sub-processes each start on a new line, again with an indentation of four spaces and a -.

So the general setup is:

```
processes:
    variable_name: expression
    process_name2: action_expression
    process_name3:
        - subprocess_31: expression
        - subprocess_32: expression
```

In this example, there are three processes, of which the first two do not have sub-processes. The third process is a procedure which consists of two sub-processes. If it is executed, subprocess_31 will be executed and then subprocess_32.

Contrary to normal processes, sub-processes (processes inside procedures) names do not need to be unique. In the above example, it is possible for subprocess_31 and subprocess_32 to have the same name, and hence simulate the same variable. Procedure names (process_name3) does not directly refer to a specific endogenous variable.

example

```
processes:
    ageing:
        - age: age * 2 # in our world, people age strangely
```

```
        - age: age + 1
        - agegroup: trunc(age / 10) * 10
```

The processes on *age* and *agegroup* are grouped in *ageing*. In the simulation block you specify the *ageing*-process if you want to update *age* and *agegroup*.

By using procedures, you can actually make *building blocks* or modules in the model.

4.4.1 Temporary variables

Temporary variables defined/computed within a procedure are local to that procedure: they are only valid within that procedure. If you want to pass variables between procedures you have to define them in the **fields** section.

(bad) example

```
person:
    fields:
        - age: int

    processes:
        ageing:
            - age: age + 1
            - isold: age >= 150    # isold is a local variable

        rejuvenation:
            - age: age - 1
            - backfromoldage: isold and age < 150  # WRONG !
```

In this example, *isold* and *backfromoldage* are local variables. They can only be used in the procedure where they are defined. Because we are trying to use the local variable *isold* in another procedure in this example, LIAM 2 will refuse to run, complaining that *isold* is not defined.

4.4.2 Actions

Actions inside procedures don't even need a process name.

example

```
processes:
    death_procedure:
        - dead: age > 150
        - remove(dead)
```

4.5 Expressions

4.5.1 Deterministic changes

Let us start with a simple increment; the following process increases the value of a variable by one each simulation period.

```
age: age + 1
```

The name of the process is *age* and what it does is increasing the variable *age* of each individual by one, each period.

simple expressions

- Arithmetic operators: +, -, *, /, ** (exponent), % (modulo)

Note that an integer divided by an integer returns a float. For example "1 / 2" will evaluate to 0.5 instead of 0 as in many programming languages. If you are only interested in the integer part of that result (for example, if you know the result has no decimal part), you can use the *trunc* function:

```
agegroup5: 5 * trunc(age / 5)
```

- Comparison operators: <, <=, ==, !=, >=, >
- Boolean operators: and, or, not

Note that you have to use parentheses when you mix *boolean operators* with other operators.

```
inwork: (workstate > 0) and (workstate < 5)
to_give_birth: not gender and (age >= 15) and (age <= 50)
```

- **Conditional expressions:** if(condition, expression_if_true, expression_if_false)

example

```
agegroup_civilstate: if(age < 50,
                        5 * trunc(age / 5),
                        10 * trunc(age / 10))
```

Note that an *if*-statement has always three arguments. If you want to leave a variable unchanged if a condition is not met, specify its value in the *expression_if_false*

```
# retire people (set workstate = 9) when aged 65 or more
workstate: if(age >= 65, 9, workstate)
```

You can nest if-statements. The example below retires men (gender = True) over 64 and women whose age equals at least the parameter/periodic global "WEMRA" (Women Retirement Age).

```
workstate: if(gender,
               if(age >= 65, 9, workstate),
               if(age >= WEMRA, 9, workstate))
```

mathematical functions

- log(expr): natural logarithm (ln)
- exp(expr): exponential
- abs(expr): absolute value
- round(expr[, n]): returns the rounded value of expr to specified n (number of digits after the decimal point). If n is not specified, 0 is used.
- trunc(expr): returns the truncated value (by dropping the decimal part) of expr as an integer.
- clip(x, a, b): returns a if x < a, x if a < x < b, b if x > b.
- min(x, a), max(x, a): the minimum or maximum of x and a.

aggregate functions

- **grpcount([condition]): count the objects in the entity. If filter is given, only** count the ones satisfying the filter.
- grpsum(expr[, filter=condition]): sum the expression
- grpavg(expr[, filter=condition]): average
- grpstd(expr): standard deviation
- grpmax(expr), grpmin(expr): max or min
- grpmedian(expr): median

- grpgini(expr[, filter=condition]): gini

grpsum sums any expression over all the individuals of the current entity. For example *grpsum(earnings)* will produce the sum of the earnings of all persons in the sample. The expression *grpsum(nch0_11)* will result in the total number of children 0 to 11 in the sample.

grpcount counts the number of individuals in the current entity, optionally satisfying a (boolean) condition. For example, *grpcount(gender)* will produce the total number of men in the sample. Contrary to **grpsum**, the grpcount does not require an argument: *grpcount()* will return the total number of individuals in the sample.

Note that, grpsum and grpcount are exactly equivalent if their only argument is a boolean variable (eg. grpcount(ISWIDOW) == grpsum(ISWIDOW)).

example

```
macros:
    WIDOW: civilstate == 5
processes:
    cnt_widows: show(grpcount(WIDOW))
```

link functions

(one2many links)

- countlink(link[, filter])
- sumlink(link, expr[, filter])
- avglink(link, expr[, filter])
- minlink/maxlink(link, expr[, filter])

example

```
entities:
    household:
        fields:
            # period and id are implicit
            - nb_persons: {type: int, initialdata: false}
        links:
            persons: {type: one2many, target: person, field: household_id}

        processes:
            household_composition:
                - nb_persons: countlink(persons)
                - nb_students: countlink(persons, workstate == 1)
                - nch0_11: countlink(persons, age < 12)
                - nch12_15: countlink(persons, (age > 11) and (age < 16))
```

temporal functions

- lag: value at previous period
- value_for_period: value at specific period
- duration: number of consecutive period the expression was True
- tavg: average of an expression since the individual was created
- tsum: sum of an expression since the individual was created

If an item did not exist at that period, the returned value is -1 for a int-field, nan for a float or False for a boolean. You can overide this behaviour when you specify the *missing* parameter.

