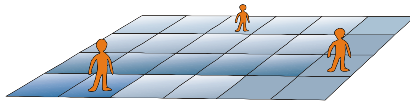
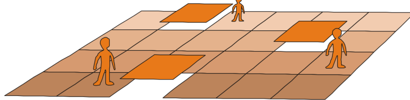
Mathematical Programming-based Multi-Agent Systems (MPMAS)

MpmasQL 2.52

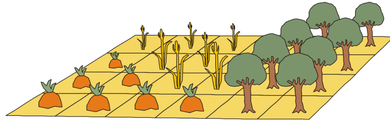
Layer 1
Human actors/
Communication
networks



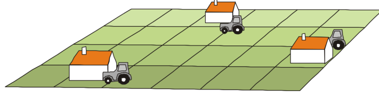
Layer 2
Land and
water markets



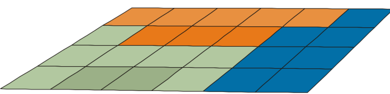
Layer 3
Landuse/
cover



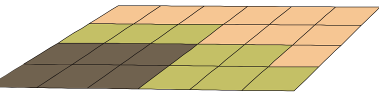
Layer 4
Farmsteads



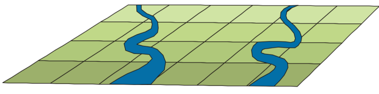
Layer 5
Ownership

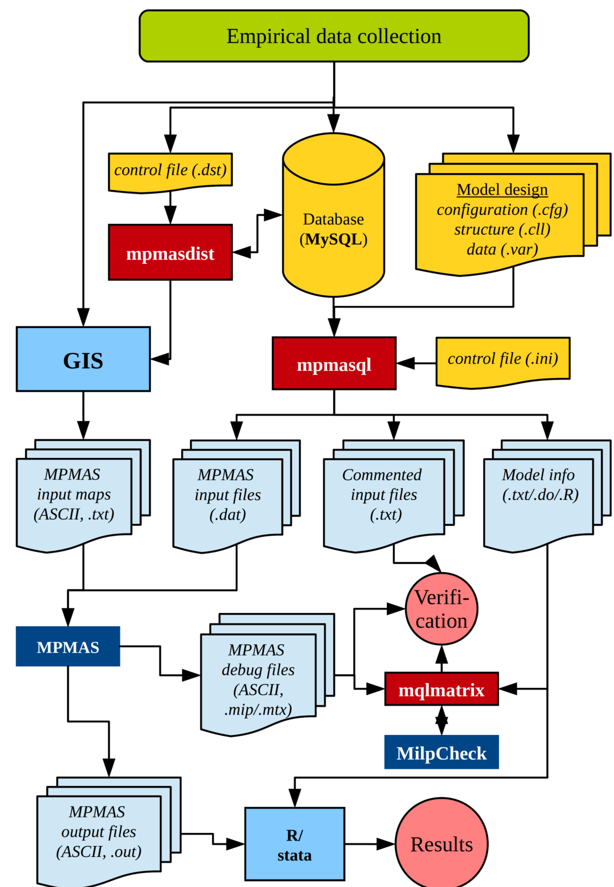


Layer 6
Soil quality



Layer 7
Water flow





Christian Troost

Using MPMAS with the MpmasQL toolbox

User manual & reference

UNIVERSITÄT HOHENHEIM

MpmasQL 2.52

User manual & reference

Contact: Christian Troost

Dept. of Land Use Economics in the Tropics and Subtropics (490d)
Hohenheim University, 70599 Stuttgart, Germany

christian.troost@uni-hohenheim.de

This manual is available at `https://mp-mas.uni-hohenheim.de`

Manual version 1.20, created August 14, 2014
© Christian Troost, Universität Hohenheim 2011-2014

Contents

1 Installation 1

2 Overview 3

3 Model design with mpmasql: A basic MPMAS model 5

3.1 Folder structure . . . 5
3.2 Control file . . . 6
3.3 The model configuration . . . 6
3.4 The landscape . . . 7
3.5 The agent population . . . 8
3.5.1 Farmsteads and land ownership . . . 8
3.5.2 Populations, clusters and networks . . . 9
3.5.3 Household members, and the dynamics of aging and labor provision . . . 9
3.5.4 Money, machinery and other farm assets . . . 12
3.6 Production and investment decisions . . . 15
3.6.1 Crop production, soils and yields . . . 15
3.6.2 Liquidity needs . . . 20
3.6.3 Labor use . . . 22
3.6.4 Machinery use . . . 24
3.6.5 Investments . . . 26
3.6.6 Hiring machinery . . . 28
3.7 Running the model . . . 29
3.7.1 Creating the MPMAS input files with `mpmasql` . . . 29
3.7.2 Running MPMAS . . . 30
3.8 Scenarios . . . 30
3.9 Output analysis . . . 31
3.9.1 Defining output variables . . . 31
3.9.2 Importing results into stata . . . 32

3.9.3 Importing results into R . . . 32
3.9.4 Transforming results into text files/databases . . . 33

4 Model design with mpmasql: MPMAS submodels and features 34

4.1 Perennial crops . . . 34
4.2 Livestock . . . 37
4.3 Crop growth models . . . 38
4.3.1 CropWat . . . 39
4.3.2 External crop growth models . . . 40
4.3.3 Exogenous yield time series . . . 41
4.3.4 Adaptation of production decisions after harvest . . . 41
4.3.5 Yield expectations for crops not grown . . . 42
4.4 Consumption . . . 42
4.4.1 Basic consumption model . . . 43
4.4.2 Advanced three-stage consumption model . . . 43
4.5 Harvest decision . . . 48
4.6 Hydrology . . . 48
4.6.1 EDIC: Sector-based Hydrology . . . 48
4.7 Fine-tuning of OSL solver . . . 48
4.8 Quadratic Problems . . . 49
4.8.1 Example starting from a scalar valued function . . . 49
4.8.2 Example for quadratic risk programming . . . 50
4.9 Exogenous changes to matrix coefficients and agent-independent right hand sides . . 51
4.10 Fragmented markets: differentiate prices or LP coefficients between different groups of agents . . . 51
4.11 Producer organizations . . . 52
4.11.1 The producer organization . . . 52
4.11.2 The farm agents' marketing decision . . . 54
4.11.3 Linking farm agents and producer organizations . . . 55
4.12 Diffusion of innovations . . . 56
4.12.1 Defining the number of segments and its thresholds . . . 56
4.12.2 Defining innovations . . . 57
4.12.3 Defining availability and accessibility of innovations . . . 58
4.12.4 Assigning agents to networks and segments . . . 58
4.12.5 Fine-tuning the diffusion process . . . 59
4.13 Using agent attributes in the decision problem . . . 59

4.14 Using plot attributes in the decision problem 59
4.15 Household member-specific attributes 60
4.16 Selling of assets 61

5 Testing decision models with mpmasql and mqlmatrix 63

5.1 Interactive commands 63
5.2 Formats of MP problems 65
5.2.1 MPMAS input format (.dat) 65
5.2.2 MPMAS standalone MP problem format (.mtx) 66
5.2.3 MP problems automatically saved by `mpmas` for debugging 67
5.2.4 MP problems saved by `mpmas` for debugging on request 67
5.3 Indicating the location of the model info files 68
5.4 mqlmatrix.conf 68

6 Preparing agent populations with mpmasdist 69

6.1 File structure 69
6.1.1 DB 69
6.1.2 VARIABLES 70
6.1.3 TABLES 70
6.1.4 MAPS 70
6.1.5 CONTROL 70
6.1.6 DISTRIBUTION 71
6.2 Generating, removing, saving and importing agents 72
6.2.1 Generating agents 73
6.2.2 Removing agents 73
6.2.3 Saving agent populations 74
6.2.4 Restoring agent populations 74
6.3 Defining and manipulating attributes and assets 74
6.3.1 Setting and using attributes 75
6.3.2 Removing attributes 75
6.3.3 Assigning assets and household members 76
6.4 Distribution algorithms for attributes, assets and household members 77
6.4.1 Distributing a list of observations 77
6.4.2 Assignment following a probability distribution 83
6.5 Spatial distribution 86
6.5.1 Quick maps 86

6.5.2 More realistic maps 87
6.5.3 Making population, cluster, network, sector and catchment maps 90
6.6 Varying agent populations 90

References **92**

A Command line options **93**
A.1 mpmasql 93
A.2 mqlmatrix 94
A.3 mpmasdist 95
A.4 mpmas 95

B mpmasql file reference **99**
B.1 Control file 99
B.2 Model configuration 102
B.3 Model data file 106
B.3.1 [GLOBALS] 107
B.3.2 [MPMAS PARAMETERS] 107
B.3.3 [MPMAS TABLES] 112
B.3.4 [USER TABLES] 124
B.3.5 [USER VARIABLES] 124
B.3.6 [INSTANCES] 124
B.4 Model structure file 125
B.4.1 [ACTIVITY TYPES] and [CONSTRAINT TYPES] 125
B.4.2 [DEFAULTS] 127
B.4.3 [MODEL] 127

C mpmasql function language **146**
C.1 Numbers and Strings 146
C.2 Operators 146
C.3 Arrays 147
C.4 Functions 148

D Version history and changelog **151**
D.1 Version History 151
D.2 Changes from version 1.03 to 2.00 151
D.2.1 Conversion control file (.ini) 153
D.2.2 Configuration file (.cfg) 153

D.2.3 Data file (.var) 154
D.2.4 Model structure file (.cll) 155
D.3 Changes in 2.03 157
D.4 Changes in 2.04 157
D.5 Changes in 2.08 157
D.6 Changes in 2.10 157
D.7 Changes in 2.12 158
D.8 Changes in 2.14 158
D.9 Changes in 2.16 158
D.10 Changes in 2.19 158
D.11 Changes in 2.20 158
D.12 Changes in 2.21 158
D.13 Changes in 2.22 159
D.14 Changes in 2.23 159
D.15 Changes in 2.24 159
D.16 Changes in 2.26 159
D.17 Changes in 2.27 159
D.18 Changes in 2.28 159
D.19 Changes in 2.32 159
D.20 Changes in 2.33 159
D.21 Changes in 2.36 160
D.22 Changes in 2.39 160
D.23 Changes in 2.40 160
D.24 Changes in 2.41 160
D.25 Changes in 2.47 160
D.26 Changes in 2.48 160
D.27 Changes in 2.52 161

Chapter 1

Installation

mpmasql has been developed for GNU/Linux operating systems. You may have to install certain additional software (Database, perl interpreter and certain Perl modules). You also need a working internet connection, so missing Perl modules can be installed during the installation.

OS GNU/Linux (tested on Ubuntu 10.04/ 12.04)

perl The use of `mpmasql` requires the installation of a perl interpreter on your computer. Most Linux distributions come with one included. `mpmasql` 2.52 was tested under perl v5.10.1 (Ubuntu 10.04). Non-standard Perl modules will be installed by the `mpmasql` installation script.

Install `mpmasql` Installing the main program requires a working internet connection.

1. Download and extract mpmasql.zip,
2. Open a terminal and change into the directory mpmasql_release_252
3. If you do not want to use `mpmasdist`, run `sudo ./install_mpmasql`
4. If you want to use `mpmasdist`, run `sudo ./install_mpmasql -x`. (This installs also the additional Perl modules needed for `mpmasdist`. You can also skip this now and run it later whenever you decide to use `mpmasdist`.)

Installing OSL solver library The `mpmasql` installation script will automatically install the mpmas executable in `/usr/local/bin`. The mpmas executable requires the IBM OSL solver library to run and this has to be installed by the user. (Please consult i490d@uni-hohenheim.de if you have problems installing this library.)

1. Copy `v3_osllib_linux.tar` to your home folder and extract it
2. From the extracted `v3_osllib_linux` folder extract osllib.tar
3. Copy the extracted subfolder lib/ and the `install_osl` program to your home folder, e.g. /home-/lib/ (if this folder does not exist, create it)
4. Open a terminal, cd to your created folder (e.g. exampleuser) and type `sudo ./install_osl osllib academic` and press enter. The process for retrieving a license will start. To accept the license type yes if asked and enter username and institution

5. Open the file .bashrc in your home directory with a text editor (it is a hidden file, if you use the file browser to find the file you need to tick View -> Show hidden files) and append the following lines (note: the format is important here):

```
if [ "$LD_LIBRARY_PATH" = "" ]; then
  export LD_LIBRARY_PATH=~/lib
else
  export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:~/lib
fi
```

Database Although not strictly required, `mpmasql` has been developed for the use with MySQL databases. MySQL is open source. Under Linux, it is either already included in your distribution or you can get it from your Linux repository. (Install the packages mysql-client and mysql-server.)

You have to fill your database with data, of course, before you can use `mpmasql` to create MPMAS input files from it. Generally you are very flexible in the set-up of your databases, as long as you can provide tables in the required format to `mpmasql` with one single (but possibly nested) SQL "SELECT"-statement.

Don't forget to create a (or choose an existing) database user account for your script, grant use and select privileges to all databases you want to use with `mpmasql`, and adapt the database connection entries in the conversion control file.

If you want to use the tutorial database, you can create it once you have installed a MySQL database server on your computer. Use the MySQL client of your choice to create a schema named `mpmasql_example_db` and a user account for your model to which you give access to the new schema (The example model uses the username 'testuser' and password 'testuse', but you can adapt that in example.ini.). Open the script `create_example_db.sql` in the subfolder `example_db/` in a graphical client interface like MySQL Workbench and run it, or use the command-line client:

```
mysql  -u <username> -p  mpmasql_example_db < create_example_db.sql
```

Chapter 2

Overview

MpmasQL is a toolbox that contains a collection of tools that assist you in working with the multi-agent model MPMAS. All of these tools are command line programs, i.e. they do not have a graphical user interface, but are intended to be run using the Linux terminal. Input data for the tools is provided using text files and database connections.

Apart from providing the current version of the `mpmas` executable, the MpmasQL package contains the following tools:

`mpmasql` is the core tool in the toolbox, used to prepare input files for `mpmas`.

`mqlmatrix` can be used to test mixed integer programming matrices, which are used to model several decisions in `mpmas`.

`mpmasdist` can be used to create agent populations based on statistical distribution functions and theoretical assumptions, including random distribution of agricultural plots over study areas

Figure 2.1 provides an overview of the modeling process using the MpmasQL toolbox. The basic idea is to store any empirical information you gathered in the field or from secondary sources in a relational database (e.g. MySQL) in a structured and normalized form.[1] Spatial data should be processed and stored using GIS software and linked to the database (e.g. the GIS maps could contain information about the spatial distribution of soil types, but any further information on how the different soil types affect production should be stored in the database). Often detailed information about all the individuals to be represented as agents in the model is not available. In those cases, `mpmasdist` can help you create artificial, but representative agent populations using statistical information and heuristic rules.

Based on your theoretical and empirical knowledge about the processes and interactions relevant for your simulations, you can then create a model design, which you describe in three text files using the model design format required by `mpmasql`. This model design defines the model structure and contains references to information in the database, but does not contain the data itself. Further, you create a control file instructing `mpmasql`, which model design and database to use, which scenarios to create and how to adapt the model for different scenarios.

You can then run `mpmasql` and it will create the input files for `mpmas` model description files and importing scripts for statistical software. Whenever you add or update something in your database that is referenced in the model design, this will automatically be reflected in the `mpmas` input the next time you run `mpmasql`, because the model design only contains references to the data and not the data itself. Spatial input for `mpmas` can be created directly with GIS software.

After you ran `mpmas` you can use the importing scripts created by `mpmasql` to import the `mpmas` output into a statistical software package for analysis (currently Stata and R are supported).

[1] `http://en.wikipedia.org/wiki/Database_normalization`

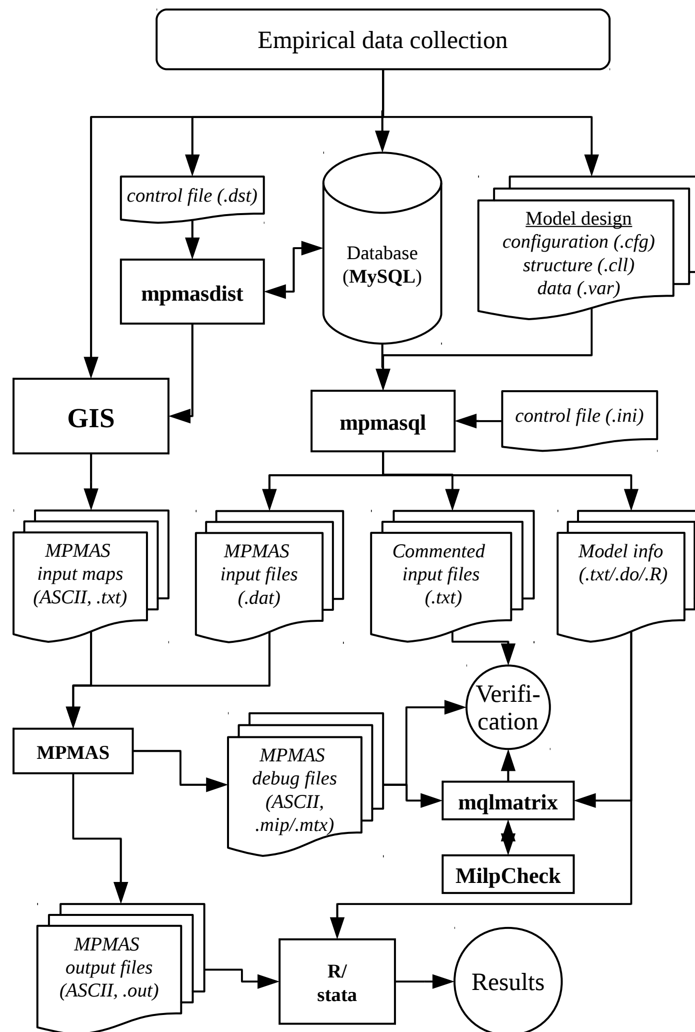


Figure 2.1: Overview of the modeling process with MPMAS using the MpmasMySQL setup

Chapter 3

Model design with mpmasql: A basic MPMAS model

`mpmasql` prepares `mpmas` input files using data stored in a database and a model design described in three text files: The model configuration file (.cfg), the model structure file (.cll) and the model data file (.var). Despite its name the model data file does not contain the actual data (except for some simple parameters), but instructs `mpmasql` which data to use from the database. A fourth text file, the control file (.ini) tells `mpmasql` which model design files and which database to use, where to write its output and which scenarios to create.

Note: The files which are described in this chapter are contained in the `mpmasql_tutorial_files/` folder of mpmasql.zip. In the subfolder `db/` you find the backup of the corresponding mysql database. You can load this to your server by first creating the corresponding schema example_db on the server, and then run

```
mysql -h hostname -u username -p example_db <
    create_mpmasql_example_db.sql
```

in a terminal.

3.1 Folder structure

It is convenient to keep all files for a model application in one folder. The following subfolder structure is suggested (but can be adapted according to your own needs, if you adapt `mpmasql` files and `mpmas` command line options).

```
<modelversion>/
   cfg/         -> mpmasql model design files
   input/
      dat/  -> input files for MPMAS
      gis/  -> input maps for MPMAS
    out/        -> MPMAS writes output here
       test/    -> MPMAS writes debugging info here
   xlsInput/    -> commented input files
```

The mpmasql control file should be saved at the lowest hierarchy level, i.e. in the folder that bears the name of the model version.

3.2 Control file

The content of the control file looks like this:

```
[INPUT]
CONFIGURATION        = cfg/example.cfg
STRUCTURE            = cfg/example.cll
DATA                 = cfg/example.var

DBTYPE               = mysql
DB                   = example_db:localhost
DBUSER               = username
DBPWD                = somepassword

[OUTPUT]
OUTDIR               = input/dat/
OUTDIR_CM            = xlsInput/
PREFIX               = example
```

The first three entries under the [INPUT] headline tell `mpmasql` the names of the model design files and that they are located in a subfolder called `cfg/`. The next four entries contain all the information needed to connect to the database: Type, database name and host, user name and password. If you do not specify the password in the file, you will be asked for it at runtime.

The three entries in the [OUTPUT] section tell `mpmasql` to write the MPMAS input files to the subfolder `input/dat/`, the commented files to the subfolder `xlsInput/` and use "example" as the first part of the filename. (The rest of the filenames consists of a scenario name in case you use the scenario function, and fixed parts required by MPMAS to recognize the files.)

There are many more entries you can include into the control file. They will be explained in later sections. The full overview is given in appendix B.1.