example

```
lag(age, missing=0) # the age each person had last year, 0 if newborn
grpavg(lag(age))    # average age that the current population had last year
lag(grpavg(age))    # average age of the population of last year

value_for_period(inwork and not male, 2002)

duration(inwork and (earnings > 2000))
duration(educationlevel == 4)

tavg(income)
```

random functions

- uniform: random numbers with a uniform distribution
- normal: random numbers with a normal distribution
- randint: random integers between bounds

example

```
# a random variable with the stdev derived from errsal
normal(loc=0.0, scale=grpstd(errsal))
randint(0, 10)
```

4.5.2 Stochastic changes I: probabilistic simulation

choice

Monte Carlo or probabilistic simulation is a method for iteratively evaluating a deterministic model using sets of random numbers as inputs. In microsimulation, the technique is used to simulate changes of state dependent variables. Take the simplest example: suppose that we have an exogenous probability of an event happening, P(x=1), or not P(x=0). Then draw a random number u from an uniform (0,1) distribution. If, for individual i, ui<p(1), then xi=1. If not, then xi=0. The expected occurrences of x after, say, 100 runs is then P(x=1)*100 and the expected value is 1xP(1)+0xP(0)=P(1). This type of simulation hinges on the confrontation between a random variable and an exogenous probability. In the current version of LIAM 2, it is not possible to combine a choice with alignment.

In LIAM 2, such a probabilistic simulation is called a **choice** process. Suppose i=1..n choice options, each with a probability prob_option_i. The choice process then has the following form:

```
choice([option_1, option_2, ..., option_n],
       [prob_option_1, prob_option_2, ..., prob_option_n])
```

Note that both lists of options and pertaining probabilities are between []'s. Also, the variable containing the options can be of any numeric type.

A simple example of a choice process is the simulation of the gender of newborns (51% males and 49% females), as such:

```
gender=choice([True, False], [0.51, 0.49])
```

The code below illustrates a more complex example of a choice process (called *collar process*). Suppose we want to simulate the work status (collar=1 (blue collar worker), white collar worker) for all working individuals. We however have knowledge one's level of education (education_level=2, 3, 4).

The process *collar_process* has collar as the key endogenous variable and has four sub-processes.

The first sub-process defines a local variable filter-bw, which will be used to separate those that the procedure should apply to. These are all those that do not have a value for collar, and who are working, or who are in education or unemployed, which means that they potentially could work.

The next three "collar" sub-processes simulate whether one is a white or blue collar worker, depending on the level of education. If one meets the above filter_bw and has the lowest educational attainment level, then one has a probability of about 84% (men) and 69% (women) of being a blue collar worker. If one has 'education_level' equal to 3, the probability of being a blue collar worker is of course lower (64% for men and 31% for women), and the probability of becoming a blue collar worker is lowest (8 and 4%, respectively) for those having the highest educational attainment level.

```
collar_process:  # working, in education, unemployed or other inactive
    - filter_bw: (
                   ((workstate > 0) and (workstate < 7))
                   or
                   (workstate == 10)
                  ) and (collar == 0)
    - collar: if(filter_bw and (education_level == 2),
                  if(gender,
                     choice([1, 2], [0.83565, 0.16435]),
                     choice([1, 2], [0.68684, 0.31316]) ),
                  collar)
    - collar: if(filter_bw and (education_level == 3),
                  if(gender,
                     choice([1, 2], [0.6427, 1 - 0.6427]),
                     choice([1, 2], [0.31278, 1 - 0.31278]) ),
                  collar)
    - collar: if(filter_bw and (education_level == 4),
                  if(gender,
                     choice([1, 2], [0.0822, 1 - 0.0822]),
                     choice([1, 2], [0.0386, 1 - 0.0386]) ),
                  collar)
```

4.5.3 Stochastic changes II: behavioural equations

- **Logit:**
  - logit_regr(expr[, filter=None, align='filename'])
  - logit_regr(expr[, filter=None, align=percentage])
- **Alignment :**
  - align(expr[, take=take_filter, leave=leave_filter], fname='filename.csv')
- Continuous (expr + normal(0, 1) * mult + error_var): cont_regr(expr[, filter=None, mult=0.0, error_var=None])
- Clipped continuous (always positive): clip_regr(expr[, filter=None, mult=0.0, error_var=None])
- Log continuous (exponential of continuous): log_regr(expr[, filter=None, mult=0.0, error_var=None])

example

```
divorce: logit_regr(0.6713593 * household.nch12_15
                    - 0.0785202 * dur_in_couple
                    + 0.1429621 * agediff,
                    filter=FEMALE and (civilstate == 2),
                    align='al_p_divorce.csv')

wage_earner: if((age > 15) and (age < 65) and inwork,
                if(MALE,
                   align(wage_earner_score,
                         fname='al_p_wage_earner_m.csv'),
                   align(wage_earner_score,
                         fname='al_p_wage_earner_f.csv')),
                False)
```

logit_regr

Suppose that we have a logit regression that relates the probability of some event to explanatory variables X.

p*i=logit-1(ßX + EPSi)

This probability consists of a deterministic element (as before), completed by a stochastic element, EPSi, a log-normally distributed random variable. The condition for the event occurring is p*i > 0.5.

Instead, suppose that we want the proportional occurrences of the event to be equal to an overall proportion X. In that case, the variable p*i sets the rank of individual i according to the risk that the relevant event will happen. Then only the first X*N individuals in the ranking will experience the event. This process is known as 'alignment'.

In case of one logit with one alignment process -or a logit without alignment-, *logit_regr* will result in the logit returning a Boolean whether the event is simulated. In this case, the setup becomes:

```
- single_align: logit_regr(<logit arguments>,
                           [filter=<filter arguments>,
                           align='name.csv'])
```

example

```
birth:
    - to_give_birth: logit_regr(0.0,
                                filter=FEMALE and
                                       (age >= 15) and (age <= 50),
                                align='al_p_birth.csv')
```

The above generic setup describes the situation where one logit pertains to one alignment process.

logit_score

In many cases, however, it is convenient to use multiple logits with the same alignment process. In this case, using a **logit_score** instead of **logit_regr** will result in the logit returning intermediate scores that - for all conditions together- are the inputs of the alignment process. A typical behavioural equation with alignment has the following syntax:

```
name_process:
    # initialise the score to -1
    - score_variable: -1

    # first condition
    - score_variable: if(condition_1,
                         logit_score(logit_expr_1),
                         score_variable)
    # second condition
    - score_variable: if(condition_2,
                         logit_score(logit_expr_2),
                         score_variable)

    # ... other conditions ...

    # do alignment based on the scores calculated above
    - name_endogenous_variable:
        if(condition,
           if(gender,
              align(score_variable,
                    [take=conditions,]
                    [leave=conditions,]
                    fname='filename_m.csv'),
              align(score_variable,
                    [take=conditions,]
                    [leave=conditions,]
```

```
                    fname='filename_f.csv')),
          False)
```

The equation needs to simulate the variable *name_endogenous_variable*. It starts however by creating a score that reflects the event risk p*i. In a first sub-process, a variable *name_score* is set equal to -1, because this makes it highly unlikely that the event will happen to those not included in the conditions for which the logit is applied. Next, subject to conditions *condition_1* and *condition_2*, this score is simulated on the basis of estimated logits. The specification *logit_score* results in the logit not returning a Boolean but instead a score.