3.3 The model configuration

In the model configuration, you control the general setup of your MPMAS model. When does the simulation start, how many years should it run, which submodules shall be used and so on. Your configuration should at least contain the following entries:

```
[TIME]
STARTYEAR            = 2010
STARTMONTH           = 7
STARTDAY             = 1

ENDYEAR              = 2030
ENDMONTH             = 6
ENDDAY               = 30

SEASONSTART          = 10
SEASONEND            = 9

NORTHERN             = 1

[LANDSCAPE]
NSOIL                =  3
```

The first six entries under the headline [TIME] mean that the simulation shall cover the time span from July 1, 2010 to June 30, 2030, i.e. 20 simulation years. The next two entries inform MPMAS that the cropping season starts every year in October and ends in September, and setting NORTHERN to 1 indicates that the study region is situated in the Northern hemisphere (choose 0 for Southern hemisphere). The NSOIL entry informs MPMAS how many different soil types it has to expect in the soil input maps.

Again, there are many more entries you can include into the configuration file. They will be explained at the appropriate places in later sections of this manual. The full overview is given in appendix B.2.

3.4 The landscape

Land is the defining resource of agriculture and no MPMAS model can do without a landscape, which is defined using maps in ESRI ASCII raster format. Maps in this format are just text files, meaning you can create them in a simple text editor or spreadsheet software. Sooner or later, however, when your model gets larger it is much better to use real GIS software[1], nearly all of which can create and read maps in this format.

Each map file in ESRI ASCII format starts with lines of metadata:

```
NCOLS 5
NROWS 5
XLLCORNER 3513136
YLLCORNER 5403903
CELLSIZE 100
NODATA_VALUE -1
```

The first two lines indicate the number of columns and rows of the cell raster (grid). So, in the example, the map has 25 cells, distributed over five rows and five columns. The next two entries indicate the geodetic coordinates of the lower left corner of the lower left cell of the grid. These values do not matter to MPMAS, but are important if you import the maps into a GIS and want to compare it to other spatial information. The cell size indicates the size of one cell (i.e. the length of one side), and the sixth line says that all cells containing a -1 should be regarded as empty. After this header comes the actual cell grid. Each line equals to one row of the grid, and in each row there are NCOLS numbers separated by spaces (tabs do work too for MPMAS) , each containing the actual value observed at that point in the landscape.

Three files define the study area, its subdivisions (villages, administrative areas, hydrological catchments, sectors or similar), and the different terrains (e.g. soil types) that provide different opportunities for agricultural production. Subdivisions are required for certain submodels (e.g. the EDIC model), but should be avoided if not necessary. Any farm agent you later on place into the landscape can only own and rent land in the sector, where his farmstead lies. All map files should be placed into the input/gis subfolder.

Study area The map containing the study area delimitation is called `CatchMap00Catchment.txt`, for historical reasons.[2] It might look like this, indicating that the three cells in the upper right corner do not belong to the study area.

```
NCOLS 5
NROWS 5
XLLCORNER 3513136
YLLCORNER 5403903
```

[1] Have a look at Quantum GIS (`www.qgis.org`) if you look for a free, open-source GIS.

[2] Theoretically it is possible to have several study areas, or hydrological catchments, in one model, but this feature is currently still experimental. Files of additional catchments would then be named `CatchMap01Catchment.txt`.

```
CELLSIZE 100
NODATA_VALUE -1
0 0 0 -1 -1
0 0 0 0 -1
0 0 0 0 0
0 0 0 0 0
0 0 0 0 0
```

Sectors The map containing the subdivision of study areas into sectors is called `CatchMap00Sector.txt`. In the usual case of no subdivisions, it can look the same as the study area file, defining just one sector with the ID 0.

Terrains The different types of terrains you want to distinguish in your landscape, have to be numbered starting from zero. As the soil type is usually a major determinant of agricultural production the corresponding map is called `CatchMap00Soil.txt`. However, the soil types in this map do not necessarily have to refer to pedological soil types, but could also be used e.g. to distinguish between irrigable and non-irrigable soil, flat and inclined areas, and so on. The following example shows a map with three soil types. (Remember to set the model configuration parameter NSOIL accordingly).

```
NCOLS 5
NROWS 5
XLLCORNER 3513136
YLLCORNER 5403903
CELLSIZE 100
NODATA_VALUE -1
0 0 0 -1 -1
0 2 2 2 -1
1 0 2 2 1
1 1 2 0 0
0 0 1 1 0
```

3.5 The agent population

A multi-agent model needs, of course, a number of agents, which we call the agent population. In MPMAS the main class of agents are farm households. (In fact there may be several agent populations with different characteristics, e.g. different ethnicities, and also very different types of agents. But we'll come to that later.)

3.5.1 Farmsteads and land ownership

Every farm household owns a farmstead, which is identified using an ID in the farmstead map, `CatchMap00Farm.txt`. This farmstead ID is the identifier of the farm in all other model design files. You can freely choose any integer number greater than zero, and smaller than 999900000. E.g. the following map contains three farmsteads.

```
NCOLS 5
NROWS 5
XLLCORNER 3513136
YLLCORNER 5403903
CELLSIZE 100
```

```
NODATA_VALUE -1
-1 -1 300 -1 -1
-1 -1 -1 -1 -1
-1 -1 -1 15 -1
67 -1 -1 -1 -1
-1 -1 -1 -1 -1
```

Every farm household should also own some land. It definitely owns the land where its farmstead is located, but it can of course also own other cells, as long as they lie in the study area, in the same sector as the farmstead, and on a cell for which a soil type has been defined. The land properties of the farm household are specified in the property map, `CatchMap00Prop.txt`. In our example, household 300 owns seven plots, household 15 owns five plots and household 67 owns eight plots. Two plots are owned by no one.

```
NCOLS 5
NROWS 5
XLLCORNER 3513136
YLLCORNER 5403903
CELLSIZE 100
NODATA_VALUE -1
300 300 300 -1 -1
67 300 300 15 -1
67 15 300 15 15
67 15 300 67 -1
67 67 67 -1 67
```

3.5.2 Populations, clusters and networks

Farm households can belong to different populations and different communication networks. Different populations may have different life expectancies, birth rates and labor force, while communication networks are relevant for simulating the diffusion of innovations (see 4.12) and may face different conditions for financing. Populations may be further subdivided into clusters, which are used during the initial allocation of farm assets (see **??**). Clusters, networks and populations are counted from zero and, for historical reasons, the association of a farm household with populations, clusters and networks has to be specified in maps. For our example and assuming all agents are associated with the same population, cluster and network, all three maps (`CatchMap00Pop.txt` , `CatchMap00Clu.txt` and `CatchMap00Netw.txt`) would look like this:

```
NCOLS 5
NROWS 5
XLLCORNER 3513136
YLLCORNER 5403903
CELLSIZE 100
NODATA_VALUE -1
-1 -1 0 -1 -1
-1 -1 -1 -1 -1
-1 -1 -1 0 -1
0 -1 -1 -1 -1
-1 -1 -1 -1 -1
```

3.5.3 Household members, and the dynamics of aging and labor provision

Every farm household consists of a number of household members. Before you can assign household members to a household, you first have to define, which types of household members you want

to be able to assign. You do so using a table of five columns, which you assign to the entry HOUSEHOLD_MEMBER_TYPES, which is part of the section [MPMAS TABLES] in the model data file.

You do not put the table itself into the model data file, rather you provide a link to the table, either by specifying the name of a text file containing the table in comma-separated format, or by providing an SQL statement that retrieves a table of the required format from the database. The advantage of a database query is that you do not actually need to have a table of the required format in the database, you just need to be able to formulate a query that results in a suitably formatted table. For the definition of household member types, this is not so important, but for many other tables it gives you the opportunity to store your data in a way that is best suited for the data, and then extract a table format that suits your model by queries. If you change your model setup, or design scenarios, you then simply need to adapt your queries and not your tables. For the following we will always assume that you use SQL queries to specify tables.

A suitable query for the definition of household member types could look like this:

```
[MPMAS TABLES]
 HOUSEHOLD_MEMBER_TYPES = (sql) { "SELECT hh_member_type, career,
    sex, lowage, upage FROM tbl_hh_member_types"}
```

The column names are not actually important, what is important is the order of the columns. The first column gives a name to the household member type. It serves as an identifier for the type. Identifiers are used very much in `mpmasql` and may consist of the letters a-z or A-Z, the numbers 0-9 and the underscore. The second column links the member type to a course of life or career that describes what happens to the person when it gets older. This is discussed in detail a bit further below. The third column indicates the gender, where 0 indicates female and 1 indicates male. The fourth and fifth column the lower and upper age. You could be very specific and define a member type for each potential age a household member might have, or you could be rather coarse and give an age range, then MPMAS will choose the age the person has at the beginning of the simulation randomly from within this range.

The table in your database could look like this:

hh_member_type	career	sex	lowage	upage
boy	male_farmer	1	0	16
young_man	male_farmer	1	17	30
man	male_farmer	1	31	60
elder_man	male_farmer	1	61	80
girl	female_farmer	0	0	16
young_woman	female_farmer	0	17	30
woman	female_farmer	0	31	60
elder_woman	female_farmer	0	61	80

Now, you can use these household member categories to assign household members to each household. You do this with the entry HOUSEHOLD_COMPOSITION, which is also part of the [MPMAS TABLES] section. This table has three columns, the first refers to the farm household by its farmstead id, the second to the type of household member and the third to the number of members of that type, which form part of the household.

```
 HOUSEHOLD_COMPOSITION  = (sql) {"SELECT farmstead_id,
    hh_member_type, quantity FROM tbl_hh_members"}
```

The corresponding table in your database could look like this:

farmstead_id	hh_member_type	quantity
300	man	1

300	woman	1
300	boy	2
300	girl	1
67	young_man	1
67	elder_woman	1
67	elder_man	1
15	young_man	1
15	young_woman	1

This table provides the composition of households at simulation start. Unless you turn off aging of households (setting OFFHHAGEING to 1 in the model configuration), they will get older when the simulation years pass. This has consequences for the type and amount of labor they can provide, the probability of dying or giving birth, and in case you use an advanced consumption model, also for the amount of food they need.

The corresponding information is given in the table HOUSEHOLD_DYNAMICS, which has at least seven columns.

```
HOUSEHOLD_DYNAMICS = (sql) {"SELECT pop, career,  upperage,
                                     labgrp, labprov, mortality,
                                        fertility
                                   FROM tbl_hh_dynamics"}
```

The first column refers to the population for which the information holds (remember that we could have several agent populations), the second is the code for the career and the upper bound of the age range for which the following information holds. (The lower bound is given by the next lowest upper bound in another line referring to that career or zero.) The fourth column specifies which type of labor the household member is able to provide at that age. This gives you the possibility to distinguish different types of labor to reflect cultural, legal or educational constraints to certain activities. The fifth column gives the amount of labor (in the unit of your choice) a household member can provide at that age in one year. The sixth and seventh columns give the probability of dying, respectively giving birth to a child, in each year of the age range.

For example, a household dynamics table could look like the following, if we want to distinguish only one type of farm labor and take the yearly work power of an adult man as a reference unit for the labor provision.

pop	career	upperage	labgrp	labprov	mortality	fertility
0	female_farmer	12	farm	0	0.01	0
0	female_farmer	16	farm	0.3	0.001	0.001
0	female_farmer	21	farm	0.9	0.003	0.003
0	female_farmer	45	farm	0.9	0.003	0.03
0	female_farmer	70	farm	0.9	0.005	0
0	female_farmer	90	farm	0.2	0.003	0
0	female_farmer	91	farm	0	1	0
0	male_farmer	12	farm	0	0.01	0
0	male_farmer	16	farm	0.3	0.001	0
0	male_farmer	25	farm	1	0.008	0
0	male_farmer	70	farm	1	0.004	0
0	male_farmer	90	farm	1	0.3	0
0	male_farmer	91	farm	1	1	0

3.5.4 Money, machinery and other farm assets

Now, you will probably assign some other resources or assets to farm households, apart from land. This is done using a table called ASSET_ENDOWMENTS. It looks very similar to the household composition table:

```
ASSET_ENDOWMENTS    = (sql) { "SELECT farmstead_id, asset, quantity
    FROM tbl_farm_assets" }
```

Cash

One of the basic assets of a household is cash, or liquidity as it is called in MPMAS. You have two options, how to assign cash to a household in the model. You can give a default amount of cash to every household, by setting the LIQUIDITY entry in the [MPMAS PARAMETERS] section of the model data file. E.g.

```
[MPMAS PARAMETERS]
LIQUIDITY   =  1000
```

Now every household in the model would have 1000 monetary units cash at the start of your simulation. You can override this default for every household by assigning the special asset 'liquidity' to a household. E.g. if the table tbl_farm_assets in your database contains the following lines,

farmstead_id	asset	quantity
300	liquidity	8000
15	liquidity	250

Household 300 would start with 8000 and household 15 with 250 monetary units, while household 67 would start with the default 1000 monetary units.

Other assets

Cash and land are the only assets that are predefined by MPMAS. All other assets have to be defined in the model structure. So let's say you want to assign some basic implements for crop production to the agents, e.g. ploughs and tractors. While you do have to create a specific asset plough and an asset tractor, these will look very similar as both are machinery and only differ in some of their attributes, e.g. price and lifetime. Later on, other model elements, like LP activities and constraints, will be associated to these assets and they will also look very similar and only distinguish itself in a few coefficients. We can therefore create a blueprint for machinery that tells `mpmasql`, which model elements need to be created for each type of machinery. Objects that share similar characteristics can be said to belong to a class, e.g. in our case the class 'machinery'. The objects that belong to a class are called its instances, i.e. in our case a plough, a tractor or a seeder would all be instances of the class machinery.

Defining assets Blueprints for classes are defined in the [MODEL] section of the model structure file.

```
 [MODEL]
#class MACHINERY {
   #asset machinery_asset {
          name  #=    #this(instance)
          div      #= 0
```

```
        lifetime #= #table(machinery, 2, [ #this(instance)] )
        acqcost  #= #table(machinery, 3, [ #this(instance)] )
        eqshare  #= 0.25
        irate    #= #var(INTEREST_LGCRD)

    }
}
```

The above code tells mpmasql that for every instance of the class MACHINERY, an asset, which we identify as machinery_asset has to be created. The name of the asset, which is used as an information in commented input and model description files, shall simply be the identifier of the instance. Machinery assets are indivisible, i.e. you cannot buy half a tractor, so we set the 'div' attribute to 0.

The expected use life of the asset (lifetime) can be found in the second column of the user-defined table 'machinery' in the row, whose key contains the identifier of the instance. The price that was or has been paid when buying the asset (acqcost), can be found in the third column of the same table. When such an asset is bought, the buyer has to pay 25% of the price in cash (eqshare), the rest can be financed using a credit that has the same duration as the use life of the asset at an interest, which is the same for all assets of the class machinery and equal to the MPMAS parameter INTEREST_LGCRD, i.e. the interest on long-term credits.

Similar to household members also assets get older every year. Standard assets, however, do not change their characteristics like household members when getting older. However, their lifetime is limited and once it is reached they have to be renewed or the agent has to do without them. During their lifetime, the household has to pay debt service for the share it did not finance itself and the associated interest. (There are special assets like livestock or perennial crops, which do change their characteristics with age, but these will be discussed in the next chapter.)

The functions \#this(), \#table() and \#var() constitute part of the mpmasql function language (MFL) and are used to form expressions that help you make values of model elements (or even the structure of model elements) dependent on instances or scenarios. MFL expressions can be used at many different locations in the model design files. A full reference of all the functions is given in section C.

We now still have to tell mpmasql, which instances are part of the MACHINERY class and provide the data that we referred to in the class definition. First, we create the instance list in the [INSTANCES] section of the model data file.

```
 [INSTANCES]
MACHINERY   =  [tractor, plough, seeder]
```

In the example, we have included three instances into the model by including their identifiers into the instance list for the machinery class. An identifier must consist only of letters [a-z, A-Z], digits and underscores. The identifier is used to create a unique identifier for the assets, which consist of the instance identifier, an underscore and the identifier you used in the element definition, i.e. in our case, the model will contain assets of type tractor_machinery_asset, plough_machinery_asset and seeder_machinery_asset.

Second, we have to provide the table 'machinery', which contains the data on lifetime and acquisition cost. This is a user-defined table, because contrary to the MPMAS tables, the format is not predefined by MPMAS. The most important thing you have to define is, how many key columns your table contains. Key columns are used to identify the rows in your table. This key can consists of the content of one or several columns, which have to be the first columns in the table. In our case, we need only key column, which contains the instance identifier, i.e. the type of machinery.

Definitions of user tables are placed into the [USER TABLES] section of the model data file, and are of the following format:

```
tablename   = (tabletype, number of key columns) { tablesource: MFL
   expression that can
```

```
                    span several lines and results in an SQL query
                       for sql tables, and
                    a filename for ascii tables
                      }
```

For example,

```
machinery   = (sql,1) {"SELECT machinery, lifetime, acqcost FROM
   tbl_machinery"}
```

The corresponding table might look like this:

machinery	lifetime	acqcost
tractor	10	10000
plough	6	1000
seeder	12	3000
harrow	8	760
cultivator	8	900

When `mpmasql` now creates the asset for the tractor instance, it will encounter the MFL expression `#table(machinery, 2, [ #this(instance)] )` for the field lifetime. The first thing it will do is replace `#this(instance)` by 'tractor', leaving it with `#table(machinery, 2, [ tractor ] )`. It then looks into the table machinery, looks for the row which has a value equal to 'tractor' in the first column and takes the value it finds in the second column of that row, and then knows that the lifetime of a tractor is 10 years.

Note that the table contains information for more types of machinery than we used in our model. We simply deliberately chose only to consider tractors, ploughs and seeders, the additional information is read by `mpmasql` but remains unused. If we want to change that and include all the machinery for which there is information in the table, we can use the #table function in the definition of the instance list:

```
 [INSTANCES]
MACHINERY   =  #table(machinery, 0, [])
```

The third argument to the #table function is the so called key array, whenever it is empty the function simply returns all different entries in the first column of the table, so in our case it would be equivalent to writing

```
 [INSTANCES]
MACHINERY   =  [tractor, plough, seeder, harrow, cultivator]
```

The main difference is, that when you add a row to the table machinery, e.g. information for a combine, this will automatically be added to the instance list in the first case, but not in the second.

Finally, we still have to set a value for the MPMAS parameter INTEREST_LGCRD. This is done in the [MPMAS PARAMETERS] section of the model data file. E.g. if the interest rate for long-term financing is 8%, we should wirte the following:

```
 [MPMAS PARAMETERS]
INTEREST_LGCRD =  0.08
```

Assigning assets to agents The asset definitions we just created, can now be used to assign assets to the agents. We simply have to add corresponding entries to the AGENT_ENDOWMENTS table we already introduced above. E.g. if household 300 owns a full set of machinery, household 67 only a plough and household 15 no machinery at all, the table would have to look like this:

farmstead_id	asset	quantity
300	liquidity	8000
15	liquidity	250
300	tractor_machinery_asset	1
300	plough_machinery_asset	1
300	seeder_machinery_asset	1
300	harrow_machinery_asset	1
300	cultivator_machinery_asset	1
67	plough_machinery_asset	1

3.6 Production and investment decisions

We now have defined the households, its members and assets, and what will happen to them inevitably during the course of their life. The most interesting thing about an agent, however, is that it can act according to his own decisions. In MPMAS, the most important decision for a farm household is what it grows or produces this year in order to make a living. This decision is modeled using constrained mathematical programming, i.e. the agent chooses the production plan which is optimal with respect to his objective, in most cases maximizing expected income, while at the same time feasible given all technical, financial and resource constraints to production and satisfying all other needs and objectives of the household, e.g. not going bankrupt or producing enough food to feed the household members in the case of subsistence farmers. The decision is modeled using mixed integer linear programs and thus provides a very flexible tool, that can be adapted by the model user to reflect all kind of empirical or theoretical situations.

A linear program consists of equations for the objective and the constraints, which contain linear combinations of variables (activities), and is often presented in matrix format, where the columns represent variables and the rows the equations and each matrix cell represents the coefficient of the variable in the equation. Often only non-zero coefficients are included into the matrix. Equations are equalities or inequalities and are usually arranged such that all terms containing variables are arranged on the left hand side of the (in)equality sign and all constant terms on the right hand side.

3.6.1 Crop production, soils and yields

Formulating equations For our farm household, we will first consider two kinds of activities. It can produce a certain range of crops and it can sell them. To reflect this, we need to include a production and a selling activity S for each crop c into the model. These activities are connected by a balance equation, i.e. you cannot sell more of a crop c than you produced. Production is equal to the yield (y_c) times area on which you produced the crop A_c:

$$S_c \leq y_c * A_c \tag{3.1}$$

Rearranged such that all variable elements are left and all constant elements right. In this case there is no constant element, thus the right hand side is zero.

$$S_c - y_c * A_c \leq 0 \tag{3.2}$$

Of course, the household cannot cultivate an infinite area, but is restricted by the amount of land it owns. Further, we expect different yields on different soil types and consequently we have to distinguish production depending on the soil type j it is produced on. The equation then becomes

$$S_c - \sum_j y_{c,j} * A_{c,j} \leq 0 \tag{3.3}$$

.

The area restriction, aslo distinguished by soil types, can be formulated as follows

$$\sum_c A_{c,j} \leq N_j \quad , \tag{3.4}$$

where N_j is the amount of land of soil type j the household owns.

The objective equation, representing the income of the agent, would contain the revenue of his crop sales, i.e. price (p_c) times amount sold, minus the production cost, i.e. variable cost (v_c) times area:

$$\max! \quad p_c * S_c - v_c * A_c \tag{3.5}$$

Implementation in `mpmasql` Again using the concept of classes and instances, looking at the above we can say we are dealing with two classes, the class CROPS and the class SOILS, and we can already add two instances lists for these two classes in the [INSTANCES] section of the model data file:

```
SOILS =  #steps(0, #var(NSOIL) - 1, 1)
CROPS =  [wheat, barley, maize]
```

While we provide the CROPS instances explicitly, the SOILS list is formed using the `\#steps` function. To create a list of numbers starting from 0 and ending with the number of soils in the model minus one, with a distance of 1 between each number and the neighboring. In our case this is equivalent to writing [0,1,2], i.e. the soil type identifiers we used in the map. However, whenever we start using more soil types, the list will be automatically updated.

Next, we should create the blueprints for each class in the model structure file:

```
#class CROPS {
   #constraint balance {
      name  #= #this(instance) ~ "_balance"
      unit     #= t
      type     #= product_balance
      eqtype   #= 1
   }
   #activity sell {
      name  #= "sell "~#this(instance)
      type  #= MASsell
      unit     #= t

      orow  #= #table(prices, 3 , [#this(instance), #var(STARTYEAR) -
         1 ] )
      orow_#foreach(YEARS) #= #table(prices, 3 , [#this(instance),
         #this(field,1) ] )

      #this(instance)_balance #= 1

   }
   #activityby production_on_soil #foreach(SOILS) {
      name  #= "produce " ~ #this(instance) ~ " on soil " ~
         #this(by,1)
      unit     #= ha
```

```
        type     #= crop_production

        orow  #= -1 * #table(variable_costs , 2 , [#this(instance) ] )

        #this(instance)_balance #= -1 * #table(yields , 3,
           [#this(instance), #this(by,1) ] )
        #this(by,1)_soil       #= 1
    }
}
#class SOILS {
    #constraint soil {
        name  #= "Area of soil " ~#this(instance)
        type     #= MASsoil
    }
}
```

This looks quite complicated at the first glance, but we will go through the fields and expressions one by one.

name, unit Like with assets, the ‘name’ field in all model elements contains a description of the activity respectively constraint that will appear in the commented model files and the model info. The ‘unit’ field indicates the unit of the variables, respectively the terms in the equation. It serves also only informative purposes and is not used by MPMAS itself.

eqtype The ‘eqtype’ field of a constraint indicates the equation sign used between left and right hand side. A one stands for less or equal($\leq$), a two for greater or equal ($\geq$), and a three for equal ($=$). Less or equal is the default, so we could also omit this in our case, as we have done for the soil constraints.

type The type field indicates special elements, e.g. the selling activity is of the type ‘MASsell’. Selling activities are special, because product prices may change over time and have to be used to recalculate income in case yields were not as expected (when using a crop growth model). Also the soil constraints are special, because MPMAS needs to insert the n_j, the soil owned by the household, on the right hand side, and the ‘MASsoil’ type help it to recognize the corresponding constraints. The actual identifier (‘soil’) can be freely chosen, so you could also name it ‘terrain’ or ‘land_class’ or whatever you like. All special types start with‘MAS’. You can also use your own types. These are used to group model elements and help you navigate through large matrices, e.g. using `mqlmatrix` (see chapter 5).

We have used the types product_balance and crop_production, which we have to include into the lists of user defined activity, respectively constraint types, which are located under the corresponding headings at the top of the model structure file.

```
[ACTIVITY TYPES]
crop_production

[CONSTRAINT TYPES]
product_balance
```

orow The ‘orow’ field contains the objective function coefficient of an activity. More specifically for the selling activities, it contains the objective function coefficient of the selling activity in the first year, or in other words the price the household expects to receive in the first year. As shortly mentioned

before, prices may change over time and usually the price they will be able to sell their crop for at the end of the season is not known to the household, while making the production plan at the beginning of the year. The actual price households receive in a specific year is given using fields that have the format `orow_<year>`, so e.g. `orow_2012` contains the actual price for 2012. (The prices used by the households for planning differ and depend on your choice of price expectation submodel, which will be discussed in the next chapter.)

In our case, we spare ourselves the work of typing each year by using the `\#foreach()` function referring to the user-defined variable YEARS. This leads us to an important feature: You can also use MFL functions in field names, and whenever you use a function that results in a list, instead of one field, a whole list of field is created. So if the list YEARS contains all years from 2010 to 2030, we can create the fields orow_2010, orow_2011, orow_2012, ... and thus supply the prices for all years using just one line of code.

The list YEARS has to be defined in the [USER VARIABLES] section of the model data file. We can use the step function to include all years of our simulation. E.g.

```
[USER VARIABLES]
 YEARS    =  #steps(#var(STARTYEAR), #var(ENDYEAR), 1)
```

#table The 'orow' fields receive their values using a table function. We already introduced the table function above, but we have not discussed table functions with more than one key. The table 'prices' could be declared in the model data file like this:

```
 prices   =  (sql, 2) {"SELECT product, year, price FROM tbl_prices"}
```

And the corresponding table prices could contain the following lines:

product	year	price
wheat	2009	170
maize	2009	160
barley	2009	110
wheat	2010	150
maize	2010	170
barley	2010	100
wheat	2011	200
maize	2011	190
barley	2011	130
wheat	2012	130
maize	2012	180
barley	2012	90
...		

As you can see, this table contains more than one line for each crop. So only by using the first column we cannot uniquely identify a row in the table. On the other hand, we want to record only one price for each crop in each year, and so the first two columns are enough to form our row key. In the `\#table` function we thus also need two elements in the key array. For 'orow' , the first argument is again the instance and the second argument is the start year minus one. In other words, we make the very simple assumption that farmers expect to receive the same price in 2010 that they received in 2009.

#this(field,1) For the other orows, we use the function #this with the arguments 'field' and 1. Whenever you have used an MFL expression resulting in a list to define a field name, the #this(field) function refers to the current index in the list. More specifically, #this(field,0) refers to the whole field name,

#this(field,1) to the result of the first function that resulted in an array, #this(field,2) to the second in case you used more than one.

To understand this better, you can imagine `mpmasql` doing an intermediate step. From

```
orow_#foreach(YEARS) #= #table(prices, 3 , [#this(instance),
   #this(field,1) ] )
```

it first creates this

```
orow_2010   #= #table(prices, 3 , [#this(instance), 2010 ] )
orow_2011   #= #table(prices, 3 , [#this(instance), 2011 ] )
orow_2012   #= #table(prices, 3 , [#this(instance), 2012 ] )
orow_2013   #= #table(prices, 3 , [#this(instance), 2013 ] )
...
```

before calculating the field values.

#this(instance)_balance All fields of an activity that have no predefined meaning, are interpreted as the coefficient of an activity in the corresponding constraint. As mentioned before, the full element identifier is formed from the instance identifier, an underscore and the element identifier you used in the element blueprint. So the crop balance constraints of our model will be called wheat_balance, maize_balance and barley_balance. To say, that the selling activity of a crop should get a coefficient of one in the balance constraint of the crop, we wrote:

```
#this(instance)_balance #= 1
```

#activityby Remember that we needed to create a production activity for every combination of soil type and crop. The '-by' suffix enables us to use MFL functions also in the definition of an element identifier, and e.g. to create several similar activities for one instance of a class. However, you need to first supply a fixed part (the 'produce' in our case) before you can add an MFL expression. The activity names are then formed according to the following format:

```
<instance id>_<fixed_element_id>_<MFL>
```

In our example, we get the following activities:

```
wheat_production_on_soil_0
wheat_production_on_soil_1
wheat_production_on_soil_2
maize_production_on_soil_0
maize_production_on_soil_1
maize_production_on_soil_2
barley_production_on_soil_0
barley_production_on_soil_1
barley_production_on_soil_2
```

#this(by,1) Similar to fields, you can use #this(by, 1) to refer to the current index in the by expression. So, the yield and soil balance fields

```
#this(instance)_balance #= -1 * #table(yields, 3, [#this(instance),
   #this(by,1) ] )
#this(by,1)_soil        #= 1
```

become

```
wheat_balance   #= -1 * #table(yields , 3, ["wheat", 0 ] )
0_soil          #= 1
```

for activity `wheat_production_on_soil_0` in the "intermediate" step.

What is now still missing is incorporating the yield and the variable costs tables into the model:

```
 [USER TABLES]
yields   =  (sql ,2)  {  "SELECT crop , soil , yield FROM tbl_yields" }
variable_costs =  (sql ,1) { "SELECT crop , variable_cost FROM
   tbl_variable_cost"}
```

The corresponding two tables in the database could look like this:

crop	soil	yield
wheat	0	6
wheat	1	7.5
wheat	2	8
maize	0	8
maize	1	10
maize	2	9.5
barley	0	5.5
barley	1	6
barley	2	6.2

crop	variable_cost
wheat	700
maize	950
barley	500

3.6.2 Liquidity needs

Apart from soil, we assigned three other resources to the agents: cash, labor and machinery. We still need to connect these to the production opportunities of the household. Cash is used in MPMAS to invest (I_m) into new assets (which is discussed in the investments section below) and to reflect the cash constraints a farm household may face during the season. Farms have to pay a lot of inputs upfront for production activities (phc_c) and receive their revenue only at the end of the season, so either they have enough liquidity to pay all inputs or they have to take credit ($stcrd$) and pay interest. If they have more cash than they need, they can put it on the bank to receive interest for short-term deposits ($stdep$.)

Since you expect to be able to receive at least the amount of cash you paid for inputs as a revenue at the end of the year, you can use the “same” money for investments and pre-financing of inputs. (You can of course make different assumptions here, and change the model structure accordingly.)

We thus have two cash balance equations:

$$\sum_{c,j} phc_c * A_{c,j} + stdep - stcrd \leq n_{\text{cash}} \tag{3.6}$$

$$\sum_{m} eq_m * acq_m * I_m + stdep \leq n_{\text{cash}} \tag{3.7}$$

Farm households cannot take unlimited credit. One assumption may be, that they can use the standing crop as a collateral and receive credit on the upfront input cost. The credit limit restriction could be formulated and rearranged as follows:

$$stcrd \leq \sum_{c,j} phc_c * A_{c,j} \tag{3.8}$$

$$\Leftrightarrow stcrd - \sum_{c,j} phc_c * A_{c,j} \leq 0 \tag{3.9}$$

The liquidity-related model elements cannot really be associated to a class, and they occur only once. Most of them are also special, because they are required by MPMAS. To implement this in the model structure, we introduce a class MASSTANDARD, which contains all elements required by MPMAS, which occur only once. We just assign one instance MAS to the class in the model data file:

```
MASSTANDARD = [MAS]
```

The class implementation would look like this:

```
#class  MASSTANDARD {
  #constraint liqendow {
     name           #= "Liquidity endowment"
     unit              #= Euro
     type           #=    MASliq
  }

  #constraint year1liq {
     name           #= "Year 1 liquidity"
     unit              #=    Euro
     type           #= MASly1
    }
    #constraint  ongliq   {
     name           #= "Pre-harvest liquidity"
     unit              #=    Euro
     type           #= MASong
    }
    #constraint  stcredit {
     name           #= "Short-term credit limit"
     unit              #=    Euro
     type           #= MASscl
  }
  #activity   stcreditact {
     name           #= "Short-term credit"
     unit              #= Euro
     type           #= MASscred
     orow           #= -1 * #var(INTEREST_SHCRD)
     #this(instance)_ongliq     #= -1
     #this(instance)_stcredit   #= 1
  }
  #activity   stdeposit {
     name           #= "Short-term deposits"
     type           #= MASdep
     unit              #=    Euro
     orow           #= #var(INTEREST_SHDEP)
     #this(instance)_liqendow      #= 1
  }
```

```
    #activity   liqtrans {
        type            #= transfer
        name            #= "Transfer Liquidity"
        unit               #= Euro
        #this(instance)_ongliq      #= -1
        #this(instance)_year1liq    #= -1
        #this(instance)_liqendow    #= 1
    }
}
```

MPMAS inserts the households cash reserves into the right hand side of the constraint of type MASliq. There can only be one of these constraints, but we need the value on the right hand side of two equations, that is why we use a transfer activity (liqtrans). Equation 3.6 is marked as MASong, equation 3.7 is marked as MASly1, and equation 3.9 as MASscl. The short-term credit and deposit activities are marked as types MASscred and MASdep. The two variables INTEREST_SHCRD and INTEREST_SHDEP are MPMAS parameters, we have to include into the model data file, e.g.

```
 [MPMAS PARAMETERS]
INTEREST_SHCRD = 0.1
INTEREST_SHDEP = 0.02
```

Then we still need to specify the coefficients for the pre-harvest liquidity and short-term credit limit equations to the definition of the production activities. We simply assume that 80% of variable cost occur before the crop is harvested.

```
 #class CROPS {
    ...
    #activityby production_on_soil #foreach(SOILS) {
        ...
        MAS_ongliq      #= 0.8 * #table(variable_costs, 2 ,
            [#this(instance) ] )
        MAS_stcredit       #= -0.8 * #table(variable_costs, 2 ,
            [#this(instance) ] )
        ...
    }
    ...
}
```

3.6.3 Labor use

Labor is mainly an issue in peak seasons, e.g. during harvest times. Household members can only work a certain amount of time during the maybe 8 days, where wheat can be harvested or the 7 days where barley can be harvested, respectively the 20 days were maize can be harvested. Assuming these three harvest periods do not overlap and neglecting all other labor peaks, which there certainly are, we need to incorporate three equations of the form:

$$\sum_{c,j} wdh_c * A_{c,j} \leq (n_l - OL) * \frac{hd_c}{365} + TL_c \tag{3.10}$$

where wdh_c stands for the number of days needed to harvest a hectare of crop c, n_l is the yearly capacity of labor of the household, which is calculated by MPMAS using the information in the POPULATION_DYNAMICS table and the household member information, and hd_c is the length of the harvest period of crop c in days. The equation also takes into account that household members may have off-farm employment OL and may hire temporary labor TL_c to increase the amount of work that can be done during harvest times.

Because of the multiplication on the right hand side, this equation is a bit more difficult to rearrange, and we need the help of a transfer activity Θ_l and adding an additional equation to the system:

$$\Theta_l + OL \leq n_l \tag{3.11}$$

$$\sum_{c,j} wdh_c * A_{c,j} - TL_c - \Theta_l * \frac{hd_c}{365} \leq 0 \tag{3.12}$$

We did not distinguish different labor types, but to be general, we would have to repeat these equations for each different labor type. We thus put them into the class LABOR, to which we assign our labor type ('farm') as an instance:

```
 [INSTANCES]
LABOR =  [farm]
```

The blueprint for the labor CLASS would look like this:

```
#class LABOR {
   #constraint hhlabor {
      name  #= "Household labor of type: " ~#this(instance)
      unit     #=  "person year"
      type  #=  MASlab
      labgrp   #=  #this(instance)
   }
   #constraintby labor_capacity #foreach(CROPS) {
      name  #= "Labor capacity of type " ~ #this(instance) ~" in
         harvest season of " ~ #this(by,1)
      unit     #= "person days"
   }
   #activityby templabin #foreach(CROPS) {
      name  #= "Hiring in temporary labor of type: " ~
         #this(instance) ~" in harvest seasons of " ~ #this(by,1)
      unit     #= "person days"
      type  #= MASlabtin
      orow  #= -1*  #var(WAGE_PER_DAY)
      #this(instance)_labor_capacity_#this(by,1)   #= -1
   }
   #activity hhlabout {
      name  #= "Hiring out household labor of type: " ~
         #this(instance)
      unit     #= persons
      type  #= MASlabpout
      orow  #=  #var(WAGE_HOUSEHOLD)
      #this(instance)_hhlabor #= 1
    }
    #activity labor_transfer {
      name  #= "Transfer household labor of type:" ~#this(instance)
      unit     #=  persons
      #this(instance)_hhlabor #= 1
      #this(instance)_labor_capacity_#foreach(CROPS)  #= -1 *
         #table(length_of_harvest_period, 2, [#this(field, 1)])
   }
}
```

The household labor capacity calculated by MPMAS is entered on the right hand side of the equation, which is marked as MASlab and whose attribute labgrp is set to the corresponding labor type.

WAGE_PER_DAY and WAGE_HOUSEHOLD can be set in the [USER VARIABLES] section of the model data file: E.g.

```
 WAGE_PER_DAY  =  120
WAGE_HOUSEHOLD =  28000
```

The harvest labor requirements of crop production can then be incorporated like this:

```
  #class CROPS {
   ...
   #activityby production_on_soil #foreach(SOILS) {
      ...
      farm_labor_capacity_#this(instance) #= #table(harvest_labor , 2,
         [#this(instance)])
      ...
   }
   ...
}
```

The two tables harvest_labor and length_of_harvest_period would have to be added to the model, which could look like this:

```
 [USER TABLES]
harvest_labor     = (sql ,1) {"SELECT crop, labor_req_harvest FROM
   tbl_crops_labor"}
length_of_harvest_period      = (sql ,1)   { "SELECT crop,
   length_of_harvest_period FROM tbl_harvest_periods" }
```

crop	labor_req_harvest
wheat	.5
barley	.6
maize	.8

crop	length_of_harvest_period
wheat	8
barley	7
maize	20

3.6.4 Machinery use

While labor, soil and liquidity are special assets, whose handling is controlled by MPMAS using the special type marks, you have to define the relation between all other assets and the equation in the decision problem in a more specific way.

For simplicity, let us assume our tractors have a certain capacity per year, e.g. with one plough you can work 50 ha a year. (Of course, we could also be more specific and distinguish different peak labor periods or different machinery sizes, et cetera.) The corresponding constraint equation for each machinery type would then be:

$$\sum_{c,j} w_{c,m} * A_{c,j} \leq cap_m * n_m \tag{3.13}$$

where $w_{c,j}$ is the number of times you need to work the field with a certain type of machinery for crop

c; n_m is the number of equipments of type m the household owns, and cap_m is the yearly working capacity of that machinery type.

As you need one constraint for each machinery type, we include it into the MACHINERY class and we add the corresponding coefficients to the crop production activities.

```
#class MACHINERY {
    #asset machinery_asset {
            ...
            con    #= #this(instance)_capacity
            multiplier  #= #table(machinery, 4, [#this(instance)])
    }
    #constraint capacity {
        name #= #this(instance) ~" capacity"
    }
}
#class CROPS {
    ...
    #activityby production_on_soil #foreach(SOILS) {
        ...
        table(work, 0, [#this(instance)] )_capacity  #= #table(work, 3,
            [#this(instance), #this(field,1)])
        ...
    }
    ...
}
```

con The 'con' attribute of an assets links it to the constraint of its capacity, you simply have to assign it the identifier of the corresponding capacity constraint.

multiplier The 'multiplier' specifies the capacity per asset, i.e. it would be 50 for our plough. It is read from the fourth column of our machinery table, so we have to make sure our machinery table also contains four columns. The updated table might look like this:

machinery	lifetime	acqcost	capacity
tractor	10	10000	500
plough	6	1000	50
seeder	12	3000	50
harrow	8	760	100
cultivator	8	900	100

And we need to make sure, the additional column is also imported into our model:

```
machinery   = (sql,1) {"SELECT machinery, lifetime, acqcost, capacity
    FROM tbl_machinery"}
```

#table(work, ...) The machinery requirements for the different crop production activities are read from a table named 'work', which might look like this:

crop	machinery	frequency
wheat	plough	1

maize	plough	1
barley	plough	1
wheat	harrow	1
maize	harrow	2
barley	harrow	1
wheat	seeder	1
maize	seeder	1
barley	seeder	1
maize	cultivator	1

As you can see, the different crops have multiple and different field preparation requirements, so this table needs two keys to uniquely identify rows. Also tractors are not mentioned, but we can conclude that for every work you always need a tractor, so the tractor demand of a crop production activity is the sum of all frequencies of the different works. We could add additional lines to the table, but then we would always have to be sure to update it manually, whenever we update the table, because we learned farmers in our study area rather use the harrow twice for wheat and not once. So, rather we add the lines for tractor use with a UNION statement, while importing the table into `mpmasql`.

```
 work =  (sql, 2) {  "SELECT crop, machinery, frequency
                     FROM tbl_work
                   UNION SELECT crop, 'tractor', SUM(frequency)
                     FROM tbl_work
                     GROUP BY crop"
}
```

We already mentioned, that a #table function call with an empty key array as third argument returns all different entries in the first column. This behavior is extended to incomplete key arrays, too. To assign the capacity requirement to our production activities we used the following code:

```
table(work, 0, [#this(instance)] )_capacity  #= #table(work, 3,
   [#this(instance), #this(field,1)])
```

The field name definition contains a table function call that contains less keys that are actually needed to uniquely identify one row. In this case the function returns a list of all different values of the second column, where the first column matches they key provided. In other words, in the imaginary intermediate step the expression would be expanded as follows for a the 'wheat' instance:

```
plough_capacity   #= #table(work, 3,  ["wheat", "plough"])
harrow_capacity   #= #table(work, 3,  ["wheat", "harrow"])
seeder_capacity   #= #table(work, 3,  ["wheat", "seeder"])
tractor_capacity  #= #table(work, 3,  ["wheat", "tractor"])
```

Whereas the expansion for 'maize' would look like this:

```
plough_capacity   #= #table(work, 3,  ["maize", "plough"])
harrow_capacity   #= #table(work, 3,  ["maize", "harrow"])
seeder_capacity   #= #table(work, 3,  ["maize", "seeder"])
cultivator_capacity  #= #table(work, 3,  ["maize", "cultivator"])
tractor_capacity  #= #table(work, 3,  ["maize", "tractor"])
```

3.6.5 Investments

Of course, it should also be possible for households to invest into new or additional machinery. In MPMAS, investment and production decisions are separated into two steps. The production decision

is always taken for the coming cropping season, while an investment decision needs to take a longer perspective into account, it is taken with an average year in the near future in mind. This becomes especially important, when you have assets, whose characteristics change over time like perennial crops. E.g. for perennial crops, the average yield to be expected over the next years may differ considerably from this season's, especially in the first year, when the expected yield is often still zero.