Note that by specifying the endogenous variable *name_score* without any transformations under the 'ELSE' condition makes sure that the score variable is not manipulated by a sub-process it does not pertain to.

align

After this step, the score is known and this is the input for the alignment process. Suppose -as is mostly the case- that alignment data exists for men and women separately. Then the alignment process starts by a *if* to gender. Next comes the align command itself. This takes the form

```
align(score_variable,
      filter=conditions,
      [take=conditions,]
      [leave=conditions,]
      fname='name.csv')
```

The file *name.csv* contains the alignment data. A standard setup is that the file starts with the prefix *al_* followed by the name of the endogenous variable and a suffix *_m* or *_f*, depending on gender.

The optional *take* and *leave* commands forces inclusion or exclusion of objects with specified characteristics in the selection of the event. The individuals with variables specified in the *take* command will a priori be selected for the event. Suppose that the alignment specifies that 10 individuals should experience a certain event, and that there are 3 individuals who meet the conditions specified in the *take*. Then these 3 individuals will be selected a priori and the alignment process will select the remaining 7 candidates from the rest of the sample. The *leave* command works the other way around: those who match the condition in that statement, are a priori excluded from the event happening. The *take* and *leave* are absolute conditions, which mean that the individuals meeting these conditions will always (*take*) or never (*leave*) experience the event.

Their *soft* counterparts can easily be included by manipulating the score of individuals. If this score is set to a strong positive or negative number, then the individual will a priori have a high of low probability of the event happening. These *soft take* and '*soft leave*'s will implement a priority order in the sample of individuals, but will not under all circumstances conditionally include or exclude.

Note that even if the score is -1 an item can be selected by the alignment procedure. This happens when there are not enough candidates (selected by the score) to meet the alignment needs.

The below application describes the process of being (or remaining) a wage-earner or employee. It illustrates a *soft leave* by setting the a priori score variable *wage_earner_score* to -1. This makes sure that the a priori selection probability for those not specified in the process is very low (but not zero, as in the case of *leave* conditions).

Next come three sub processes setting a couple of common conditions, in the form of local (temporary) variables. These three sub- processes are followed by six subsequent *if* conditions, separating the various behavioural equations to the sub-sample they pertain to. The first three sub conditions pertain to women and describe the probability of being a wage-earner from in work and employee previous year (1) from in work but not employee previous year (2), and from not in work previous year (3). The conditions 4 to 6 describe the same transitions but for women.

```
wage_earner_process:
    - wage_earner_score: -1
    - lag_public: lag((workstate == 2) or (workstate == 3))
    - inwork: (workstate > 0) and (workstate < 5)
    - lag_inwork: lag((workstate > 0) and (workstate < 5))
    - men_inwork: gender and (age > 15) and (age < 65) and inwork
```

```
    # === MEN ===
    # Probability of being employee from in work and employee previous year
    - wage_earner_score:
        if(men_inwork and ((lag(workstate) == 1) or (lag(workstate) == 2)),
           logit_score(0.0346714 * age + 0.9037688 * (collar == 1)
                       - 0.2366162 * (civilstate == 3) + 2.110479),
           wage_earner_score)
    # Probability of becoming employee from in work but not employee
    # previous year
    - wage_earner_score:
        if(men_inwork and ((lag(workstate) != 1) and (lag(workstate) != 2)),
           logit_score(-0.1846511 * age - 0.001445 * age **2
                       + 0.4045586 * (collar == 1) + 0.913027),
           wage_earner_score)
    # Probability of becoming employee from not in work previous year
    - wage_earner_score:
        if(men_inwork and (lag(workstate) > 4),
           logit_score(-0.0485428 * age + 1.1236 * (collar == 1) + 2.761359),
           wage_earner_score)

    # === WOMEN ===
    - women_inwork: not gender and (age > 15) and (age < 65) and inwork

    # Probability of being employee from in work and employee previous year
    - wage_earner_score:
        if(women_inwork and ((lag(workstate) == 1) or (lag(workstate) == 2)),
           logit_score(-1.179012 * age + 0.0305389 * age **2
                       - 0.0002454 * age **3
                       - 0.3585987 * (collar == 1) + 17.91888),
           wage_earner_score)
    # Probability of becoming employee from in work but not employee
    # previous year
    - wage_earner_score:
        if(women_inwork and ((lag(workstate) != 1) and (lag(workstate) != 2)),
           logit_score(-0.8362935 * age + 0.0189809 * age **2
                       - 0.000152 * age **3 - 0.6167602 * (collar == 1)
                       + 0.6092558 * (civilstate == 3) + 9.152145),
           wage_earner_score)
    # Probability of becoming employee from not in work previous year
    - wage_earner_score:
        if(women_inwork and (lag(workstate) > 4),
           logit_score(-0.6177936 * age + 0.0170716 * age **2
                       - 0.0001582 * age**3 + 9.388913),
           wage_earner_score)

    - wage_earner: if((age > 15) and (age < 65) and inwork,
                      if(gender,
                         align(wage_earner_score,
                               fname='al_p_wage_earner_m.csv'),
                         align(wage_earner_score,
                               fname='al_p_wage_earner_f.csv')),
                      False)
```

The last sub-procedure describes the alignment process. Alignment is applied to individuals between the age of 15 and 65 who are in work. The reason for this is that those who are not working obviously cannot be working as a wage-earner. The input- files of the alignment process are 'al_p_wage_earner_m.csv' and 'al_p_wage_earner_f.csv'. The alignment process sets the Boolean *wage earner*, and uses as input the scores simulated previously, and the information it takes from the alignment files. No 'take' or 'leave' conditions are specified in this case.

Note that the population to align is the population specified in the first condition, here *(age>15) and (age<65) and (inwork)* and not the whole population.