Investing into an asset is an activity in the LP decision problem. Both, investment and production are based on the same decision matrix, except that in the investment decision the coefficients that allow an investment are added to the investment activities. These are

- the coefficient in the capacity constraint of the investment,
- the objective row containing the annualized cost of the investment (debt service and linear depreciation on equity capital),
- the cash demand in the year of investment (the self-financed part of the purchase price)
- the average cash demand for the next years (the fixed equity)
- the expansion of the credit limit, because the self-financed part of the investment can be used as a collateral for short-term credits (the fixed equity)

In the production decision these coefficients are zero, such that the activity is ignored by the household. In the model structure file these coefficients should be empty. The corresponding values are calculated automatically by MPMAS using the long term interest rate parameters (INTEREST_LONGCRD, INTEREST_SHDEP), and the 'lifetime', 'acqcost' and 'multiplier' fields of the asset.

In case you want to test the investment decision problem in standalone mode (i.e. outside of MPMAS, see chapter 5), you will want to have the coefficients that MPMAS calculates in the matrix. For this purpose, `mpmasql` provides the function `\#ifalone`. The first argument to this function is the value used when creating a standalone matrix, the second the one used when preparing MPMAS input files. In this way, you can make sure, the coefficients automatically have the desired value, whenever you switch between conversion modes, without having to update your files by hand.

In the following example, we have used the #ifalone function to show you the formulas MPMAS uses internally to calculate the coefficients. The actual value found in the input file matrix is, however, zero in any case.

```
#class MACHINERY {
   #asset machinery_asset {
      ...
      eqshare  #= #var(EQSH)
      act   #= #this(instance)_invest
   }
   #activity invest {
      name  #= "invest into "~ #this(instance)
      integ #= 1
      ubound   #= 10

      orow  #= -1 * #ifalone(  #var(EQSH) * #table(machinery, 3,
         [#this(instance)]) / #table(machinery, 2,
         [#this(instance)])  + #annuity(#var(INTEREST_LGCRD), (1 -
         #var(EQSH) ) * #table(machinery, 3, [#this(instance)]),
         #table(machinery, 2, [#this(instance)])) , 0)

      #this(instance)_capacity   #= -1 * #ifalone(  #table(machinery,
         4, [#this(instance)]) , 0)
      MAS_ongliq        #= #ifalone( #fvalue(#var(INTEREST_SHDEP),
         #table(machinery, 2, [#this(instance)]) ) * #var(EQSH) *
         #table(machinery, 3, [ #this(instance)] ) , 0)
```

```
        MAS_year1liq       #= #ifalone(#var(EQSH)  * #table(machinery,
           3, [ #this(instance)] ) , 0)
    }
}
```

act The investment activity needs to be connected to an asset, so whenever the activity is chosen by the household (i.e. greater than zero in the solution), MPMAS knows that it has to add this asset to the household's endowments and account for the upfront payment and debt service in the balance sheet of the household. This is achieved by assigning the identifier of the activity to the field 'act' of the asset. This field is a required field, i.e. you need to have an investment activity for every asset, even if you do not want to allow investments into that asset, otherwise you will get an error. (You can set a multiplier of zero, an unbound of zero, or use the innovation diffusion submodel to avoid investments into the asset.)

integ All our machinery assets are indivisible, and thus we also need to make sure that their investment activity can only take integer values. This is achieved by setting the 'integ' attribute of an activity to one.

ubound Usually, activities have an upper bound of $10^3 1$, i.e. they can take near infinite values, if the constraints allow this. Integer activities should be constrained to some lower, realistically expectable maximum value, in order to help the mixed integer solver find a solution. This can be done using the 'ubound' attribute. In our case, we expect it to be highly unrealistic that any household might want to buy more than 10 pieces of the same machinery type at the same time.

#var(EQSH) In section 3.5.4, we had explicitly set the equity share of machinery investments to 25%. As we use the equity share in several of the calculations, it is a good idea to define a variable for it and set the value once for all. It is good practice to use variables, whenever a value is used in more than one place. This ensures consistency and saves you work, whenever you change a parameter value later on.

We add this variable to the [USER VARIABLES] or the [GLOBALS] section of the model data file.

```
 [GLOBALS]
EQSH  =  0.25
```

3.6.6 Hiring machinery

Two of our example farms do not have a tractor and one does not have any machinery at all. Investing into machinery is often not profitable, and households hire machinery when needed. To complete our introductory model, we should also consider this in the model structure.

We simply add a hiring activity for each machinery type to our model:

```
 #class MACHINERY {
...
   #activity hire {
      name  #= "hire " ~#this(instance)
      unit     #= ha

      orow     #= -1 * #table(machinery, 5, [#this(instance)] )
      MAS_ongliq  #= 1 * #table(machinery, 5, [#this(instance)] )
```

```
        #this(instance)_capacity    #= -1

    }
...
}
```

The price per hectare is simply added to the machinery table:

machinery	lifetime	acqcost	capacity	hiring_price
tractor	10	10000	500	50
plough	6	1000	50	15
seeder	12	3000	50	25
harrow	8	760	100	10
cultivator	8	900	100	12

```
machinery   = (sql,1) {"SELECT machinery, lifetime, acqcost,
    capacity, hiring_price FROM tbl_machinery"}
```

As this activity is not linked to any asset, it does not lead to a permanent increase in asset capacity, but only extends the asset capacity for this season.

3.7 Running the model

3.7.1 Creating the MPMAS input files with `mpmasql`

After the model design has been created, the creation of MPMAS input files is started by running `mpmasql` with the name of the conversion control file as an argument. Open a terminal, change into the model directory, and type, e.g. for our example application.

```
mpmasql example.ini
```

`mpmasql` will then write the input files for our example into the subfolder input/dat/ and a commented version of the files into xlsInput/. Further it will create a file called `<prefix>run`, i.e. in our case examplerun, and place it into the model directory. This file is a shell script that you can run to start the simulation. Since it is also merely a text file, you can open it in a text editor and have a look at it. It should look like this:

```
 #!/bin/sh
mpmas -Nexample__
```

The first line just instructs Linux that this is a shell script and the second line is the command that is to be run in the shell, in our case, the MPMAS executable with the option -N, which tells MPMAS to read the input files that start with `example__`. MPMAS automatically expects these to lie in a subfolder called input/dat/.

If you want you can edit the file and e.g. append another MPMAS option to the program call, e.g. -Y5, which tells MPMAS to stop the simulation after five years.

```
 #!/bin/sh
mpmas -Nexample__ -Y5
```

You can also tell `mpmasql` to always append an option to the program call by including a FLAGS entry into the conversion control file. Best placed under a heading [RUN SCRIPT].

```
[RUN SCRIPT]
FLAGS = -Y5
```

3.7.2 Running MPMAS

Especially the first time you try to run MPMAS on a computer, you'll probably need to make sure it is permitted to execute mpmas and the `<prefix>run` script (assuming you have installed the OSL properly). If you get problems with permissions, set the executable permissions, by typing the following in a terminal in your model directory:

```
chmod a+x examplerun
```

No you can simply run the examplerun script,

```
./examplerun
```

and MPMAS will start running and writing output to the out/ subfolder.

3.8 Scenarios

For each scenario you want to run, you have to create a separate set of MPMAS input files. This does however not mean that you have to create new model configuration files for each scenario. `mpmasql` offers an easy-to-use scenario mechanism, which allows to adapt any parameters or variables you have used in your model design according to scenario settings you provide to `mpmasql`.

More specifically you can adapt every entry in the model configuration, any MPMAS parameter, all global and user-defined variables, any instance list and the source string of every MPMAS and user-defined table using the scenario mechanism. So basically you can change everything except the table types and structure, even the model structure if you use your variables and lists wisely in the definition of the model structure.

The basis of the scenario mechanism is a table, which is assigned to the [SCENDEF] field in the conversion control file. It needs only three columns: 'scenario', 'variable', 'value'. Column ordering is significant, column naming is not. The first two fields serve as key, thus if the same combination of scenario and variable appears several times only the last entry is used. Scenario names have to follow the identifier naming rules [A-Z, a-z, 0-9, _]

`mpmasql` will automatically find out what kind of variable is referred to, so please – in general – do not user parameter names if you define your own variables.

In the example (Tab. 3.15), the user-defined variable WAGE_PER_DAY would be set to 140 in Scenario 'scen1' and to 80 in Scenario 'scen2'. The MPMAS parameter SCONEXTRA would be set to 0.6 in Scenario 'scen2', while in Scenario 'scen3' the user-defined list LABPERIODS would be replaced by the list retrieved from a database according to the specified SQL-Statement. Variables not listed for a scenario simply remain the same as in the baseline.

Table 3.15: Sample SCENDEF table

scenario	variable	value
scen1	EQSH	0.2
scen2	EQSH	0.02
scen2	SCONEXTRA	0.6
scen3	LABPERIODS	#sql(SELECT DISTINCT Class2 FROM labperiods)

The second step is to tell `mpmasql`, which scenarios to create by assigning a list of scenario identifiers to the control file entry SCENLIST. If you include a scenario in this list, that does not have any entry in the SCENDEF table, this will simply be equal to the original settings of your model configuration files.

E.g. assuming the sample SCENDEF table is stored in your database as 'tbl_scenarios', the complete control file section could look like this:

```
[SCENARIOS]
SCENDEF = (sql) { "SELECT scenario, variable, value FROM scenarios"}
SCENLIST = (list) { [base, scen1, scen2, scen3] }
```

This would result in four scenarios, one equal to the baseline, the others with the changes as described above. Of course, you can also include less scenarios than specified in your table:

```
[SCENARIOS]
SCENDEF = (sql) { "SELECT scenario, variable, value FROM scenarios"}
SCENLIST = (list) { [base,  scen2] }
```

If you do not specify any scenarios or omit SCENLIST entirely your scenarioname is an underscore. In general, the MPMAS input file names follow the following format:

```
<PREFIX><scenarioname>_<predefined MPMAS file name>.dat
```

With four scenarios as above, your examplerun file, will now look like this:

```
#!/bin/sh
mpmas -Nexamplebase_
mpmas -Nexamplescen1_
mpmas -Nexamplescen2_
mpmas -Nexamplescen3_
```

So, if you now run examplerun, MPMAS will simulate one scenario after the other and always write the output files to out/.

3.9 Output analysis

MPMAS writes the simulation results into plain text files, which are similar as the input files and hard to read, because they contain no commentary and easily grow very large, especially if your decision model is large. To work and analyze the results, it will usually be necessary to import the results into a statistical software package. `mpmasql` automatically generates import scripts for the two statistical software packages `stata` and `R`. The R script can also be used to transform the data into more conveniently formatted, tab-separated text files, which can then be imported into your database, spreadsheet software or any statistical software package of your choice.

3.9.1 Defining output variables

The MPMAS output contains for each agent and simulation period, among others, the updated solution and the right- and left hand side capacities of the last decision model solved, i.e. the production plan, as well as a number of performance indicators (e.g. income, cash flow, equity).

Especially, in large decision models, many variables (e.g. transfer activities) will not really be of interest to you. Further, often the variable of interest is split over several activities, e.g. to get the total area grown with a certain crop, you may have to sum over all the different activities representing the different combinations of terrain types and management options associated to that crop. The

importing scripts can automatically do that for you, if you indicate to which aggregate variable an activity belongs.

This is done using user-defined fields, which are marked with `#resultgroup>`. E.g. in our example, the following would give you three result variables for each agent: crop_wheat, crop_maize and crop_barley, each containing the total area grown with the respective crop, summed over all soil types:

```
#activityby production_on_soil #foreach(SOILS) {
      ...
      #resultgroup> crop   #= #this(instance)
      ...
   }
```

You could additionally create a variable for each soil type, giving you the total area of crops grown on each soil type, e.g.

```
#activityby production_on_soil #foreach(SOILS) {
      ...
      #resultgroup> crop   #= #this(instance)
      #resultgroup> cultivated_area_of_soil  #= #this(by, 0)
      ...
   }
```

The above will create the three crop variables as above, as well as the three variables cultivated_area_of_soil_0, cultivated_area_of_soil_1, and cultivated_area_of_soil_2. Of course, you can also create variables from capacities by adding `#resultgroup>` fields to constraints.

3.9.2 Importing results into stata

Whenever you create a new input file set for MPMAS, `mpmasql` will also create an import script for stata. It is named `<prefix>_import_to_stata.do` and placed into the same folder as the `<prefix>_run` script.

If you use the standard folder setup you can just open stata, change to the MPMAS root directory and run the script there. You will then find one stata file per scenario in the subfolder Stata/combined/, containing performance data, RHS, LHS, solution and all user-defined variables for each agent and period.

If you use a different folder setup or want the files to be written into different places, you can change the first part of the script accordingly or let `mpmasql` directly create the script according to your needs, by setting the corresponding entries in the [STATA SCRIPT] section of your control file (see B.1).

3.9.3 Importing results into R

Like for stata, whenever you create a new input file set for MPMAS, `mpmasql` will also create an import script for R. It is named `<prefix>_import.R` and placed into the same folder as the `<prefix>_run` script.

You can run it by typing `R --vanilla < <prefix>_import.R` at the terminal, after changing into the root directory of your MPMAS version. [3] By default, the data is stored in the file `<prefix>.Rdata`, located in the subdirectory `R/data/workspaces/`. You can open it in R using the `load` function, after that, your workspace will at least contain two data frames per scenario, one containing the combined performance information, RHS and solution for each agent and period (_combined) and one containing the combined sets of user variables (_userset). Further, there will one data frame for each set of user-defined variables and scenario.

[3] The `--vanilla` just tells R not to neither restore nor save any data into the workspace of the current directory.

You can fine tune the behavior of the script by adapting it manually yourself, or let `mpmasql` do that for you according to the entries of the [R SCRIPT] section of the conversion control file (see B.1).

3.9.4 Transforming results into text files/databases

If you do not want to use neither R nor stata to analyze your data, you may still prefer to transform the raw MPMAS output to more conveniently formatted text files, including your user-defined variables. If you set the `R_TO_TAB` entry in the control file to 1, you `mpmasql` will create an R import script that saves the same setup you would find in your R workspace as tab-separated text files in the subdirectory `R/data/tab/`. You run it the same way as described above. By default, the .RData output is also created, but you can omit this by setting the control file entry `R_TO_WORKSPACE` to zero.

Chapter 4

Model design with mpmasql: MPMAS submodels and features

The model setup described in the previous section is a relatively simple one, it does not contain agent-agent interactions, no real interactions with the environment and only very simple kinds of assets. MPMAS contains complex features for all of these, and in this section the ones implemented with `mpmasql` are presented.

4.1 Perennial crops

Perennial crops are crops, which once planted occupy land for a certain lifespan longer than a year. Yields, production costs and input requirements may change with age. Planting them is an investment decision.

Within MPMAS, perennial crops are represented by several model elements: As perennial crops are investments, there has to be an asset, an investment activity and an endowment constraint. As they are production processes, there is a growing activity and because the same perennial plant can be managed in different ways there are switching activities between different management practices. Further, as yields and input requirements change with age there is a perennial file entry, which supplies the respective values for each age.

Every perennial is equal in this respect and thus perennials of course in principle form a decision class. However, `mpmasql` saves you the work of specifying each model element individually by providing a special type of element: the perennial.

You can activate the perennial crops model by setting the PERMCROP_MODEL entry under [SUBMODELS] in the configuration to 1.

Perennial elements are specified in the .model structure file, in a similar way as definitions for basic model elements (activities, constraints, assets). As usual there is also a `perennialby`-element.

Let's look at an example:

```
#class PERENNIALS {
   #perennialby trees #foreach(SOILS) {

      'shared fields
      name  #= #table(perennials, 0, [#this(instance)]) ~ " on soil
         #this(by,0)"
```

```
        lifetime #= #table(perennials, 4, [#this(instance)])
        acqcost  #= #table(perennials, 5, [#this(instance)])
        terrain  #= #table(perennials, 7, [#this(instance)])
        $$managementswitch    #=   #table(perennial_switching,
           2,[#this(instance)])~"_trees"

        'specific fields
        #asset>perm        #=    #table(perennials, 3,
           [#this(instance)])_balance
        #asset>innogrp     #=    default
        #asset>landreq     #= 1

        #dynamic> perennial_yield      #= #table(perennial_yields, 3,
           [#this(instance), #this(dyn)])
        #dynamic> perennial_preharvest_cost #= 0
        #dynamic> perennial_harvest_cost #= 0
        #dynamic>  farm_labor_capacity_#foreach(CROPS) #=
           #table(perennial_labour, 4, [#this(instance), #this(dyn),
           #this(field,0)])
        #investment> #this(by,0)~"_soil"  #=   1
        #investment> #this(by,0)~"_invsoil"  #=   1
        #production> #this(by,0)~"_soil"  #=   1
    }
}
```

There are so-called shared fields, which are used by several of the created model elements, and there are specific fields, which are used by only one of the elements. Shared fields are `name`, `lifetime`, `acqcost`, `terrain` and `\$\$managementswitch` and except for the later these are already known from the basic model elements. Specific fields are marked with a `#` `>` surrounding the subelement they belong to.

The subelement `asset` contains all fields for the network object. (Including `perm`, which refers to the yield balance constraint of the crop). The `act` and `con` fields are filled automatically by `mpmasql` and should not be given.

Subelements `production` and `investment` refer to fields and coefficients of the growing and investment activity that do not change with age.

Those coefficients that do change with age are included in the subelement `#dynamic>`. Three fields are obligatory: `perennial_yield`, `perennial_preharvest_cost` and `perennial_harvest_cost`.

`perennial_yield` should contain the relative yield at a certain age, as a share of the maximum yield that can be obtained in the best producing age.

Every other field is interpreted as a coefficient for a constraint. Value definitions for these fields can either result in a scalar, then they will be constant over all ages for this specific instance, or they can be a function that contain the placeholder `\#this(dyn)` for the age, and result in different values for different values of `\#this(dyn)`.

Note that you do not have to specify a value for each age. Missing values will be filled automatically by taking the value of the next higher age for which a value has been supplied. So for example, if your perennial crop produces a proportional yield of 0 in the first five years, 0.5 in the next five years and 1 over the rest of its 25 years of use life, it would be enough if the table perennial_yields in the above example would contain three entries for this instance (5; 0), (10; 0.5) and (25; 1). A value for the highest age has to be supplied, however.

During simulation MPMAS will look into the dynamic element to select the value for the current age of the perennial plantation and enter it into the matrix. (Actually, it does an area-weighted average over all currently present age levels.)

Once grown, the management applied to a perennial crop may sometimes be changed during the course of its life. For each type of management, a separate perennial object has to be introduced into the model, because age-specific yield or input requirements usually differ. However the agent later does not have to cut down his plants and replant new ones to switch management practice (e.g. change irrigation practice or fertilizer use), but he can just use one perennial object as if it was the other. This is called switching in MPMAS. Switching should of course only be possible where it makes sense, e.g. from furrow- to drip-irrigated apple, but not from apple to pear trees. So for each perennial object you can supply a list of those perennial objects from which you can switch to the present one. This list should be given as a value to the `$$managementswitch` field (The `$$` indicate it is an array field). For each entry in the list an activity is implemented automatically, which has the coefficients of the new management practice, except for the endowment constraint where the coefficient is placed in the endowment row of the original management practice.

Perennial objects need a future yield constraint and a future selling constraint. Future selling activities should be of type "MASfuture" and are otherwise similar to selling activities. Future yield constraints have to be placed directly after the current yield balance row. To achieve this they have to be of the same type as the yield balance row and should have an identifier that is automatically sorted directly behind the yield row (remember that matrix elements are sorted by type and then alphabetically by identifier).

```
#class PERENNIALS {
   #activity  sell_future {
      name     #= "Future selling of  "~#this(instance)
      type        #= MASfuture
      orow     #=  #table(prices, 3, [#this(instance),
         #var(STARTYEAR)])
      orow_#foreach(YEARS) #=  #table(prices, 3, [#this(instance),
         #this(field, 1) ])
      #this(instance)~"balance_future" #= 1
   }
   #constraint  balance_future {
      name     #= "Future balance of "~#this(instance)
      type        #= product_balance
      unit        #= kg
   }
   #constraint  balance {
      name     #= "Balance of " ~ #this(instance)
      type        #= product_balance
      unit        #= kg
   }
   #activity sell {
      name     #= "Sell "~ #table(perennials, 2, [#this(instance)])
      type        #= MASsell
      orow     #=  #table(prices, 3, [#this(instance),
         #var(STARTYEAR)])
      orow_#foreach(YEARS) #=  #table(prices, 3, [#this(instance),
         #this(field, 1) ])
      #this(instance)_balance #= 1
   }
}
```

Perennials block plots. These plots cannot be used for other crops, and especially you cannot plant other perennial crops on them. To make sure this does not happen, a special constraint of type MASinv has to be added to the MILP for each soil. MPMAS will enter the soil that has not been blocked by perennials on the RHS of these constraints during the investment stage. Each perennial investment activity should have a coefficient of 1 in the respective constraint.

```
#class SOILS {
```

```
    #constraint invsoil {
          name  #= "Investments on terrain " ~ #this(instance)
          unit     #=    ha
          type  #= MASinv
     }
}
```

Often perennial crops have a long gestation period, in the first years, during which they do not yet give full yield, they produce a negative cash flow. To make sure the agent does not go bankrupt, because the cash demand of the newly planted perennials plus the foregone revenue from the crops he had previously grown on the soil surpass his liquidity, a further constraint is added. MPMAS will place the cash surplus of the previous year to the rhs as a basis for the estimate of future cash flows. During simulation MPMAS will set the coefficient of the investment activity in this constraint to the highest cash demand for a year in the gestation period and add a certain amount on top accounting for the loss in positive cash flow from the previous use of the plot. In effect, this constraint restricts investment such that the sum of the highest cash demand during the gestation period of all perennials planted during this season is lower than last year's cash surplus minus the foregone cash flow of the crops that cannot be produced anymore (λ_s). The later is not calculated endogenously, but given exogenously as a kind of shadow cash surplus for each soil type.

```
#class MASSTANDARD {
    #constraint invcash {
        name #= "Future expected cash surplus"
        type #= MAScas
    }
}
```

Besides this, another option is to restrict investment into perennial crops to only a share of the currently free plots. The free plots are then divided by a physical investment bound. Both, physical investment bounds and shadow cash surplus can be specified for each soil type in the SEGBOUNDS table given in the [MPMAS TABLES] section of the data file (see Tab. B.26 in section B.3). The default values are 1 for the physical investment limit and 0 for the opportunity cost.

4.2 Livestock

The livestock model is a pretty detailed, individual based representation of livestock production including simulation of offspring and aging, and use of livestock as a liquidity reserve, which is especially useful to represent small-holder decision making in developing countries. For other cases, you would probably prefer an aggregate, herd-based representation, which you should be able to implement with the basic MPMAS assets.

The livestock is activated by setting the LIVESTOCK_MODEL entry under [SUBMODELS] in the model configuration to 1.

The implementation in `mpmasql` is similar to perennial objects. There is a compound livestock element (`#livestock`, resp. `#livestockby`) which is automatically transformed into a number of model elements (an investment activity, a maintenance activity, a sell-at-start-of-period and a sell-at-end-of-period activity for each age, an overall endowment constraint and one for each age, a disinvestment (sell-all) constraint and a livestockfile entry). All of these activities are doubled, because each is generated once for each sex of the livestock type. You can differentiate field and value definitions by sex using the special variable `#this(sex)` (e.g. females should produce offspring, while males should not.)

Like perennials, livestock elements contain shared fields, which are used by several of the sub-elements, and there are specific fields, which are used by only one of the generated elements. Shared

fields are name, lifetime, acqcost, unit, purchase_age, salesactivity_meat. The first five should be self-explaining, salesactivity_meat should refer to the identifier of the selling activity of the meat of the animal.

Specific fields are again marked with a # > surrounding the subelement they belong to. The subelement #asset> contains all fields for the asset (Fields act, con, and div are set automatically) . Subelements #maintain> and #investment> refer to fields and coefficients of the growing and investment activity that do not change with age. Subelements #salesstart> and #salesend> refer to the sell-at-start-of-period and sell-at-end-of-period activities. As all of these are integer activities it is recommendable to set at least a reasonable upper bound for them. Subelement #disinvest> refers to the disinvestment (sell-all) constraint.

The dynamic part of the livestock element is split up into several field group (dyn_liveweight, dyn_offspring, dyn_sales, dyn_internal, dyn_liquidity, dyn_land, dyn_labor). Value definitions for these fields can either result in a scalar, then they will be constant over all ages for this specific instance, or they can be a function that contains #this(dyn) as a variable referring to a specific age.

dyn_liveweight should contain one field, whose field definition should contain the identifier of the balance constraint for the meat of the animal and whose value definition should provide the liveweight of the animal at each age.

dyn_offspring should contain one field, whose field definition should contain the identifier of the balance constraint of the offspring and the value definition should result in the number of offspring produced by an individual of this livestock type, sex and age.

dyn_sales should contain one field per marketable product (e.g. milk, wool) that is produced by an individual of the respective livestock type, sex and age. The field definition should contain the identifier of the balance constraint of the product and the value the amount produced.

dyn_internal should contain one field per internal consumable used or produced (e.g. manure, feed) that is produced by an individual of the respective livestock type, sex and age. The field definition should contain the identifier of the balance constraint of the consumable and the value the amount produced (positive value) or consumed (negative value).

dyn_liquidityshould contain one field, whose field definition should contain the identifier of a liquidity constraint (usually MAS_ongliq) and whose value definition should indicate the required liquidity.

dyn_land may contain as many fields as you have soil types in your model though usually less. The field definition should contain the identifier of a (soil) constraint and the value definition should indicate the required amount.

dyn_labor may contain **one** field, whose field definition should contain the identifier of a labor constraint and the value definition should indicate the required amount of labour. (If you need more labor constraints put them as internal consumables).

4.3 Crop growth models

One of the special features of MPMAS is its ability to be coupled with crop-growth models. So far, we have assumed that plant growth occurs as planned, but in reality crop yields are often depending on natural conditions.

4.3.1 CropWat

CropWat is an internal MPMAS implementation of the FAO56 model for crop growth under water deficit. To simulate irrigation, it has to be combined with the EDIC hydrology model, however, if only rainfed crops are modeled, the EDIC model is not necessary. It is activated by setting the CROP_MODEL entry under [SUBMODELS] in the model configuration to 2.

```
[SUBMODELS]
CROP_MODEL  =  2
```

In the MPMAS TABLES section of the model data file, time series for precipitation and potential evapotranspiration have to be provided for each sector as well as values for the initial expectations for both (Required table columns are given in Tab. B.8, Section B.3). Optionally, you can adapt the formula used to calculate effective rainfall, i.e. the amount of rain that can be used by the plant, from observed rainfall. The default is the USDA formulation, alternatively you can specify the coefficients of a quadratic equation that includes precipitation, the crop water demand and their interaction as independent variables.

The evapotranspiration values are reference values, which are transformed into plant-specific evapotranspiration by multiplying them with the respective plant-specific coefficient for the corresponding months (k_c). These k_c are entered for each cropping activity in the model structure file, together with other essential values needed for the CropWat model. For this purpose, a new field marker `#cropwat>` is introduced that can be placed inside activities and perennial objects. Under this marker, the fields listed in Tab. 4.1 should be specified.

Table 4.1: Required fields with the `#cropwat>` marker

Field	Description	Values
#cropwat>irrigation_method	irrig_id as specified in EDICIRRIG-METHODS, or -1 for rainfed	integer
#cropwat>k_y	yield sensitivity to water deficit	number
#cropwat>ipg	irrigation priority group (-1: unirrigated, 0 highest priority)	integer
#cropwat>seasonality	seasonality (2: annual)	integer (0..2)
#cropwat>yield_potential	potential yield without deficit	number
#cropwat>yield_start	agent yield expectation in first year	number
#cropwat>kc_#foreach(MONTHS)	kc value of crop for each months of the irrigation season	number
#cropwat>yield_constraint	identifier of the yield constraint of the crop	identifier
#cropwat>sales_activity	identifier of the sales activity of the product	identifier
#cropwat>soil	soil type	integer
#cropwat>zerolab	whether the activity is a zero labour activity (needed for spatial allocation)	0-1

Further, you need to define one water balance constraint for each irrigation month of type `MASwater`, which is best placed in the class `MASSTANDARD`. E.g.:

```
#class   MASSTANDARD {
  ...
 #constraintby watersupply #foreach(MONTHS) {
   type  #= MASwater
   name  #= "Water availability in month #this(by,0)"
   unit  #= "m^3/s"
 }
 ...
```

```
}
```

4.3.2 External crop growth models

MPMAS provides an interface to communicate with external crop growth models using the TDT-library. MPMAS sends a map with a numeric ID of a cropping activity in each cultivated cell (or -1 for an uncultivated cell), and expects the crop model to return a map with the corresponding yield in each cell.

The external crop growth model interface is activated by setting, both, the CROP_MODEL as well as the LANDSCAPE_MODEL entry in the model configuration to 3.

```
[SUBMODELS]
CROP_MODEL      =  3
```

Each cropping activity, whose yield is to be calculated using the crop growth model, needs to include additional fields, which are marked with `#external_crop_growth>`.

```
#class CROPS {
   #activityby production_on_soil #foreach(SOILS) {

      #external_crop_growth>link #= #table(xcwlink,3, [
         #this(instance), #this(by,0)])
      #external_crop_growth>soil #= #this(by,0)
      #external_crop_growth>crop_type  #= 1
      #external_crop_growth>priority   #= 1
      #external_crop_growth>product1_sales_activity   #=
         #this(instance)_sell
      #external_crop_growth>product1_balance #=
         #this(instance)_balance
      #external_crop_growth>product2_balance #= -1
      #external_crop_growth>initial_yield_product1 #= #table(yields,
         3, [#this(instance), #this(by,1) ] )
      #external_crop_growth>initial_yield_product2 #= 0
   }
}
```

`link` is the ID to be put into the map for the activity. The crop growth model, or a wrapper, is supposed to understand the ID and transform it into input for the plant model.

`product1_sales_activity` The identifier of the sales activity of the product (only necessary if you do not use an explicit harvest/consumption decision, see Sec.4.3.4).

`product1_balance` The identifier of the balance constraint of the first product.

`product2_balance` The identifier of the balance constraint of the second product (only possible if you use an explicit harvest/consumption decision).

`initial_yield_product1` The expected yield of the first product for the first simulation period

`initial_yield_product2` The expected yield of the second product for the first simulation period

`priority` is an integer indicating, how close the area should be placed to the farmstead of the agent, the smaller the number the closer

`crop_type` indicates whether the activity is seasonal (0), annual (1) or perennial cop (2), or whether it is a fallow activity(3) or deactivated (-1).

At least one activity per soil type should be marked as fallow activity and this activity should not require any limited resources (e.g. labor). This is important, because in order to create a map of cropping activities, MPMAS has to round the crop areas to full integers in order to distribute them over the map cells. To avoid any conflicts with resource constraints, MPMAS will only round towards zero, and allocate the remaining area to a fallow activity, which should not require any resources.

```
#class FALLOW {
   #activityby on_soil #foreach(SOILS) {
      #external_crop_growth>link    #= #table(xcwlink,3, [
         #this(instance), #this(by,0)])
      #external_crop_growth>soil    #= #this(by,0)
      #external_crop_growth>crop_type  #= 3
      #external_crop_growth>priority      #= 1
   }
}
```

4.3.3 Exogenous yield time series

Instead of an external crop growth model, you can also provide a time series of yields for every period and cropping activity. The external crop growth interface looks the same as in the previous section, you only have to add a value for each product and simulation period as shown below.

```
#class CROPS {
   #activityby production_on_soil #foreach(SOILS) {
      ...
      #external_crop_growth>yield_product1_#foreach(YEARS)  #=
         #table(yield_time_series, 4, [#this(instance), #this(by,1),
         #this(field,1) ] )
      #external_crop_growth>yield_product2_#foreach(YEARS)  #= 0
      ...
   }
}
```

You then have to run MPMAS with the flag -T16, to make it read the yields from the input files instead of waiting for input from the crop growth model. You can have `mpmasql` do that for you by including -T16 into the FLAGS list in the control file.

```
FLAGS   =  -T16 ...
```

4.3.4 Adaptation of production decisions after harvest

Whenever a crop growth model is used, the harvested yields and as a consequence the income obtained by the agent will most probably differ from the agents original plan. As long as the farm household only sells the crops obtained, MPMAS will simply correct the farm household's revenue using the yield obtained and the price information of the associated selling activity.

This simple approach only works as long as the farm household does not consume the produced crop itself or uses it as an input for another production process, e.g. as fodder for animals or feed-stock for a biogas plant. In this case, the original production plan will have to be revised, once the production result is known, e.g. in order to buy additional food or feed from the market. If you are using the advanced consumption model, this is part of the consumption decision, in other cases you have to explicitly request a revised harvest decision by setting the corresponding entry in the model configuration to 1.

```
HARVEST_DECISION =  1
```

At the end of the season the farm household can of course not undo his planting decisions and therefore all cropping activities and all other activities that have to be considered irreversible at the end of the season have to be fixed at the values determined in pre-season decision (except the fallow activities in order to allow for rounding). This is achieved by setting the special field 'consfix' to 1 for all activities that shall be fixed.

```
#class CROPS {
    #activityby production_on_soil #foreach(SOILS) {
        consfix  #= 1
    }
}
```

4.3.5 Yield expectations for crops not grown

As the crop yield is unknown to the farm households at the time of planning, they have to take their decisions based on expectations. The yield expectation for the first year is given as a field in the crop growth model interface, and in the following years are updated according to the rules for expectation formation as described in section **??**. However, while it is rather easy for a farm household to observe all market prices, also for products it did not buy or sell, it is less easy to observe crop yields for crops it did not grow, respectively for combinations of crops, soil types and management options it did not use.

Yield expectations for cropping activities a farm household did not practice can be formed by analogy to crops that have been observed. E.g. if a farm household observes a constant decline of yields of wheat fertilized with pig manure, it might assume that the same decline would also be observed if it used cow manure instead. In order to facilitate such comparisons, you can specify similarity relationships between cropping activities. Similarities can be specified at different priority levels. This is done using the special field marker `#yield_expectation>` with fields named `similarity_<level>`.

```
#class CROPS {
    #activityby production_on_soil #foreach(SOILS) {
        #yield_expectation> similarity_1 #= #this(instance)
        #yield_expectation> similarity_2 #= #this(by,0)
    }
}
```

In the example above, we defined two similarity levels. At level one, all cropping activities of the same crop are considered similar, and at level two, all cropping activities on the same soil. Whenever new expectations have to be formed, let say for wheat production on soil type 0, and the farm household did not practice this activity, it will first look for all other wheat activities it realized. If it did not grow wheat at all, it will check all other cropping activities on soil type 0. If it did practice activities that can be considered similar, it will check whether obtained yields differed from its expectation and update its yield expectation for wheat production on soil type 0, in the same way it updated the yield expectations for similar cropping activities on average.

4.4 Consumption

In reality, farm households cannot put all of their money onto the bank or reinvest it, but they will need a certain amount of money for consumption purposes. In MPMAS, the modeler has the choice between a basic and an advanced three-stage consumption model. By default the basic consumption model is used.

4.4.1 Basic consumption model

The basic consumption model is a simple, Keynesian-style, parametric consumption model. Cash consumption is calculated using the formula

$$SCONMIN * n_{\mathrm{hhm}} + SCONEXTRA * (\pi_{\mathrm{ncs}} - SCONMIN * n_{\mathrm{hhm}}) \quad , \tag{4.1}$$

where n_{hhm} is the number of household members and π_{ncs} is the net cash surplus of the year. Cash consumption thus consists of a minimum consumption needed to maintain the family and an extra consumption that depends on the performance of the household enterprise. If cash reserves of the household are not enough to cover even the minimum consumption, the full cash reserves are consumed. You can however specify a minimum threshold as a share of the minimum consumption (SCONRED) that is the absolutely essential part of the consumption, and if this cannot be satisfied the household will exit.

The parameters of the basic consumption model are specified in the [MPMAS PARAMETERS] section of the data file. For example:

```
[MPMAS PARAMETERS]
SCONEXTRA    =  0.25
SCONMIN      =  10000
SCONRED      =  0.2
```

4.4.2 Advanced three-stage consumption model

The extended three-stage consumption model has been developed to model subsistence farming. It is a parametric model, whose parameters can be estimated econometrically, and which partitions the household income into savings, expenditure for non-food goods and expenditure for different food categories. Food demand is expressed in demand for nutrients and depends on the household size. Food demand can be covered by consuming self-produced goods or by buying food from the market. The model is quite complex and is not going to be explained in full detail here, the full rationale and equations of the model can be found in Schreinemachers [2006]. Here we only describe, how the parameters and elements of the model have to be implemented in `mpmasql`.

Model configuration

Set the CONSTYPE entry in the configuration file to 1.

The three-stage consumption model respects the nutrition requirements of household members. The configuration entry `HHNUTRIENTS` provides a list of identifiers of these nutrients and your HOUSEHOLD_DYNAMICS table should contain columns at the end indicating the nutrient requirements for each sex and age of a kind of population member. At least one nutrient has to be included.

Example:

```
[CONSUMPTION]
CONSTYPE    = 1
HHNUTRIENTS = [energy, protein]
```

MPMAS PARAMETERS

Table 4.2 lists the parameter names used to enter the estimated coefficients and the widths of linearization segments for the savings, food-nonfood expenditure and food category expenditure functions. All

are required and none has a default value, and all should be included in the MPMAS parameters section.

Table 4.2: Parameters of the

Field	Description
XC_SAV_INCSEGS	List with the widths of the income segments (except first) of the savings function
XC_SAV_COEF_INC	Estimated coefficient of income in the savings function
XC_SAV_COEF_INC2	Estimated coefficient of squared income in the savings function
XC_SAV_COEF_HH	Estimated coefficient of household in the savings function
XC_SAV_COEF_CONST	Estimated constant in the savings function
XC_FNF_COEF_CONST	Estimated constant in the food-non-food expenditure (share) function
XC_FNF_COEF_EXP	Estimated coefficient of total expenditure in the food/non-food expenditure (share) function
XC_FNF_COEF_HH	Estimated coefficient of total expenditure in the food/non-food expenditure (share) function
XC_FNF_EXPSEGS	List with the widths of the total expenditure segments in the food/non-food expenditure (share) function
XC_FOO_FEXSEGS	List with the widths of the food expenditure segments in the food category expenditure(share) function

Example:

```
 'Savings function coefficients
XC_SAV_COEF_INC   = 0.3197431
XC_SAV_COEF_INC2  = 9.09E-006
XC_SAV_COEF_HH    = -89.21698
XC_SAV_COEF_CONST = -5975.1

'Income segementation into five segements
XC_SAV_INCSEGS    =  [800, 2100, 3700, 7000, 34000]

'FOOD/NON FOOD
XC_FNF_COEF_CONST = -56.76446
XC_FNF_COEF_EXP   = 0.8757261
XC_FNF_COEF_HH    =-0.6778122

' Segmentation of food non food expenditure
XC_FNF_EXPSEGS    = [4500,  6800,   9800, 15000, 53000]
XC_FOO_FEXSEGS    = [3814, 5775, 8259, 12044, 42860]
```

Model structure

It is best to simply copy the class MASCONSUMPTION provided in the appendix, Sec. B.4.3. This class contains most of the necessary elements, which you can automatically adapt to your implemen-

tation using the settings you provide in the data file. You also need a different MASSTANDARD class, which can be found in Section B.4.3.