4.5.4 Lifecycle functions

new

new creates items initiated from another item of the same entity (eg. a women gives birth) or another entity (eg. a marriage creates a new houshold).

generic format

```
new('entity_name', filter=expr,
    *set initial values of a selection of variables*)
```

The first parameter defines the entity in which the item will be created (eg person, household, ...).

Then, the filter argument specifies which items of the current entity will serve as the origin for the new items (for persons, that would translate to who is giving birth, but the function can of course be used for any kind of entity).

Any subsequent argument specifies values for fields of the new individuals. Any field which is not specified there will receive the missing value corresponding to the type of the field ('nan' for floats, -1 for integers and False for booleans). Those extra arguments can be given constants, but also any expression (possibly using links, random functions, ...). Those expressions are evaluated in the context of the origin individuals. For example, you could write "mother_age = age", which would set the field "mother_age" on the new item to the age of their mother.

example 1

```
birth:
    - to_give_birth: logit_regr(0.0,
                                filter=not gender and
                                       (age >= 15) and (age <= 50),
                                align='al_p_birth.csv')
    - new('person', filter=to_give_birth,
          mother_id = id,
          father_id = partner.id,
          household_id = household_id,
          partner_id = -1,
          age = 0,
          civilstate = 1,
          collar = 0,
          education_level = -1,
          workstate = 5,
          gender=choice([True, False], [0.51, 0.49]) )
```

The first sub-process (*to_give_birth*) is a logit regression over women (not gender) between 15 and 50 which returns a boolean value whether that person should give birth or not. The logit itself does not have a deterministic part (0.0), which means that the 'fertility rank' of women that meet the above condition, is only determined by a logistic stochastic variable). This process is also aligned on the data in 'al_p_birth.csv'.

In the above case, a new person is created for each time a woman is scheduled to give birth. Secondly, a number of links are established: the value for the *mother_id* field of the child is set to the id-number of his/her mother, the child receives the household number of his/her mother, the child's father is set to the partner of the mother, ... Finally some variables of the child are set to specific initial values: the most important of these is its gender, which is the result of a simple choice process.

new is not limited to items of the same entity; the below procedure *get a life* makes sure that all those who are single when they are 24 year old, leave their parents' household for their own household. The region of this household is created through a simple choice-process.

example 2

```
get_a_life:
    - household_id:
        if((age == 24) and (civilstate != 2) and (civilstate != 3),
           new('household',
               start_period=period,
               region_id=choice([0, 1, 2, 3], [0.1, 0.2, 0.3, 0.4])
```

```
            ),
            household_id)
```

clone

clone is very similar to **new** but is intended for cases where most or all variables describing the new individual should be copied from his/its parent/origin instead of being set to "missing". With clone, you cannot specify what kind of entity you want to create, as it is always the same as the origin item. However, similarly to **new**, **clone** also allows fields to be specified manually by any expression evaluated on the parent/origin.

Put differently, a **new** with no fields mentioned will result in a new item of which the initial values of the fields are all set to missing and have to be filled through simulation; on the contrary, a **clone** with no fields mentioned will result in a new item that is an exact copy of the origin except for its id number which is always set automatically.

example

```
make_twins:
    - clone(filter=new_born and is_twin,
            gender=choice([True, False], [0.51, 0.49]))
```

remove

remove removes items from an entity dataset. With this command you can remove obsolete items (eg. dead persons, empty households) thereby ensuring they are not simulated anymore. This will also save some memory and, in some cases, improve simulation speed.

The procedure below simulates whether an individual survives or not, and what happens in the latter case.

```
dead_procedure:
    # decide who dies
    - dead: if(gender,
               logit_regr(0.0, align='al_p_dead_m.csv'),
               logit_regr(0.0, align='al_p_dead_f.csv'))
    # change the civilstate of the suriving partner
    - civilstate: if(partner.dead, 5, civilstate)
    # break the link to the dead partner
    - partner_id: if(partner.dead, -1, partner_id)
    # remove the dead
    - remove(dead)
```

The first sub-procedure *dead* simulates whether an individual is 'scheduled for death', using again only a logistic stochastic variable and the age-gender-specific alignment process. Next some links are updated for the surviving partner. The sub-procedure *civilstate* puts the variable of that name equal to 5 (which means that one is a widow(er) for those individuals whose partner has been scheduled for death. Also, in that case, the partner identification code is erased. All other procedures describing the heritage process should be included here. Finally, the *remove* command is called to removes the *dead* from the simulation dataset.

4.5.5 Matching functions

matching: (aka Marriage market) matches individuals from set 1 with individuals from set 2. For each individual in set 1 following a particular order (given by the expression in the *orderby* argument), the function computes the score of all (unmatched) individuals in set 2 and take the best scoring one.

You have to specify the boolean filters which provide the two sets to match (set1filter and set2filter), the criterion to decide in which order the individuals of the first set are matched and the expression that will be used to assign a score to each individual of the second set (given a particular individual in set 1).

In the score expression the fields of the set 1 individual can be used normally and the fields of its possible partners can be used by prefixing them by "**other.**".

generic setup

```
matching(set1filter=boolean_expr,
         set2filter=boolean_expr,
         orderby=difficult_match,
         score=coef1 * field1 + coef2 * other.field2 + ...)
```

The generic setup of the marriage market is simple; one needs to have selected those individuals who are to be coupled (*to_couple*=true). Furthermore, one needs to have a variable (*difficult_match*) which can be used to rank individuals according how easy they are to match. Finally, we need a function (*score*) matching potential partners.