Apart from the elements in this class you should add a self-consumption activity for all of those products which can be self-consumed for covering household food needs. These activities should be of type `MASfoodconsume` have a field `#consumption> category`, which links it to one of the food categories in the MASCONSUMPTION class. For example:

```
#class CROPS {
   #activity consume_own {
      name      #= "Consume own " ~ #this(instance)
      type      #= MASfoodconsume
   ...
      #this(instance)_balance #= 1
   ...
      CONS_hhnutrient_supplyc_#foreach(HHNUTRIENTS)   #=
         #table(products_nutrients, 3, [#this(instance),
         #this(field, 1)])
   ...
      #consumption> category  #= #table(products_food_categories,  2,
         [#this(instance)])
   }
}
```

When using the consumption model, the decision problem is solved three times: once for investments, once for the production decision at the beginning of the season, and once for the harvest and consumption decision at the end of the season. The pre- and post-season MILPs look similar, only actual prices and actual yields are inserted instead of expectations. Further, certain parts of the solution cannot be changed anymore at the end of the season, e.g. especially the crop production activities (except fallow) should be fixed at their original values. To indicate, which activities should be fixed in the consumption stage, you need to add a `consfix` attribute to the corresponding activities and set it to one. MPMAS will then set the lower and upper bound of these activities to the solution of the pre-season solution. (Note: Those activities marked as no-labor activities in the crop growth section, should not be fixed, otherwise the MILP might become infeasible.)

Any other change of coefficients between production and consumption stage can be indicated using the `#harvest>` subelement. E.g. if you used security equivalents to calibrate your model these should be set to one for the consumption stage as shown below.

```
#class CROPS {
   ...
   #activityby production_on_soil #foreach(SOILS) {
   ...
      consfix = 1
   ...
   }
   #activity sell {
      #this(instance)_balance #= 1.2
      #harvest>#this(instance)_balance #= 1

   }

}
```

Data file

To make the MASCONSUMPTION class work you should include the following entry in the [USER VARIABLES] section of your data file. It just sets the name of the monetary unit used in the definition of MASCONSUMPTION. Under [INSTANCES] you should include one instance CONS for the class MASCONSUMPTION.

```
[USER VARIABLES]
CURRENCY = <currency name>
UP_CREDIT_DEFAULT = <utility penalty for defaulting on credits>
MAX_DEFAULT     =  <maximum credit that can be defaulted>

[INSTANCES]
MASCONSUMPTION = CONS
```

Most work will however go into providing data for the tables used in MASCONSUMPTION in the [USER TABLES] section. These tables are described in Tab. 4.3. You can, of course, adapt naming of tables and columns to the structure of your database. Unless you want to adapt the MASCONSUMPTION class itself (you are of course free to do so if you dare), you should however provide columns with the indicated content in the given order.

Table 4.3: TABLES for MASCONSUMPTION

Column	Description
	foodcats
food category id	Id for the food category
name of of food category	Name of the food category
unit of food category	Unit of the food category
xc_foo_const	Constant term in the LA/AIDS expenditure function for this food category
xc_foo_coeff_expsi	Coefficient of expenditure/price index in the LA/AIDS expenditure function for this food category
xc_foo_coeff_hh	Coefficient of household size in the LA/AIDS expenditure function for this food category
budget_share	observed budget share of the food category
	crossprice
food category 1 id	id of first food category
food category 2 id	id of second food category
coefficient	Coefficient of price of second category in the LA/AIDS expenditure function for this food category
	products_food_categories
product id	id of product
food category id	id of food category the product belongs to
	food_category_prices
food category id	Id for the food category
year	model year
price	price in given model year
	food_category_nutrients
food_category_id	id of food category
hh_nutrient	id of nutrient
quantity	nutrient content
	products_nutrients
product_id	id of food category
hh_nutrient	id of nutrient
quantity	nutrient content
	utility_penalty
hh_nutrient	id of nutrient
penalty	objective function penalty for deficit

4.5 Harvest decision

If you do not want to implement a full advanced consumption model, but do want to let farmers react to the results of the harvest, e.g. by buying more animal feed if the hay harvest was worse than expected, you can let agents solve a second production MILP after harvest. Consumption is determined as usual using the basic consumption model.

The harvest decision is activated by including the following line in the model configuration:

```
HARVEST_DECISION = 1
```

This second MILP is solved after harvest. Compared to the pre-season production decision, it will contain actual yields and actual prices instead of expectations. Like for the consumption stage, you can use the `consfix` attribute to fix activities that can't be changed at the end of the season (you should at least fix all land use activities except fallow), and you can use the `#harvest>` subelement to change any coefficient that should differ from the pre-season values.

4.6 Hydrology

4.6.1 EDIC: Sector-based Hydrology

The EDIC model is a simple sector-based node-link hydrology model used in the Chilean case. It is hard-coded into the MPMAS executable and interacts with the CropWat model, which simulates crop growth under water deficit and monthly irrigation decisions. Here we only point to the necessary entries for `mpmasql`, consult a full description of the EDIC model for further information. (Note that the random water rights allocation is not yet implemented in `mpmasql`.)

You can activate the EDIC model by setting the LANDSCAPE_MODEL entry under [SUBMODELS] in the cfg-File to 1. Second, you should set the VINFLOWS variable to two, third you should activate IRRIGATION.

```
[SUBMODELS]
LANDSCAPE_MODEL = 1
IRRIGATION = 1
VINFLOWS = 2
```

To use the EDIC model you will have to provide additional input in your cfg-file. First, your `REGION`-table needs additional columns, as shown in Tab. B.22. Second, you need the additional MPMAS TABLES described in Tab. B.9, Section B.3. (Note that months should be specified as calendar months, `mpmasql` automatically transforms these into irrigation season months based on the `SEASONSTART` and `SEASONEND` values given in the `[TIME]` section.)

4.7 Fine-tuning of OSL solver

The OSL solver is a very powerful tool to solve mixed-integer programming problems. Apart from the branch-and-bound and LP solvers, it offers a number of preprocessing tools that help to improve the efficiency of the main solver. These preprocessing tools may be combined in very different constellations.

Unless you choose a quadratic problem, `mpmasql` assumes you want to use the costumizable MPMAS solver mode 4 (modes 1-3 can be considered outdated and obsolete). The general idea behind mode 4 is that you have several attempts to solve the MILP. In each attempt (except the last) a solution is searched using the specified tool. The attempt ends if either the optimal solution has been found,

or the attempt reaches specified limits on the number of nodes to process or feasible solutions to find. For each attempt you can add additional preprocessors to the solving sequence, using a simple configuration for the first attempts and only if the solution proves too complicated and the attempt reaches its limit, use more preprocessing tools in the next attempt. You can also decide to keep a solution at the end of an attempt, if the deviation of the solution found from the estimated best solution is within acceptable bounds. See section B.2 and the OSL solver manual for more information about the specific settings for mode 4.

4.8 Quadratic Problems

Apart from mixed integer linear programs, the agents' production, investment and consumption decisions may also be formulated as quadratic programming (QP) problems.

The generic formulation of a QP problem is as follows:

$$\max! \quad \frac{1}{2}\mathbf{x}^T Q\mathbf{x} + \mathbf{c}^T\mathbf{x} \tag{4.2}$$

$$A\vec{x} \leq b \tag{4.3}$$

$$\tag{4.4}$$

The linear part of the objective function ($\mathbf{c}^T\mathbf{x}$) and the constraints are implemented similar to the linear programming problem. The quadratic part ($\frac{1}{2}\mathbf{x}^T Q\mathbf{x}$) includes the matrix Q, which is symmetric with as many columns (and rows) as the length of the vector of solution variables $\mathbf{x}$.

4.8.1 Example starting from a scalar valued function

To understand the meaning of its coefficients, it may be helpful to use an example with two solution variables and rewrite the formula as a scalar function:

$$Q = \begin{pmatrix} q_{11} & q_{12} \\ q_{21} & q_{22} \end{pmatrix} \tag{4.5}$$

$$\frac{1}{2}\mathbf{x}^T Q\mathbf{x} + \mathbf{c}^T\mathbf{x} = \frac{1}{2}\left(q_{11}x_1^2 + q_{22}x_2^2 + q_{12}x_1x_2 + q_{21}x_1x_2\right) + c_1x_1 + c_2x_2 \tag{4.6}$$

$$= \frac{1}{2}q_{11}x_1^2 + \frac{1}{2}q_{22}x_2^2 + \frac{1}{2}(q_{12} + q_{21})x_1x_2 + c_1x_1 + c_2x_2 \tag{4.7}$$

So, as can be seen, the diagonal elements of the matrix, q_{11} and q_{22}, are twice the coefficient of the square of x_1, respectively x_2. The off-diagonal elements of Q represent the coefficients of the product of two variables and those associated to the same two variables should be equal (hence, the symmetry of the matrix). In case all the elements of Q related to a solution variable are zero, this variable has no quadratic component and enters the solution function only linearly. For simplification, in MPMAS only those variables with at least one non-zero element in Q need to be marked as quadratic and considered in the formulation of Q.

Implementation of a quadratic programming problem in `mpmasql` includes two steps. First, all quadratic activities have to be marked as such, using the `quadratic` attribute. Second, the subelement `#quadratic>` is used to specify the elements of the Q matrix of the row (first index) associated with the activity. The field names of the subelement refer to the columns of the Q matrix using activity identifiers. (The

actual order of columns and rows is automatically determined by `mpmasql` according to the order of activities in the decision matrix.)

Remember that the element of Q referring to the multiplication of the activity with itself (e.g. for the activity `maize_sell` the value for `#quadratic>maize_sell`) should be double the value observed in the scalar notation of the objective function. Also recall that all elements associated to the same two variables should be equal. For example, in the activity `maize_sell` the value assigned to `#quadratic>wheat_sell` needs to be the same as the value assigned to `#quadratic>maize_sell` in the activity `wheat_sell`. `mpmasql` will cross-check this.

Example implementation, model structure file,

```
#class CROPS {

   #activity sell {
   ...
      quadratic    #= 1
   ...
      #quadratic> #foreach(CROPS)_sell #= #table(quadratic_terms, 3,
         [#this(instance), #this(field, 1)])
   }
}
```

and data file:

```
 quadratic_terms   = (sql,2) {
      "SELECT 'wheat', 'wheat', 2
       UNION SELECT 'barley', 'barley', 4
       UNION SELECT 'maize', 'maize', 6
       UNION SELECT 'wheat', 'barley', 1
       UNION SELECT 'wheat', 'maize', 2
       UNION SELECT 'barley', 'wheat', 1
       UNION SELECT 'barley', 'maize', 0.5
       UNION SELECT 'maize', 'barley', 0.5
       UNION SELECT 'maize', 'wheat', 2
   "
}
```

4.8.2 Example for quadratic risk programming

Quadratic programming is often used to model risk behavior using a functional form like

$$E - 0.5rV \tag{4.8}$$

as objective function (see e.g. Hardaker et al. 2004, p. 192), where E is the expected value of the gross margin, r is the risk aversion coefficient, and V is the variance of the total gross margin. E is equivalent to the linear part of the objective function $c'x$, while V is calculated as $x'\Sigma x$, where Σ is the variance-covariance matrix of the gross margins of the individual production activities.

To implement this in MPMAS the Q matrix needs to be set to $-1 * r * \Sigma$, as can easily be seen by comparing equations 4.8 and 4.5.

4.9 Exogenous changes to matrix coefficients and agent-independent right hand sides

Sometimes you may want a matrix coefficient or and agent-independent capacity to be changed at a certain point of the simulation, e.g. to reflect a shift in policy. This can be achieved by adding fields with the `#exchange>` marker to the activity or constraint in question.

The general syntax for coefficients is

```
#activity ... {
   #exchange> <id_of_constraint>_<year_of_change>     #= <new_value>
}
```

, whereas right hand side values are exogenously changed as follows

```
#constraint ... {
   type  #= MASzero
   #exchange> value_<year_of_change>      #= <new_value>
}
```

Note that the timing of the change is given as explicit year. `mpmasql` will automatically adapt the matrix file depending on the value of your STARTYEAR configuration entry. This also holds for standalone matrix generation.

For example, in the following case:

```
#activity soil_transfer {
   #exchange>EU_livestock _limit_1994     #= -10
   #exchange>EU_livestock _limit_1999     #= -2
   #exchange>EU_livestock _limit_2001     #= -1.9
   #exchange>EU_livestock _limit_2002     #= -1.8
}
```

If your STARTYEAR is 1998, the value of the coefficient of soil_transfer in the constraint EU_livestock_limit will be -10, in any stand alone matrix and in the first period of the multi-agent model. It will change to -2 in the second period of the multi-agent simulation, then to -1.9 in the fourth, and to -1.8 in the fifth.

4.10 Fragmented markets: differentiate prices or LP coefficients between different groups of agents

The 'fragmented markets' feature let's you distinguish prices, i.e. objective function coefficients, and MILP coefficients between groups of agents. A group of agents with common prices (or coefficients) forms one market. By default, you have only one market with ID zero in your model.

If you want more markets, you need to include the following entry into your configuration file:

```
MARKETS =  <number_of_markets>
```

Markets like populations, soils, networks or clusters need to be numbered from zero to the number of markets minus one. To assign an agent to a specific market you need to include the special asset `which_market` into the asset table, and give it the corresponding number of the market. By default, all agents are part of market 0.

Within the model structure file you can then use the local variable `#this(market)` to differentiate values between different markets. This local variable works in the value definitions of `orow` and `orow_<year>`, MILP coefficients of activities, as well as `#exchange>` and `#harvest>` subelements.

It does not work in `...by` lists or field definitions, as the structure of the MILP needs to be the same as usually. It also will not work in asset definitions.

4.11 Producer organizations

Producer organizations can for example be marketing cooperatives of small holders that collect the produce of their members, process, store and sell it, and distribute their profit among members. The following examples are given in terms of such marketing organizations, however, also other types of organizations can be modeled using MPMAS producer organizations. Agents can be members in several producer organizations, and each producer organization may offer various services to its members. Note: Currently, producer organizations cannot be used in combination with fragemented markets.

If the producer organization feature is activated, agents solve a marketing decision problem after crop yields have been determined deciding how much of their harvest they want to market through their producer organization. After that the producer organizations solve their own decision problem determining what to do with the collected harvest and determining their profit. After that the profit is redistributed among its members. Only after that, farm agents then take their consumption (4.4.2) or 'harvest' decision (4.5).

4.11.1 The producer organization

Producer organizations are activiated by including a `PRODUCER_ORGANIZATIONS` table into the [MPMAS TABLES] section of the data file. `mpmasql`will automatically determine the number of producer organizations from the number of entries in thie table. The table has one key column and at least two further columns. They first (key) column should contain the running number of the producer organization starting from 0. The second column contains a name or description of the producer organization, and the third column should contain the relative path to a ".cll"-file containing the decision problem of the producer organization. Feel free to add further columns for your own use.

```
[MPMAS TABLES]
PRODUCER_ORGANIZATIONS  = (sql) {
      "SELECT  0, 'PO wheat', 'cfg/po.cll', 'wheat'
      UNION SELECT 1, 'PO maize', 'cfg/po.cll', 'maize'"
}
```

The .cll file for producer organizations looks similar to a standard .cll file except that it should additionally contain its own [INSTANCES] section. Further, as of now, producer organizations do not have assets, so `#asset`, `#perennial` or `#livestock` entries are meaningless. Only `#exchange>` subelements are defined. `MASzero` constraints can be used to set fixed right hand side values.

You can either define separate .cll files for each producer organization or have several producer organizations share a common .cll file and distinguish between then using the local variable `#this(PO)`, as in the following example. This local variable works in field and value definitions, `...by` lists, the [INSTANCES] section and `#exchange>` subelements.

In the following example, a common .cll file is used to define the decision problem of a maize and a wheat marketing organization, each of which runs a grain storage and has to decide when to sell its produce.

```
[ACTIVITY TYPES]
sales, default, storage, transfer, finance

[CONSTRAINT TYPES]
balance, default, finance
```

```
[INSTANCES]
POCROPS   = [#table(PRODUCER_ORGANIZATIONS , 4, [#this(PO)] )]
FINANCIAL = [account]

[MODEL]

#class POCROPS {

   #activity collect_from_members {
      name  #= "Collect " #this(instance) " harvest from members"
      unit  #= t
      type  #= transfer
      #this(instance)_production_of_members  #= 1
      #this(instance)_at_PO         #= -1
      account_fbalance  #= 0.001

   }
   #activity sell_immediately {
      name  #= "Sell " #this(instance) " immediately"
      unit  #=  t
      type  #= sales
      #this(instance)_at_PO   #= 1
      account_fbalance  #= - #table(prices , 3 , [#this(instance),
         #var(STARTYEAR) - 1 ] )
   }
   #activityby sell_after_month #steps(1,11,1) {
      name  #= "Sell " #this(instance) " after " #this(by , 1) "
         months"
      unit  #=  t
      type  #= sales
      #this(instance)_stored_for_#this(by,1) #= 1
      account_fbalance  #= - #table(prices , 3 , [#this(instance),
         #var(STARTYEAR) - 1 ] ) * (1.4 - 0.01 *( #this(by,1) -7 )
         ^2 )

   }
   #constraint production_of_members {
      name  #= #this(instance) " production of members"
      unit  #=  t
      type  #= balance

   }
   #constraint at_PO {
      name  #= #this(instance) " transported to PO"
      unit  #=  t
      type  #= balance
   }
   #constraintby stored_for #steps(1,11,1) {
      name  #= #this(instance) " stored for " #this(by , 1) " months"
      unit  #= t
      type  #= balance
   }
   #activityby store_until_month #steps(1,11,1) {
      name  #= "Store " #this(instance) " until end of month "
         #this(by,1)
```

```
        unit   #= t
        type  #=    storage
        #this(instance)_at_PO   #= 1
        #this(instance)_stored_for_#this(by,1) #= -1
        account_fbalance  #= 0.1 + 0.01 * #this(by,1)
    }
}

#class FINANCIAL {
    #constraint fbalance {
        name #= "Financial balance"
        unit #= #var(CURRENCY)
        type #= finance
    }
    #activity profit {
        name #= "Financial balance"
        unit #= #var(CURRENCY)
        type #= finance
        orow #= 1
        #this(instance)_fbalance #= 1

    }
    #activity fix_costs_accounting {
        name  #=  "Fix costs"
        unit #= #var(CURRENCY)
        type #= finance
        #this(instance)_fbalance #= 1
        #this(instance)_fix_costs #= -1
    }
    #constraint fix_costs {
        name  #= "Fix costs"
        unit  #= #var(CURRENCY)
        type  #= MASzero
        value    #= #if(#this(PO) == 0, 5, 10)
    }
}
```

4.11.2 The farm agents' marketing decision

Similar to the consumption or harvest decision stage, certain decisions have already been taken and can not be reversed in the the marketing stage (e.g. the cropping decision). Activities that cannot be changed in the marketing stage are marked by setting the `marketfix` attribute of the activity to one.

```
#class CROPS {
    #activityby production_on_soil #foreach(SOILS) {
      ...
        marketfix   #= #ifthenelse("'" #this(instance) "' eq 'fallow'",
            0,1)
      ...
    }
}
```

Further, the activities that represent the decision of agents to market through the producer organization have to be included. In our example, it just consists of the corresponding sales activity and the symbolic object representing membership in the PO, which is required for the agent to have the

option to use the PO marketing channel. (Note: The table POs is just an auxiliary user table linking crop to PO number. The #harvest> subelement also applies to the marketing stage.)

```
#class POCROPS {
   #activity sell_through_PO {
      name  #= "sell "~#this(instance)
      type  #= MASsell
      unit     #= t
      orow  #= #table(prices, 3 , [#this(instance), #var(STARTYEAR) -
         1] ) * 1.05
      orow_#foreach(YEARS) #= #table(prices, 3 , [#this(instance),
         #var(STARTYEAR) - 1 ] ) * 1.05
      #this(instance)_balance    #= 1.1
      #harvest>#this(instance)_balance#=  1
      #this(instance)_PO_member  #= 1
   }
   #asset PO_membership {
      name  #=    #this(instance) " PO membership"
      div   #= 0
      symbolic #= 1
          lifetime   #= 1
          acqcost #= 0
          eqshare #= 1
          irate   #= #var(INTEREST_SHDEP)
          con  #= #this(instance)_PO_member
          act  #= #this(instance)_PO_become_member
           multiplier   #= 1000
      innogrp  #=  never
      #producer_organization>membership   #= #table(POs, 2,
         [#this(instance)])
   }
   #constraint PO_member {
      name  #= "Member in PO " #this(instance)
      type  #= ability
      unit  #= t
   }
   #activity PO_become_member {
      name  #= "Become member in " #this(instance) " PO"
      integ #= 1
      ubound   #= 0
   }
}
```

4.11.3 Linking farm agents and producer organizations

Finally, farm agent activities and producer organization services still need to be linked. First, each producer organization needs to know its members. This is achieved through the #producer_organization>membership element that can be added to symbolic investment objects and takes the ID of the PO as an argument. Each agent that owns such an asset is considered a member of the PO. There may be several types of assets that indicate a membership in a PO, but each asset can only indicate membership in one single PO.