In the first step, and for those persons that are selected to be coupled, potential partners are matched in the order set by *difficult_match* and each woman is matched with the potential partner with the highest matching score. Once this is done, both individuals become actual partners and the partner identification numbers are set so that the partner number of each person equals the identification number of the partner.

example

```
marriage:
    - in_couple: MARRIED or COHAB
    - to_couple: if((age >= 18) and (age <= 90) and not in_couple,
                    if(MALE,
                       logit_regr(0.0, align='al_p_mmkt_m.csv'),
                       logit_regr(0.0, align='al_p_mmkt_f.csv')),
                    False)
    - avg_age_males_to_couple: grpavg(age, filter=to_couple and MALE)
    - difficult_match: if(to_couple and FEMALE,
                          abs(age - avg_age_males_to_couple),
                          nan)
    - work: (workstate > 0) and (workstate < 5)
    - partner_id: if(to_couple,
                     matching(set1filter=FEMALE, set2filter=MALE,
                              orderby=difficult_match,
                              score=- 0.4893 * other.age
                                    + 0.0131 * other.age ** 2
                                    - 0.0001 * other.age ** 3
                                    + 0.0467 * (other.age - age)
                                    - 0.0189 * (other.age - age) ** 2
                                    + 0.0003 * (other.age - age) ** 3
                                    - 0.9087 * (other.work and not work)
                                    - 1.3286 * (not other.work and work)
                                    - 0.6549 * (other.work and work)),
                     partner_id)
    - coupled: to_couple and (partner_id != -1)
    - newhousehold: new('household', filter=coupled and FEMALE,
                        start_period=period,
                        region_id=choice([0, 1, 2, 3],
                                         [0.1, 0.2, 0.3, 0.4]))
    - household_id: if(coupled,
                       if(MALE, partner.newhousehold, newhousehold),
                       household_id)
```

The code above shows an application. First of all, individuals eligible for marriage are all those between 18 and 90 who are not a part of a couple; the actual decision who is eligible is left to the alignment process. Next, for every women eligible to coupling, the variable *difficult_match* is the difference between her age and the average age of men eligible for coupling.

In a third step, for each eligible woman in turn (following the order set by *difficult_match*), all eligited men are assigned a score and the man with the best score is matched with that woman. This score depends on his age, his difference in age with the woman and the the work status of the potential partners.

In a next step, a new household is created for women who have just become a part of a couple. Their household number, as well as their new partners is then updated to reflect their new household.

4.6 Output

LIAM 2 produces simulation output in three ways. First of all, by default, the simulated datasets are stored in hdf5 format. These can be accessed at the end of the run. You can use several tools to inspect the data.

You can display information during the simulation using *show* or *groupby*. You can *dump* data to csv-file for further study.

If you run LIAM 2 in interactive mode, you can type in output functions in the console to inspect the data.

4.6.1 show

show evaluates expressions and prints the result to the console.

```
show(expr1[, expr2, expr3, ...])
```

example 1

```
show(grpcount(age >= 18))
show(grpcount(not dead), grpavg(age, filter=not dead))
```

The first process will print out the number of persons of age 18 and older in the dataset. The second one displays the number of living people and their average age.

example 2

```
show("Count:", grpcount(),
     "Average age:", grpavg(age),
     "Age std dev:", grpstd(age))

gives

Count: 19944 Average age: 42.7496991576 Age std dev: 21.9815913417
```

Note that you can use the special character "n" to display the rest of the result on the next line.

example 3

```
show("Count:", grpcount(),
     "\nAverage age:", grpavg(age),
     "\nAge std dev:", grpstd(age))

gives

Count: 19944
Average age: 42.7496991576
Age std dev: 21.9815913417
```

4.6.2 csv

The **csv** function writes values to a csv-file.

csv(expr1[, expr2, expr3, ..., suffix='file_suffix', fname='filename', mode='w'])

The suffix, fname and mode are optional arguments.

- 'fname' allows defining the exact file names used. You can optionally use {entity} and {period} to customize the name.
- 'suffix' allows to set the name of csv file more easily. If suffix is used, the filename will be: "{entity}_{period}_{suffix}.csv"

The default file name (if neither 'fname' nor 'suffix' is used) is "{entity}_{period}.csv".

example

```
csv(grpavg(income), suffix='income')
```

will create one file for each simulated period. Assuming, start_period is 2002 and periods is 2, it will create two files: "person_2002_income.csv" and "person_2003_income.csv" with the average income of the population for period 2002 and 2003 respectively.

- 'mode' allows appending (mode='a') to a csv file instead of overwriting it (mode='w' by default). This allows you, for example, to store the value of some expression for all periods in the same file (instead of one file per period by default).

example

```
csv(period, grpavg(income), fname='avg_income.csv', mode='a')
```

Note that unless you erase/overwrite the file one way or another between two runs of a simulation, you will append the data of the current simulation to that of the previous one. One way to do that automatically is to have a procedure in the init section without mode='a' to overwrite the file.

If you want that file to start empty, you can do so this way:

```
csv(fname='avg_income.csv')
```

If you want some headers in your file, you could write them at that point:

```
csv('period', 'average income', fname='avg_income.csv')
```

When you use the csv() function in combination with (at least one) table expressions (see dump and groupby functions below), the results are appended below each other.

csv(table_expr1, 'and here goes another table', table_expr2, fname='tables.csv')

Will produce a file with a layout like this:

```
| table 1 value at row 1, col 1 | col 2 |   ... | col N |
|                           ... |   ... |   ... |   ... |
|                   row N, col 1 | col 2 |   ... | col N |
| and here goes another table   |       |       |       |
| table 2 value at row 1, col 1 |   ... | col N |       |
|                           ... |   ... |   ... |       |
|                   row N, col 1 |   ... | col N |       |
```

You can also output several rows with a single command by enclosing values between brackets:

```
csv([row1value1, ..., row1valueN],
    ...
    [rowNvalue1, ..., rowNvalueN],
    fname='several_rows.csv')
```

example

```
csv(['this is', 'a header'],
    ['with', 'several lines'],
    fname='person_age_aggregates.csv')
```

Will produce a file with a layout like this:

```
| this is | a header      |
| with    | several lines |
```

4.6.3 dump

dump produces a table with the expressions given as argument evaluated over many (possibly all) individuals of the dataset.

general format

dump([expr1, expr2, ..., filter=filterexpression, missing=value, header=True])

If no expression is given, *all* fields of the current entity will be dumped (including temporary variables available at that point), otherwise, each expression will be evaluated on the objects which satisfy the filter and produce a table.

The 'filter' argument allows to evaluate the expressions only on the individuals which satisfy the filter. Defaults to None (evaluate on all individuals).

The 'missing' argument can be used to transform 'nan' values to another value. Defaults to None (no transformation).

The 'header' argument determine whether column names should be in the dump or not. Defaults to True.

example

```
show(dump(age, partner.age, gender, filter=id < 10))
```

gives

```
id | age | partner.age | gender
 0 |  27 |          -1 |  False
 1 |  86 |          71 |  False
 2 |  16 |          -1 |   True
 3 |  19 |          -1 |  False
 4 |  27 |          21 |  False
 5 |  89 |          92 |   True
 6 |  59 |          61 |   True
 7 |  65 |          29 |  False
 8 |  38 |          35 |   True
 9 |  48 |          52 |   True
```

4.6.4 groupby

groupby (aka *pivot table*): group all individuals by their value for the given expressions, and optionally compute an expression for each group. If no expression is given, it will compute the number of individuals in that group. A filter can be specified to limit the individuals taken into account.

general format

```
groupby(expr1[, expr2, expr3, ...] [, expr=expression]
        [, filter=filterexpression] [, percent=True])
```

example

```
show(groupby(age / 10, gender))
```

gives

```
    gender | False | True |
(age / 10) |       |      | total
         0 |   818 |  803 |  1621
         1 |   800 |  800 |  1600
         2 |  1199 | 1197 |  2396
         3 |  1598 | 1598 |  3196
         4 |  1697 | 1696 |  3393
         5 |  1496 | 1491 |  2987
         6 |  1191 | 1182 |  2373
```

```
        7 |   684 |  671 |  1355
        8 |   369 |  357 |   726
        9 |   150 |  147 |   297
    total | 10002 | 9942 | 19944
```

example

```
show(groupby(inwork, gender))
```

gives

```
gender | False | True |
inwork |       |      | total
 False |  6170 | 5587 | 11757
  True |  3832 | 4355 |  8187
 total | 10002 | 9942 | 19944
```

example

```
show(groupby(inwork, gender, percent=True))
```

gives

```
gender | False |  True |
inwork |       |       |  total
 False | 30.94 | 28.01 |  58.95
  True | 19.21 | 21.84 |  41.05
 total | 50.15 | 49.85 | 100.00
```

example

```
groupby(workstate, gender, expr=grpavg(age))
```

gives the average age by workstate and gender

```
   gender | False |  True |
workstate |       |       | total
        1 | 41.29 | 40.53 | 40.88
        2 | 40.28 | 44.51 | 41.88
        3 |  8.32 |  7.70 |  8.02
        4 | 72.48 | 72.27 | 72.38
        5 | 42.35 | 46.56 | 43.48
    total | 42.67 | 42.38 | 42.53
```

4.7 Interactive console

LIAM 2 features an interactive console which allows you to interactively explore the state of the memory either during or after a simulation completed.

You can reach it in two ways. You can either pass "-i" as the last argument when running the executable, in which case the interactive console will launch after the whole simulation is over. The alternative is to use breakpoints in your simulation to interrupt the simulation at a specific point (see below).

Type "help" in the console for the list of available commands. In addition to those commands, you can type any expression that is allowed in the simulation file and have the result directly. Show is implicit for all operations.

examples

```
>>> grpavg(age)
53.7131819615

>>> groupby(age / 20, gender, expr=grpcount(inwork))
```

```
    gender | False | True |
(age / 20) |       |      | total
         0 |    14 |   18 |    32
         1 |   317 |  496 |   813
         2 |   318 |  258 |   576
         3 |    40 |  102 |   142
         4 |     0 |    0 |     0
         5 |     0 |    0 |     0
     total |   689 |  874 |  1563
```

4.7.1 breakpoint

breakpoint: temporarily stops execution of the simulation and launch the interactive console. There are two additional commands available in the interactive console when you reach it through a breakpoint: "step" to execute (only) the next process and "resume" to resume normal execution.

general format

breakpoint([period])

the "period" argument is optional and if given, will make the breakpoint interrupt the simulation only for that period.

example

```
marriage:
    - in_couple: MARRIED or COHAB
    - breakpoint(2002)
    - ...
```

CHAPTER

FIVE

LINKS

Entities can be linked with each other or with other entities, for example, individuals belong to households, and mothers are linked to their children, while partners are interlinked as well.

A typical link has the following form:

```
name: {type: <type>, target: <entity>, field: <name of link field>}
```

LIAM 2 uses integer fields to establish the link between entities. Those integer fields contain the id-number of the linked individual.

LIAM 2 allows two types of links: many2one and one2many.

5.1 many2one

A **many2one** link the item of the entity to *one* other item in the same (eg. a person to its mother) or another entity (eg. a person to its household).

This allows the modeller to use information stored in the linked entities.

```
entities:
    person:
        fields:
            - age: int
            - income: float
            - mother_id: int
            - father_id: int
            - partner_id: int

        links:
            mother: {type: many2one, target: person, field: mother_id}
            father: {type: many2one, target: person, field: father_id}
            partner: {type: many2one, target: person, field: partner_id}

        processes:
            age: age + 1
            mother_age: mother.age
            parents_income: mother.income + father.income
```

To access a field of a linked individual (possibly of the same entity), you use:

```
link_name.field_name
```

For example, the *mother_age* process uses the 'mother' link to assign the age of the mother to the *mother_age* field. If an individual's link does not point to anything (eg. a person has no known mother), trying to use the link would yield the missing value (eg. for orphans, mother.age is -1 and parents_income is *nan*).

As another example, the process below sets a variable *to_separate* to *True* if the variable *separate* is True for either the individual or his/her partner.

```
- to_separate: separate or partner.separate
```

Note that it is perfectly valid to chain links as, for example, in:

```
grand_parents_income: mother.mother.income + mother.father.income +
                      father.mother.income + father.father.income
```

Another option to get values in the linked individual is to use the form:

```
link_name.get(expr)
```

this syntax is a bit more verbose in the simple case, but is much more powerful as it allows to evaluate (almost) any expression on the linked individual.

For example, if you want to get the average age of both parents of the mother of each individual, you can do it so:

    mother.get((mother.age + father.age) / 2)

5.2 one2many

A **one2many** links an item in an entity to at least one other item in the same (eg. a person to its children) or other entity (a household to its members).

```
entities:
    household:
        links:
            persons: {type: one2many, target: person, field: household_id}

    person:
        fields:
            - age: int
            - income: float
            - household_id : int

        links:
            household: {type: many2one, target: household, field: household_id}
```

- *persons* is the link from the household to its members.
- *household* is the link form a person to the household.