Second, members activities need to be accounted for in the PO decision problem, and the result of the PO activity needs to be reinserted into the final farm agent outcome. These linkages are defined in the table PRODUCER_ORGANIZATION_SERVICES in the [MPMAS TABLES] section. This table needs

six columns, two of which are key columns. The first column contains the PO number, the second the PO service (remember a PO may offer several services). The third column contains the identifier of the activity in the farm agent decision problem that indicates the quantity of the service the agent wants to apply for, i.e. in our case the quantity of produce it offers to the PO. The fourth column indicates the constraint in the PO decision problem where the sum of all member service applications is entered on the right hand side, i.e. in our case the total amount of produce offered to the PO by its members. The fifth column contains the identifier of the activity in the PO decision problem which indicates how much of the produce was actually taken up by the PO. The sixth column contains the identifier of the activity in the PO decision problem which indicates the income obtained by the PO related to this service that is to be redistributed among the agents.

```
 PRODUCER_ORGANIZATION_SERVICES = (sql) {
      "  SELECT 0, 'wheat_sale',  'wheat_sell_through_PO',
          'wheat_production_of_members',
          'wheat_collect_from_members', 'account_profit'
         UNION SELECT 1, 'maize_sale',  'maize_sell_through_PO',
            'maize_production_of_members','maize_collect_from_members',
            'account_profit'
      "
}
```

4.12 Diffusion of innovations

MPMAS includes a module for the communication process involved in the diffusion of innovations. This model assumes that the adoption of an innovation consists of two phases: First, an agent needs to learn about an innovation and be convinced that it actually works, then second he will check, calculate and assess whether it is profitable for him to actually use it himself. The second phase is inherently represented by solving the optimization problem in the investment/production decision, if the activities related to the innovation are part of the solution, the farmer adopted the innovation.

The diffusion of innovations module represents the first phase, and thus deals with the question, whether the activities related to the innovation are at all included in the optimization problem. For short, agents are grouped into communication networks and innovativeness segments. Networks determine, which other agents actually have an influence on an agent's knowledge about an innovation. Different networks are completely isolated, i.e. even if an innovation diffuses through network one, most agents in network two might never hear of the innovation. Networks might e.g. be defined on geographical terms, or distinguishing socioeconomic strata e.g. commercial and family farms. The innovativeness segments reflect agent traits like curiosity, education, risk aversion and connectedness relative to other agents. Highly innovative agents are well-connected, hear first about and innovation and are always prepared to try something new, less innovative agents are more skeptical and less well connected, they wait and observe the experience of other agents before they themselves try an innovation. In the model, when an innovation is introduced it becomes available first only to the highly innovative segment of agents, and only if a certain percentage of this segment (defined by the OVERLAP parameter) has adopted the innovation, it becomes available to the next higher segment. The theory behind the module is described elsewhere in more detail, here we focus on its implementation in `mpmasql`.

4.12.1 Defining the number of segments and its thresholds

First, you need to define into how many networks and innovativeness segments you want subdivide your agent population. You do this by specifying the number of networks and a list of lower thresholds for each segment in the model configuration. E.g. one network and the five standard segments would be implemented as follows.

```
NETWORKS    =  1
THRESHOLDS  =  [0, 0.2, 0.16, 0.5, 0.84]
```

4.12.2 Defining innovations

Then you need to define your innovations, respectively innovation groups. An innovation in MPMAS is always tied to an investment asset. The adoption of the innovation is actually modeled as an investment into this asset.

To make an asset an innovation, you have to assign it to an innovation group. This is done by setting the innogrp attribute of the asset.

If your innovation is actually represented by an asset anyway (e.g. in the case of a new type of machinery) you can directly work with this asset. E.g. the following would make all machinery part of the innovation 'mechanization' (Of course, most of the times you would probably use a `\#table`-function to assign different types of machinery into different innovation groups.)

```
#class MACHINERY {
   #asset machinery_asset {
      ...
      innogrp  #= mechanization
      ...
   }
}
```

In other cases, (e.g. when the innovation is the access to a new market), you have to introduce a symbolic access asset without any cost and infinite lifetime, and also the corresponding investment activity and constraints. All activities related to the innovation are then restricted by a positive requirement in the access constraint.

E.g. for an access to export markets, which allows selling products at a higher price, one might use the following implementation.

```
#class PRODUCTS {
   #activity sell_for_export {
      ...
         export_access     #= 1
      ...
   }
}
#class ACCESS {
   #asset access_asset  {
      name  #= ...
      symbolic #= 1
      act   #=   #this(instance)_get_access
      con   #= #this(instance)_access
      multiplier  #= -1000000
      lifetime #= 100
      interest #= 0
      eqshare  #= 1
      innogrp  #= #this(instance)
      div      #= 0
   }
   #constraint access {
      name  #= ...
      unit     #= ...
```

```
    }
    #activity get_access {
        name  #= ...
        unit     #= ...
        integ #= 1
        ubound   #= 1
    }
}
```

The multiplier should of course be chosen adequately high and the instance 'export' should be included in the class ACCESS.

4.12.3 Defining availability and accessibility of innovations

To define which object is available in which segment at the beginning you need to include the table INNOVATIONS into the MPMAS TABLES section. E.g.

```
[MPMAS TABLES]
INNOVATIONS = (sql) { "SELECT network, segment, innovation_group,
   availability, accessibility FROM tbl_innovations"}
```

A corresponding table for our example could look as follows: In the example, mechanization has

network	segment	innovation_group	availability	accessibility
0	0	mechanization	0	1
0	1	mechanization	0	1
0	2	mechanization	0	1
0	3	mechanization	0	0
0	4	mechanization	0	0
0	0	export	4	0
0	1	export	4	0
0	2	export	4	0
0	3	export	4	0
0	4	export	4	0

already diffused into the early majority segment, while export access is an innovation which will only become available in the fourth period of the simulation.

4.12.4 Assigning agents to networks and segments

Agents can be assigned to an innovation segment in two different ways. On the one hand, the special asset 'innovativeness' can be included into the ASSET_ENDOWMENTS, respectively ASSET_CDF tables and assigned the value of the segment.

On the other hand, the innovativeness of an agent may be determined by the assets she owns. To do so, you can assign an innovativeness attribute to each asset. The agent will be assigned to the lowest innovativeness value indicated by any of her assets, the value given in the ASSET_ENDOWMENTS, resp. ASSET_CDF table will then only apply if it is lower than this.

E.g. in our example agents, which own machinery should be assigned to segment 2 or lower, otherwise the initial distribution of assets would be inconsistent with the diffusion of the innovation. It would therefore be necessary to include the innovativeness attribute for all machinery assets and set it to two:

```
#class MACHINERY {
   #asset machinery_asset {
       ...
       innovativeness #= 2
       ...
   }
}
```

4.12.5 Fine-tuning the diffusion process

The diffusion process can be fine-tuned using the OVERLAP (see B.3) and the CUMADOPT and COMTYPE parameters (see B.2).

4.13 Using agent attributes in the decision problem

Note: This feature currently only works with in the Germany-version of MPMAS. Ask a maintainer to activate it for your application if you want to use it.

Sometimes you will want to make agent decisions dependent on agent-specific characteristics that are not covered by any of the other submodels. The `#agent_attribute_to>` subelement allows you include further agent-specific characteristics into the matrix.

Used within constraint definitions,

```
#agent_attribute_to> value #= <agent attribute name>
```

would enter the agent-specific value for the desired characteristic on the right-hand side of that constraint.

Within activity definitions,

```
#agent_attribute_to> orow #= <agent attribute name>
```

would enter the agent-specific value for the desired characteristic as the objective function coefficient of that activity, while

```
#agent_attribute_to> <constraint identifier> #= <agent attribute
    name>
```

would enter it as a coefficient of that activity in the indicated constraint.

Currently the following agent attributes have been activated for the use with this feature:

Attribute identifier	Description
age_household_head	The age of the household head (only with Advanced Demography Model)
age_successor	The age of the next successor to the household head (only with Advanced Demography Model)
last_income	Last year's income

4.14 Using plot attributes in the decision problem

Note: This feature currently only works with the special country switch for Germany (5)

Similar to agent attributes also plot-related attributes can be entered into the matrix. These plot attributes are specified in user-defined maps, which are named CatchMap00Udef00.txt, CatchMap00Udef01.txt, ... Each of these maps may contain a user-defined attribute that is specific to a grid cell of the map. To use these maps you need to set NUDEF in the model configuration to the number of maps you want to use. (Note: Maps need to be consecutively numbered from 0 to 1 - NUDEF).

For incorporating these values into the decision problem a `#landscape_attribute_to>` subelement is provided by `mpmasql`, with the generic syntax:

```
#landscape_attribute_to> <what > #= <aggregated_plot_attribute>
```

`<what>` can be 'value' for the RHS value of a constraint, 'orow' for the objective function coefficient of an activity or any 'constraint_idientifier' for the coefficient of an activity in the constraint.

Plot attributes need to be aggregated by soil type, respectively nutrient response unit to be entered into the matrix. The `<aggregated_plot_attribute>` expression consists of three elements:

```
<aggregation_function>_S<soil_type>_P<mapid>
```

Aggregation functions currently defined are:

Code	Function
mean	mean, average
sum	sum
med	median (with average of two values for even samples)
min	minimum
max	maximum
medl	(with smaller of two values for even samples)
medu	(with greater of two values for even samples)

For example, the following would average the value in the CatchMap00Udef00.txt file over the cells with soil type/NRU 0 and enter it on the right hand side of constraint elevation.

```
#constraint elevation {
  ...
  #landscape_attribute_to> value #= mean_S0_P0
  ...
}
```

4.15 Household member-specific attributes

Note: This feature currently only works with the advanced demography model and the special country switch for Germany (5)

For example, for modeling farm succession, it is useful to be able to mark certain household members with their own attributes, e.g. to make sure that one person can receive subsidies for young farmers only once.

By adding the `#hhmember_mark>` subelement to an activity, you can request that the solution value of of this activity in the investment problem be added to or subtracted from a household member-specific marking value. The field definition of the `#hhmember_mark>` subelement indicates for which household members the value should be recorded – currently, 'household_head' and 'successor' are the only options – and the value definition defines whether the value should be added ('add') or subtracted ('subtract').

For example, the following would add the solution value of the employ_successor activity to the mark attribute of the next potential successor of the household head.

```
#activity employ_successor {
  ...
  #hhmember_mark> successor #= add
  ...
}
```

The recorded marks can be used in any decision problem using the feature described in Section 4.13. The corresponding attribute names are 'mark_household_head' and 'mark_successor'.

4.16 Selling of assets

Note: This feature currently only works with the special country switch for Germany (5), not for perennial or livestock assets or any other assets with land demand.

To allow agents the selling of farm assets, you can include activities of type 'MASdisinvest'. Since an agent may own several units of the same type of asset with different ages and thus different potential resale values it is useful to include several disinvestment activities for each asset you want an agent to be able to sell.

To define a disinvestment activity, you only need to set the activitiy type to 'MASdisinvest' and add an `#disinvest>asset` field that indicates the type of asset which is sold by this activity. Remeber to use an integer activity for indivisible assets.

For example, the following would define five disinvestment activities for the tractor asset:

```
#activityby tractor_disinvest #steps(1,5,1) {
   type #= MASdisinvest
   #disinvest>asset #= tractor
   integ    #=   1
   ubound   #=   0
   lbound   #=   0
}
```

The values for the upper and lower bounds should be set to zero (mpmasql will do so automatically anyway), as the upper bound will be set by MPMAS according to the number of assets owned by the agent represented by this activity.

While you are free to define coefficients for the activity, MPMAS will fill certain coefficients of the activity according to the individual assets owned by the agent. It will set the multiplier value into coefficient of the capacity constraint to account for the reduction of capacity and it wil set coefficients in the MASong, MASly1 and the CASHFLOW_COEF constraint, if applicable, to record the additional liquidity and the reduction in debt service achieved by the disinvestment. (Note that on selling the asset the agent will have to repay any loan taken for it, even if the resale value is lower than the remaining principal to be paid.) It will also set the objective function coefficient of the activity. Assuming a correct objective function value is a bit tricky. In terms of accounting, the sale of an asset only produces income if the return of the resale is greater then the book value of the asset accounted for in the balance sheet, if it is lower it even results in a loss. However, for plaaning we might want to assume that a farmer considers the book value of an asset he does not use anymore to be zero (it will only lose value over time, but not generate any income). Further disinvestment in MPMAS is associated to the direct repayment of any loan taken on the asset, so that disinvesting will also save interest which otherwise would have to be paid.

The objective function coefficient is therefore calculated as

```
( resale_share * time value of asset - resale_book_share * time
   value of asset + saved_interest ) * INTEREST_SHDEP
```

(Multiplication with INTEREST_SHDEP transforms the change into an infinte annuity for the annualized form of investment calculation used in the MPMAS investment decision.) The two share coefficients used in the formula have to be defined in the corresponding asset. For example:

```
#asset tractor {
...
  resale_share = 0.9
  resale_book_share = 1.0
  disinvest_completely = 0
...
}
```

The `resale_share` determines, which share of the time value of the asset the agent may recover by selling it. Besides the objective function coefficient it directly affects the increase in cash caused by the selling. The `resale_book_share` only affects the objective row and determines whether the agent fully considers the change in the balance sheet for his decision. In some cases the asset might denote e.g. a stable place and the amount of the asset owned by an agent does not correspond to a number of single asset objects, but rather to the size of one asset (i.e. a stable). In this case, youo would typically want the agent to either sell the full asset (the whole stable) or nothing rather than only a few stable places. This can be achieved by setting `disinvest_completely` attribute to one.

Chapter 5

Testing decision models with mpmasql and mqlmatrix

While developing and calibrating your model, you will often encounter situations where things do not work as expected. In case you suspect the error to lie in the decision matrix, you will need to have a look at the prepared matrix, experiment again with some different coefficients and solve it outside of MP-MAS. Although you can open the matrix (or its commented version) in a spreadsheet program, MPMAS matrices can easily grow large and hard to handle as a whole in spreadsheets.

`mqlmatrix` provides an interface to browse, experiment and solve your matrices using the standalone solver of MPMAS (MilpCheck). It can read a number of different matrix formats, that can are either produced as debugging output of MP-MAS or directly created using `mpmasql` (see 5.2). All of these files only contain numbers and no descriptions, and so `mqlmatrix` has to rely on the ActivityInfo and ConstraintInfo files that `mpmasql` creates when creating `mpmas` input files.

```
mqlmatrix <name_of_matrix_file> <typeflag> <otherflags>.
```

where `<typeflag>` specifies what type of matrix the browser should expect. (See 5.2 for more information on matrix types). It is required and should be one of:

-V ⟨number⟩	`mpmas` input matrix (.dat)
-S	standalone matrix (.mtx)
-F	failed matrix (.mip/.mpx)
-D	requested matrix (.mip/.mpx)

The optional `<otherflags>` are used, inter alia to help `mqlmatrix` find the model info files required (see 5.3). For an overview of all command line options run

```
mqlmatrix --help
```

One of the most useful functions of `mqlmatrix` is showing selected parts of the matrix in a spreadsheet. At least under Ubuntu 10.04, we strongly recommend installing `gnumeric` for this task, as OpenOffice takes rather long to open. `mqlmatrix` will automatically choose gnumeric if it is available.

5.1 Interactive commands

Once you loaded the matrix into memory, `mqlmatrix` provides an interactive command-line environment.

The following commands are available:

GENERAL:

exit	quits the interactive session
help	prints this help
quit	quits the interactive session

I/O

printmtx <suffix>	saves the matrix as .mtx into a file with the indicated <suffix>
printtxt <suffix>	saves the commented matrix into a file with the indicated <suffix>
reload	reloads the matrix from the original file
solve {<suffix>}	tries to solve the matrix and to import the solution vector if successful
impsol <filename>	import solution vector from file

COLUMN/ROW INFORMATION

info	prints general information about the matrix
act <id>	prints information about activity with mpmasql identifier <id>
con <id>	prints information about constraint with mpmasql identifier <id>
col <C>	prints information about matrix column no. <C>
row <R>	prints information about matrix row no. <R>
coef col <C> {-l}	prints all coefficients <> 0 of matrix column no. <C>
coef row <R> {-l}	prints all coefficients <> 0 of matrix row no. <R>
rhs <R>	prints the rhs for row <R>
lhs <R>	calculates the LHS for row <R> based on the current solution
lhslb <R>	calculates the lhs for row <R> based on the lbounds of columns
Q <C>	show Q matrix entries for column C
multiply Q <value>	multiply Q matrix by <value>
divide Q <value>	divide Q matrix by <value>

LISTING

list act {-l -o -t}	list all activities
list con {-l -o -t}	list all constraints
list == {-l -o -t}	list all equality constraints
list lhs {-l -f -e -d -o -t}	prints all LHS values <> 0 (or all with -f)
list rhs {-l -f -e -d -o -t}	prints all RHS values <> 0 (or all with -f)
list sol {-l -e -d -o -t}	lists the solution for all activities where it is <> 0
list int {-l -o -t}	list all integer activities
list lb {-l -o -t}	list all activities with lower bounds > 0
list lhslb {-l -f -e -d -t -o}	list the LHS based on lbounds where LHS <> 0

Options for listing commands:

-l	redirects the output to pager in a separate window

-o <filename>	redirects the output to filename
-e	scientific number format
-d <n>	number of decimals (default n = 3)
-f	prints all elements, not only those <> 0
-t <type>	list only constraints/activities of mpmasql type <type>

TABLE VIEW	
tab <C1>..<C2> : <R1>..<R2>	view tableau between columns C1 and C2 and rows R1 and R2 in spreadsheet
tab <typelistA> : <typelistC>	view tableau for activities/constraint of types given in spreadsheet, where <typelist> is a comma-separated list of mpmasql types
MODIFYING	
modify row <R> (<attr> = <new value>)	sets attribute <attr> of row <R> to <new value>
modify col <C> (<attr> = <new value>)	sets attribute <attr> of column <C> to <new value>
modify rows <R1:R2> (<attr> = <new value>)	sets attribute <attr> of rows <R1> to <R2> to <new value>
modify cols <C1:C2> (<attr> = <new value>)	sets attribute <attr> of column <C1> to <C2> to <new value>
modify coef <C> <R> <new value>	sets coefficient in column <C> and row <R> to <new value>
OTHER OPTIONS	
set_risk_aversion <value>	set risk aversion coefficient for evaluation of quadratic part of objective values. (Note: has no effect on current Q matrix)

`{}` *denotes optional arguments*

5.2 Formats of MP problems

`mqlmatrix` can read `mpmas` input or standalone matrices created by `mpmasql`, matrices saved during `mpmas` simulations automatically, because the OSL failed solving it, or matrices you requested to be saved by `mpmas` during simulation with the DBGMILPS entry in the `mpmasql` control file.

5.2.1 MPMAS input format (.dat)

You can directly read in the `mpmas` matrix input file (usually named <scenarioname>_MILP.dat) for inspection. Solving such a matrix usually does not make sense, as the RHS is empty and several coefficients are usually missing (e.g. yields or investment cost). The type flag to use is `-V`, and you have to provide the additional information, whether you use a second production or a consumption decision in your model. (Note: if you use TSPC this won't work yet).

-V 0	basic format
-V 1	with second production stage ("harvest milp")
-V 2	with advanced consumption model

If your matrix is very large, it will take a long time to write it and to load it into `mqlmatrix`. If you run mpmasql with the flag `--compact` it will write only non-zero coefficients in triplet format into the matrix file. This file cannot be used by MP-MAS, however you can look at it in `mqlmatrix` if you also use the `--compact` flag when loading.

5.2.2 MPMAS standalone MP problem format (.mtx)

As noted above, solving a standard `mpmas` input matrix does not make sense, as the RHS is empty and several coefficients are usually missing (e.g. yields or investment cost). It is often useful to really test your matrices before including them into a full MP-MAS setup. Fortunately, `mpmasql` has a special feature for this task.

Creating standalone MP problems with `mpmasql`

If you run `mpmasql` with the flag `-S` it omits creating all other input files and creates a matrix file in the correct layout for standalone matrix use (.mtx). (Like for .dat matrices, you can also use the `-compact` flag to save disk space and time.)

However, you have to help `mpmasql` by putting in the additional coefficients and providing right hand sides. To avoid having to manually change your files whenever you want to create a standalone matrix, use the `#ifalone` function (see C), then mpmasql will know which value to take when run with `-S` and which when not.

You can create several matrices each with a different RHS in one go, similar to scenarios. Sets of Right-hand-side values (Right-hand-sides or RHS, for short) are defined in much the same way as scenarios are. You can convert several matrix files with differing RHS at the same time. You can also mix this with the scenarios feature, i.e. use the same collection of RHS once for every scenario.