To use information stored in the linked entities you have to use *aggregate functions*

- countlink (eg. countlink(persons) gives the numbers of persons in the household)
- sumlink (eg. sumlink(persons, income) sums up all incomes from the members in a household)
- avglink (eg. avglink(persons, age) gives the average age of the members in a household)
- minlink, maxlink (eg. minlink(persons, age) gives the age of the youngest member of the household)

example

```
entities:
    household:
        fields:
            - num_children: int

        links:
            # link from a household to its members
            persons: {type: one2many, target: person, field: household_id}

        processes:
            num_children: countlink(persons, age < 18)
```

```
    person:
        fields:
            - age: int
            - household_id: int

        links:
            # link form a person to his/her household
            household: {type: many2one, target: household,
                        field: household_id}

        processes:
            num_kids_in_hh: household.num_children
```

The num_children process, once called will compute the number of persons aged 17 or less in each household and store the result in the *num_children* field (of the **household**). Afterwards, that variable can be used like any other variable, for example through a many2one link, like in the *num_kids_in_hh* process. This process computes for each **person**, the number of children in the household of that person.

Note that the variable *num_kids_in_hh* could also have been simulated by just one process, on the "person" level, by using:

```
- num_kids_in_hh: household.get(countlink(persons, age < 18))
```

CHAPTER

SIX

IMPORTING DATA

6.1 data files

As of now, you can only import CSV files, one file for each entity. Their first row should contain the name of the fields. You need at least two integer columns: “id” and “period” (though they do not necessarily need to be named like that in the csv file).

6.2 description file

To import CSV files, you need to create a description file. Those description files have the following general format:

```
output: <path_of_hdf5_file>.csv

# compression is optional. compression type can be 'zlib', 'bzip2' or 'lzo'
# level is a digit from 1 to 9 and is optional (defaults to 5).
# Examples of valid compression strings are: zlib, lzo-1, bzip2-9.
# You should experiment to see which compression scheme (if any) offers the
# best trade-off for your dataset.
compression: <type>-<level>

globals:
    periodic:
        path: <path_of_globals_file>.csv
        # if the csv file is transposed (each field is on a row instead of a
        # column and the field names are in the first column, instead of the
        # first row), you can use "transpose: true". You do not need to
        # specify anything if the file is not transposed.
        transposed: true

entities:
    <entity1_name>:
        path: <path_to_entity1_data>.csv

        # if you want to manually select the fields to be used, and/or
        # specify their types, you can do so in the following section.
        # If you want to use all the fields present in the csv file, you
        # can simply omit this section. The field types will be
        # automatically detected.
        fields:
            # period and id are implicit
            - <field1_name>: <field1_type>
            - <field2_name>: <field2_type>
            - ...

        # if you want to keep your csv files intact but use different
```

```
        # names in your simulation than in the csv files, you can specify
        # name changes here.
        oldnames:
            <fieldX_newname>: <fieldX_oldname>
            <fieldY_newname>: <fieldY_oldname>

        # if you want to invert the value of some boolean fields
        # (True -> False and False -> True), add them to the "invert" list
        # below.
        invert: [list, of, boolean, fields, to, invert]

    <entity2_name>:
        ...
```

Most elements of this description file are optional. The only required elements are "output" and "entities". If an element is not specified, it uses the following default value:

- if *path* is omitted, it defaults to a file named after the entity in the same directory than the description file (ie *local_path\name_of_the_entity.csv*).
- if the *fields* section is omitted, all columns of the csv file will be imported and their type will be detected automatically.
- if *compression* is omitted, the output will not be compressed.

Note that if an "entity section" is entirely empty, you need to use the special code: "{}".

example

```
output: normal.h5

globals:
    periodic:
        path: input\globals_transposed.csv
        transposed: true

entities:
    household:
        path: input\household.csv

    person:
        path: input\person.csv
        fields:
            - age:           int
            - gender:        bool
            - workstate:     int
            - civilstate:    int
            - partner_id:    int

        oldnames:
            gender: male
```

simpler example

```
output: simple.h5

globals:
    periodic:
        path: input\globals.csv

entities:
    household:
        path: input\household.csv
```

```
    person:
        path: input\person.csv
```

simplest example

```
output: simplest.h5

entities:
    household: {}
    person: {}
```

This will try to load all the fields of the household and person entities in "*household.csv*" and "person.csv" in the same directory than the description file.

6.3 importing the data

Once you have your data as CSV files and created a description file, you can import your data.

- If you are using the bundled editor, simply open the description file and press F5.
- If you are using the command line, use:

```
    liam2 import <path_to_description_file>
```

CHAPTER

SEVEN

INDICES AND TABLES

- *genindex*
- *search*

CHAPTER

EIGHT

APPENDIX

8.1 Technical choices

8.1.1 Python

We use the Python language (http://www.python.org/) for the development of LIAM 2.

> Python runs on Windows, Linux/Unix, Mac OS X, and has been ported to the Java and .NET virtual machines.
>
> Python is free to use, even for commercial products, because of its OSI-approved open source license.

8.1.2 HDF5

We store the used data in an hdf5-format (http://www.hdfgroup.org).

> HDF5 is a data model, library, and file format for storing and managing data. It supports an unlimited variety of data types, and is designed for flexible and efficient I/O and for high volume and complex data. HDF5 is portable and is extensible, allowing applications to evolve in their use of HDF5. The HDF5 Technology suite includes tools and applications for managing, manipulating, viewing, and analyzing data in the HDF5 format.
>
> HDF is open-source and the software is distributed at no cost. Potential users can evaluate HDF without any financial investment. Projects that adopt HDF are assured that the technology they rely on to manage their data is not dependent upon a proprietary format and binary-only software that a company may dramatically increase the price of, or decide to stop supporting altogether.

This allows us to handle important data sets.

8.1.3 YAML

The definition of the data and the model is done in the YAML-language (http://www.yaml.org).

> YAML: YAML Ain't Markup Language
>
> What It Is: YAML is a human friendly data serialization standard for all programming languages.

8.2 Change log

8.2.1 Version 0.4.1

Released on 2011-12-02.

Miscellaneous improvements:

- validate both import and simulation files, i.e. detect bad structure and invalid and missing keywords.
- improved error messages (both during import and the simulation), by stripping any information that is not useful to the user. For some messages, we only have a line number and column left, this is not ideal but should be better than before. The technical details are written to a file (error.log) instead.
- improved "incoherent alignment data" error message when loading an alignment file by changing the wording and adding the path of the file with the error.
- reorganised bundle files so that there is no confusion between directories for Notepad++ and those of liam2.
- tweaked Notepad++ configuration:
  - added explore command as F7
  - removed more unnecessary features.

Fixes:

- disallowed using one2many links like many2one (it was never intended this was and produced wrong results).
- fixed groupby with a scalar expression (it does not make much sense, but it is better to return the result than to fail).
- re-enabled the code to show the expressions containing errors where possible (in addition to the error message). This was accidentally removed in a previous version).
- fixed usage to include the 'explore' command.

8.2.2 Version 0.4.0

Released on 2011-11-25.

New features:

- added grpgini function.
- added grpmedian function.
- implemented filter argument in grpsum().
- implemented N-dimensional alignment (alignment can be done on more than two variables/dimensions in the same file).
- added keyword arguments to csv():
  - 'fname' to allow defining the exact name of the csv file.
  - 'mode' to allow appending to a csv file instead of overwriting it.
- reworked csv() function to support several arguments, like show. It also supports non-table arguments.
- added 'skip_shows' simulation option, to make all show() functions do nothing.
- allowed expressions in addition to variable names in alignment files.
- added keyword arguments to dump():
  - 'missing' to convert nans into the given value.
  - 'header' to determine whether column names should be in the dump or not.
- improved import functionality:

  - compression is now configurable.
  - any csv file can be transposed, not just globals.
  - globals fields can be selected, renamed and inverted like in normal entities.
- added "explore" command to the main executable, to launch the interactive console on a completed simulation without re-simulating it.

Miscellaneous improvements:

- expressions do not need to be quoted anymore.
- reverted init to old semantic: it happens in "start_period - 1", so that lag(variable_set_in_init) works even for the first period.
- purge all local variables after each process to lower memory usage.
- allowed the result of new() to not be stored in a variable.
- allowed using temporary variables in matching() function.
- added a tolerance of 1e-6 to the sum of choice's probabilities to be equal 1.0
- added explicit message about alignment over and underflows.
- nicer display for small (< 5ms) and large (>= 1 hour) timings.
- improved error message on missing parenthesis around operands of boolean operators.
- improved error message on duplicate fields.
- improved error message when a variable which is not computed yet is used.
- added more information to the console log:
  - number of individuals at the start and end of each period.
  - more stats at the end of the simulation.
- excluded unused components in the executable to make it smaller.

Fixes:

- fixed logit_regr(align=float).
- fixed grpavg(bool, filter=cond).
- fixed groupby(a, b, c, expr=grpsum(d), percent=True).
- fixed having several grpavg with a filter argument in the same expression.
- fixed calling the main executable without argument (simply display usage).
- fixed dump with (some kind of) aggregate values in combination with a filter.
- fixed void data source.

8.2.3 Version 0.3.0

Released on 2011-06-29.

New features:

- added ability to import csv files directly with the main executable.

Miscellaneous improvements:

- made periodic globals optional.
- improved a few sections of the documentation.

Fixes:

- fixed non-assignment "actions" in interactive console (csv, remove, ...).
- fixed error_var argument to cont_regr, clip_regr and log_regr.

8.2.4 Version 0.2.1

Released on 2011-06-20.

Miscellaneous improvements:

- simplified and cleaned up the demonstration models.
- improved the error message when a link points to an unknown entity.
- the evaluator creates fewer internal temporary variables in some cases.

Fixes:

- added log and exp to the list of available functions (they were already implemented but not usable because of that).
- fixed log_regr, cont_regr and clip_regr which were comparing their result with 0.5 (like logit_regr when there is no alignment).
- fixed new() function, which created individuals correctly but in some cases returned values which did not correspond to the ids of the newly created individuals, due to a bug in numpy.

8.2.5 Version 0.2

Released on 2011-06-07.

New features:

- added support for retrospective simulation (ie simulating periods for which we already have some data): at the start of each simulated period, if there is any data in the input file for that period, it is "merged" with the result of the last simulated period. If there is any conflict, the data in the input file has priority.
- added "clone" function which creates new individuals by copying all fields from their "origin" individuals, except for the fields which are given a value manually.
- added breakpoint function, which launches the interactive console during a simulation. Two more console commands are available in that mode:
  - "s(tep)" to execute the next process
  - "r(esume)" to resume normal execution

  The breakpoint function takes an optional period argument so that it triggers only for that specific period.
- added "tsum" function, which sums an expression over the whole lifetime of individuals. It returns an integer when summing integer or boolean expressions, and a float for float expressions.

- implemented using the value of a periodic global at a specific period. That period can be either a constant (eg "MINR[2005]") or an expression (eg "MINR[period - 10]" or "MINR[year_of_birth + 20]")
- added "trunc" function which takes a float expression and returns an int (dropping everything after the decimal point)

Miscellaneous improvements:

- made integer division (int / int) return floats. eg 1/2 = 0.5 instead of 0.
- processes which do not return any value (csv and show) do not need to be named anymore when they are inside of a procedure.
- the array used to run the first period is constructed by merging the individuals present in all previous periods.
- print timing for sub-processes in procedures. This is quite verbose but makes debugging performance problems/regressions easier.
- made error messages more understandable in some cases.
- manually flush the "console" output every time we write to it, not only within the interactive console, as some environments (namely when using the notepad++ bundle) do not flush the buffer themselves.
- disable compression of the output/simulation file, as it hurts performance quite a bit (the simulation time can be increased by more than 60%). Previously, it was using the same compression settings as the input file.
- allowed align() to work on a constant. eg:

```
align(0.0, fname='al_p_dead_m.csv')
```

- made the "tavg" function work with boolean and float expressions in addition to integer expressions
- allowed links to be used in expression given in the "new" function to initialise the fields of the new individuals.
- using "__parent__" in the new() function is no longer necessary.
- made the "init" section optional (it was never intended to be mandatory).
- added progress bar for copying table.
- optimised some parts for speed, making the whole simulation roughly as fast as 0.1 even though more work is done.

Fixes:

- fixed "tavg" function:
  - the result was wrong because the number of values (used in the division) was one less than it should.
  - it yielded "random" values when some individuals were present in a past period, but not in the current period.
- fixed "duration" function:
  - it crashed when a past period contained no individuals.
  - it yielded "random" values when some individuals were present in a past period, but not in the current period.
- fixed "many2one" links returning seemingly random values instead of "missing" when they were pointing to an individual which was not present anymore (usually because the individual was dead).
- fixed min/max functions.
- fields which are not given an explicit value in new() are initialised to missing, instead of 0.

- the result of the new() function (which returns the id of the newly created individuals) is now -1 (instead of 0) for parents which are not in the filter.
- fixed some expressions crashing when used within a lag.
- fixed the progress bar to display correctly even when there are only very few iterations.

8.2.6 Version 0.1

First semi-public release, released on 2011-02-24.

INDEX