You need a RHSDEF table, with the columns ‘rhs’, ‘constraint’, ‘rhsvalue’. Again, the column ordering is important, the column naming is not and the combination of the first two should be unique. ‘constraint’ refers to a constraint by the internal constraint identifier, and value specifies the value to be put on the right hand side. Values not specified are set to zero.

Table 5.2: Sample RHSDEF table

rhs	constraint	value
farm1	farm_labendow	3
farm2	farm_labendow	1
farm1	1_machinery_owned	2
farm2	2_machinery_owned	1

In the example (Tab. 5.2), assuming the RHSs represent a farming household each, ‘farm1’ has three units of labour and two units of ‘MacT067_owned’ (which in the sample files stands for a 67kW Tractor) and ‘farm2’ has one unit of labour and three units of ‘MacT200_owned’(i.e. a 200kW Tractor in the sample files).

There are three different ways to specify RHSDEF in the control file:

As an SQL query to the database:

```
RHSDEF   = (sql) { "SQL SELECT statement which can also
      span over several lines" }
```

As a path to an text file in tab-separated format:

```
RHSDEF   = (ascii) { "name of an ascii file with tab separated
   values" }
```

or as a request to transform the asset table of the full multi-agent model.

```
RHSDEF   = (transform_assets)
```

If you use the third option, the RHS is directly created from the ASSET_ENDOWMENT, HOUSEHOLD_COMPOSITION and AGENT_INFO tables (in the data file), and the soil RHS is either directly read from `mpmas` input maps, if you specify the map directory in the control file:

```
RHS_MAPDIR  = directory
```

, or alternatively read from a table specified in the control file:

```
RHS_SOILTAB = (sql) {"SQL SELECT statement which can also
                    span over several lines"}
```

Similar to the SCENLIST entry, you then further need an RHSLIST entry in your control file, telling `mpmasql`, which RHS should actually be created.

You also have three options to specify RHSLIST :

```
RHSLIST  = (sql) { "SQL SELECT statement which can also
        span over several lines" }

RHSLIST  =(ascii) { "name of an ascii file with one entry on each new
   line" }

RHSLIST  = (list) {[ list of comma-separated values ]}
```

Loading standalone MP problems with `mqlmatrix`

To load a standalone matrix you have to run `mqlmatrix` the using the `-S` type flag, and additionally the `--compact` flag in case you created it with this flag in `mpmasql`.

5.2.3 MP problems automatically saved by `mpmas` for debugging

Whenever a decision problem of an agent cannot be solved by the OSL, because it is infeasible, unbounded or solving it surpasses the time or iteration limit, MP-MAS saves the associated matrix in the out/test/ folder. These matrices have the extension .mip or if you run `mpmas` with the `-M` flag in the much more compact .mpx format. To load such matrices into `mqlmatrix`, you have to use the type flag `-F`. `mqlmatrix` uses the extension to distinguish between .mip and .mpx format.

5.2.4 MP problems saved by `mpmas` for debugging on request

You can also request that the decision problems of a specific agent be always saved by `mpmas`, e.g. to check the coefficients dynamically generated at runtime. You only have to include the farmstead identifier into the DBGMILPS list of the mpmasql control file. E.g.

```
DBGMILPS = 1235, 124, 345
```

will save the decision matrices of agents with farmstead id 1235, 124 and 345. The files will be written into the out/test/ directory and be either in .mip or .mpx format, depending whether you used the `-M` flag or not. To load such matrices into `mqlmatrix`, you have to use the type flag `-D`. `mqlmatrix` uses the extension to automatically distinguish between .mip and .mpx format.

5.3 Indicating the location of the model info files

`mqlmatrix` needs the ActivityInfo and ConstraintInfo files produced by MPMASQL to interprete the matrix file. There are several options: you can let `mqlmatrix` guess, help it with guessing, or explicitly give the files.

If you do not specify file names explicitly mqlmatrix will guess these filenames from the name of the matrix file and look for them in `xlsInput/`, e.g.

```
mqlmatrix  -F ../Mpmas/out/test/Failed\_BSL\_\_ 1035\_
    Preloaded122.mip
```

will make `mqlmatrix` look for BSL_ActivityInfo.txt and BSL_ConstraintInfo.txt in `xlsInput/`

Or, you can specify a different folder using the flag `-I <infofiles>`, e.g.

```
mqlmatrix  -F ../Mpmas/out/test/Failed\_BSL\_\_ 1035\_
    Preloaded122.mip  -I input/dat/
```

will make `mqlmatrix` look for BSL_ActivityInfo.txt and BSL_ConstraintInfo.txt in `input/dat/`. Or, you can specify a different folder by placing a file named `mqlmatrix.conf` into the working directory (the directory from which you call `mqlmatrix`) and put in it

```
infofiles   = <folder name>
```

Or, you can explicitly use the flags -A and -C to tell the script, which files to use, e.g.

```
-A ../input/dat/GEN\_ActivityInfo.txt  -C
    ../input/dat/GEN\_ConstraintInfo.txt.
```

5.4 mqlmatrix.conf

By default, mqlmatrix will write files to the current working directory and look for MilpCheck in `/usr/local/bin/MilpChe`
This can be overriden by using the `-O <outdir>` and `-M <milpcheck>` flags.

As it would be pretty cumbersome to always specify all these locations by hand, especially as you usually work in an MPMAS version folder, which has some conventions, you can set your own defaults for all of these locations by placing a file named `mqlmatrix.conf` into the working directory (the directory from which you call `mqlmatrix`). If such a file exists, `mqlmatrix` will read the defaults from these files, though they can still be overridden using the flags.

The mqlmatrix.conf is a simple text file and can have the following entries (all of them are optional):

```
outdir   = <output directory>
milpcheck   = <milpcheck executable>
infofiles   = <location of the ActivityInfo and ConstraintInfo file>
```

Chapter 6

Preparing agent populations with mpmasdist

The tool `mpmasdist` allows its user to create, endow and spatially distribute model agents. It employs similar concepts and language elements as `mpmasql`. Data may be provided in the form of SQL queries, tables in text files and maps in ESRI ASCII format.

Using `mpmasdist`, agent populations are sequentially built up over a number of steps. For example, in a first step, a list of agents is created and in a second step, the number of plots each agent owns could be determined. In a third step, agents' farmsteads could be spatially distributed and in a fourth step, their plots. Subsequently, buildings and machinery could be determined. Content and sequence of steps can be freely determined by the model user and are defined in the control file in text format (by convention with the extension .dst).

Each agent is associated to a list of assets and a number of attributes that are generated, changed, used or deleted during the distribution process. After each step, all the information associated to the agent population can be saved. Each step can start with the agents in memory or by restoring a stored agent population from file. In this way, the generation process can be stopped and re-initiated allowing alterations to later steps without having to run through previous steps again, and also giving the user the opportunity to split up the generation algorithm over several control files.

The distribution is started by calling `mpmasdist` with the name of the control file, e.g.

```
mpmasdist example.dst
```

6.1 File structure

The control file is subdivided into six parts that bear the headings DB, VARIABLES, TABLES, MAPS, CONTROL and DISTRIBUTION. The actual steps of the distribution process are defined in the DISTRIBUTION section. The CONTROL section determines, which of these steps will be run. The other sections define or import the necessary data.

6.1.1 DB

The DB section contains the information necessary to connect to the database, similar to the `mpmasql`syntax. It is of course only needed if database queries are to be used. If the password is not provided in the file, `mpmasdist` will ask for it at runtime.

```
[DB]
DB     =  mpmasql_example_db:localhost
DBTYPE       =  mysql
DBUSER       =  testuser
DBPWD    =  testuse
```

6.1.2 VARIABLES

In the VARIABLES subsection, users may define variables and lists for use in the other section using the MpmasQL Function Language similar to the use in `mpmasql`.

```
[VARIABLES]
SEED     =  123
SOILS        =  #steps(0, 4, 1)
```

The values of these variables may be overridden when running `mpmasdist` by using the command line option `--set` (see also section A.3).

6.1.3 TABLES

The tables section has the same same syntax as the USER TABLES section in `mpmasql` (cf. section 3.5.4). For example, the code below defines a table 'land_cdf" with two key columns and fills it with data retrieved from the database using the shown query.

```
[TABLES]
land_cdf = (sql, 2) {
      "    SELECT soil_type, upper_bound, quantity FROM
          soil_distribution"
}
```

6.1.4 MAPS

The MAPS section is used to load maps in ESRI ASCII format. The code below, for example, loads the map contained in the file `study_area_land_use.txt` and makes it available for further use under the name 'landuse'.

```
[MAPS]
 landuse     =  "study_area_land_use.txt"
```

6.1.5 CONTROL

The CONTROL subsection can be used to control which of the distribution steps defined under DISTRIBUTION will actually be performed. The keyword INCLUDE can be used to number the steps which should be performed. Alternatively, the keyword EXCLUDE to let `mpmasdist` skip some steps. If both keywords are used at the same time, only those steps which are listed under INCLUDE, but not under EXCLUDE are performed (e.g. 11, 11, 13, 14) below.

```
[CONTROL]
INCLUDE = [10, 11, 12, 13, 14]
EXCLUDE = [ 4, 5, 12]
MKPATHS = ["input/gis/", "population/"]
```

The MKPATHS can be used to ensure, that certain folders will be created if they do not exists. This is especially helpful if you create various populations with one control file, and want them to be saved in different folders.

6.1.6 DISTRIBUTION

The steps that make up the distribution algorithm are listed in the DISTRIBUTION section. The description of each step starts with STEP <number> \{ and ends with a closed bracket. Steps will be processed by mpmasdist starting from the step with the lowest to the step with the highest number. The order they appear in the file is irrelevant, although for consistency and understandability it is, of course, recommendable to list them in order. Numbers need not be consecutive, so you can leave gaps in case you might want to introduce a step in between at a later point of time.

Common fields and sections for all distribution types

Steps may perform very different tasks and require very different types of information to do so. All types of distribution steps do, however, accept the fields DESCRIPTION, TYPE, VERBOSE, SEED, and the sections AGENTS, OUTPUT, and LOOPS. The TYPE field is required and first defines what type of distribution should be performed, the DESCRIPTION field is just a comment that will be printed on screen during runtime in order to facilitate supervision of program runs. The VERBOSE field sets the level of debugging output, with zero (or omission) being no debugging output. What it is written on screen depends very much on the type of distribution performed in the step. The SEED field resets the random generator with the given seed value before performing the step.

The AGENTS section indicates whether a new agent population is created or loaded from a file, or whether the agent population currently loaded in memory is to be used in the distribution. It also allows to add or change agent attributes before the distribution starts. The OUTPUT section defines whether and what information should be written to disk at the end of the steps, and in which files it should be saved.

As a very simple example, the following code does nothing but saving the current agent population in reloadable, binary format into the file `agents_after_step_10.dat` in the folder defined by the `outfolder` variables, which was set in the VARIABLES section of the control file.

```
[VARIABLES]
outfolder   =  "out/"

[DISTRIBUTION]
...
STEP 10 {
   DESCRIPTION =  "Save agents"
   TYPE  =  none
   OUTPUT {
      STORE_AGENTS   #var(outfolder) "agents_after_step_10.dat"
   }
}
...
```

The LOOPS section defines whether the whole step should be repeated and defines iterator variables, which take on a different value in each loop. For example, the following code would loop over a class variable and a sector variable and repeat all tasks of the distribution step 20 for each combination of class and sector (cf. section C.4 for the use of the `#table` function).

```
STEP 20 {
...
```

```
    LOOPS {
        class = [family, part_time, commercial]
        sector = #table(sectors, 0,  [])
    }
...
}
```

Note: The OUTPUT section will be processed after running through all loops.

Distribution types

Currently, the following types of distributions have been implemented. They will be described in more detail in the following sections.

Table 6.1: Distribution types

TYPE	Description
none	does nothing, only the AGENTS, OUTPUT, DROP_IF, and CLEAN fields an sections will be processed
fixed	deterministic assignment
fixed_repeated	deterministic assignment which is repeated until a specified condition is true
distribute_observation_list	distribute a list of individual assets among agents
assign_by_cdf	distribute assets to agents following a probability distribution function
match	match agents and list of assets respecting two-sided constraints
match_cdf	have the asset distribution match a probability distribution function with two-sided constraints
fit_into_subarea	distribute agents over subareas such that upper limits to attributes in the area are respected
maps_assign_farmstead	spatially distributes farmsteads on a suitability map
maps_assign_plots	spatially distributes agents plots on a suitability map
maps_artificial	generate simplified maps for cases where spatial structure is irrelevant
make_secondary_maps_for_mpmas	create population, cluster, network, sector and catchment maps in MPMAS format

6.2 Generating, removing, saving and importing agents

At the beginning of each step, the user may define that the next step should be performed with a newly generated agent population, the agent population currently in memory, or restored from a previously saved file. If the AGENTS section is completely omitted, or the space between the AGENTS keyword and the opening bracket is empty, the population currently existing in memory is used.

```
AGENTS {
}
```

6.2.1 Generating agents

If an mpmasql function language expression is found between the AGENTS keyword and the opening bracket, it is interpreted as a list of agent identifiers for a new agent population to be created.

Example 1

```
AGENTS [1, 2,  3 ] {
}
```

This would create an agent population consisting of three agents identified by the agent IDs 1,2 and 3 respectively.

Example 2

```
AGENTS #steps(400,500, 2) {
}
```

This would create an agent population consisting of 51 agents identified by the agent IDs 400, 402, 404, . . . , 500.

Example 3

```
AGENTS #table(land, 0, [] ) {
}
```

This would create an agent population based on the table 'land'. Each unique value found in the first column of the table becomes the identifier of one agent (cf. section C.4 for the use of the `#table` function).

Example 4

```
 LOOPS {
   sector   =  [12, 14, 15]
 }
AGENTS  #this(sector) * 1000 +  #steps(1, #table( sectors, 4, [
   #this(sector) ] ), 1)  {
 }
```

In this rather complex example, the algorithm loops over a list of three sectors. For each sector, the number of agents to be created is read from column for of the table sectors (cf. section C.4 for the use of the `#table` function). Agent identifiers have five digits, with the first two indicating the sector and the last three indicating the running number of the agent within the sector.

6.2.2 Removing agents

Individual agents can be dropped from the agent population in the memory at the end of a distribution step using the DROP_IF field, which takes a condition to determine which agents should be dropped. This is done after all loops and before any output is saved.

For example, the following code would drop all agents with an agent ID larger or equal to 4000 at the end of step 30, and all agents whose land attribute is smaller than 10 at the end of step 40.

```
STEP 30 {
...
  DROP_IF  =  #this(agent, id) >= 4000
```

```
...
}

STEP 40 {
...
  DROP_IF  =  #this(agent, land) < 10
...
}
```

6.2.3 Saving agent populations

Agents populations can be saved in two formats: (i) as a tab separated table of agent attributes in human readable text files by using the AGENTS keyword in the OUTPUT section, or (ii) as a binary file, which allows to reload the agent population into `mpmasdist` at a later point of time, by using the STORE_AGENTS keyword in the OUTPUT section.

The following example code would save the agent population in human-readable format in the file ‘agents.tab’ and in binary format in the file ‘agents.dat’ at the end of step 20.

```
STEP 20 {
 ...
  OUTPUT {
        AGENTS   "agents.tab"
        STORE_AGENTS   "agents.dat"
  }
 ...
}
```

(Note: For the use in Linux pipes `mpmasdist` also accepts the keyword STDOUT as a file name in the OUTPUT section.)

The population attribute list saved by the AGENTS keyword does not contain the specifically designated list of agent assets and agent household members (see further below), which can be saved in human-readable format using the keyword ASSIGNMENTS and MEMBERS. This allows an easier separation of agent characteristics, which are only needed during the distribution process and those which should enter as asset or household member information into the model. The member and assigments lists have a file format that facilitates upload to a database.

6.2.4 Restoring agent populations

Agent populations that have been saved with the STORE_AGENTS keyword can be restored the RESTORE keyword with the filename between the AGENTS keyword and the curly brackets, e.g.

```
AGENTS RESTORE "agents.dat" {
}
```

.

6.3 Defining and manipulating attributes and assets

Agents can have attributes that describe characteristics of the agent relevant in the distribution process. Agents can be assigned assets and household members, which are kept in separate lists that

are themselves attributes of the agents. The list of assignments and household members cannot be used to formulate conditions or calculate agent characteristics during the distribution, they are rather intended as a cleaned list facilitating the extraction of those attributes relevant for use in MPMAS later on. It may thus make sense to let the distribution steps first define and distribute assets as attributes, and only at the end transform them to designated asset and household member assignments.

6.3.1 Setting and using attributes

Attributes can be created and changed in the AGENTS section. For example, the following code would create an attribute named liquidity and set it to 1000 for all agents, an attribute named sector, which is equal to the looping iterator sector and an attribute class, which is an integer between 0 and 4 randomly determined for each agent. Each attribute definition needs to go on a separate line in the AGENTS section, and consists of a definition of the attribute name and the value it should receive separated by a #=.

```
AGENTS {
  liquidity #= 1000
  sector    #= #this(sector)
  class     #= #floor( #random_number(5))
}
```

Similar to field definitions in mpmasql, attribute definitions can also create a list of fields with one line of code using e.g. the #table, #foreach, #steps functions or an explicit array. The function #this(attribute, ...) is used to refer to elements in the original list. The function #this(agent, ...) allows to refer to previously defined attributes of the agent. So the following code would create 26 attribute for each agent (named 'male_hhmember_aged_18', 'female_hhmember_aged_18', 'male_hhmember_aged_19', 'male_hhmember_a ...), and these would be set to values according to the user-defined table 'hhmembersbyclass'. For the code to make sense, we assume this table to contain three key columns, the first containing the agent class, the second the age, and the third the gender, while in the fourth column a quantity information might be recorded.

```
AGENTS {
  [male, female] "_hhmembers_aged_" #steps(18, 30, 1)   #=
     #table(hhmembersbyclass, 4, [#this(agent, class),
     #this(attribute, 2), #this(attribute, 1)])
}
```

To set values of attributes as the result of a predefined distribution algorithm, you can use the ON_ASSIGN section, which will only assign values to those agents, which have been selected by the distribution algorithm, and provides specific local variables to refer to the results of the process. The syntax for the ON_ASSIGN section is a bit different as for the AGENTS section: Since, as you will see later on, also attributes of observations can be changed here, we need to explicitly mention that we refer to agent attributes by putting an "agent." in front of the attribute name. Specific local variables will be discussed during the presentation of individual distribution algorithms.

```
  ON_ASSIGN {
     "agent.asset_counter"      #= #this(agent, asset_counter) + 1
  }
```

6.3.2 Removing attributes

The CLEAN section allows to remove agent attributes at the end of the distribution step. For example, the following code would delete the attributes 'asset_counter' and 'unoccupied_soil' from all agents.

```
CLEAN {
  asset_counter
  unoccupied_soil
}
```

6.3.3 Assigning assets and household members

To assign something to the designated list of assets using the ASSIGN section, which is provided by the different types of distribution algorithms. The same is to with respect to designated household members, which are assigned in the MEMBERS section. The most straightforward of the algorithms is the 'fixed' distribution algorithm, which does nothing special and can for example be used to transform agent attributes into designated assets and members:

For example, the following code would assign an asset named 'fruit_plantation' to each agent, which is sized according to the attribute of the same name, but only if the size is greater than zero.

It similarly creates a designated household member for each corresponding attributes. Finally, the attributes are deleted and the assets and member lists are saved to files.

```
STEP 100 {
  DESCRIPTION = "Transform attributes to assets"
  TYPE  =  fixed
  AGENTS {
  }
  ASSIGN {
     #if( #this(agent, fruit_plantation) > 0, fruit_plantation, [])
        #=  #this(agent, fruit_plantation)
  }
  MEMBERS {
     [male, female] "_hhmembers_aged_" #steps(18, 30, 1)    #=
        #this(agent, #this(attribute, 1) "_hhmember_"
        #this(attribute, 2))
  }
  CLEAN {
     [male, female] "_hhmembers_aged_" #steps(18, 30, 1)
     fruit_plantation
  }
  OUTPUT {
     ASSIGNMENTS    #var(outfolder) "agents_assets.tab"
     MEMBERS        #var(outfolder) "agents_hhmembers.tab"
  }
}
```

(Note: each statement needs to go on one line, line breaks in the example only for readability in this manual.)

A slightly enhanced version of the 'fixed' distribution type is provided by the 'fixed_repeated' type, which allows to repeat the assignments formulated in the ON_ASSIGN, ASSIGNMENTS and MEMBERS section while the condition formulated in the DYNAMIC_CONSTRAINTS section are true.

6.4 Distribution algorithms for attributes, assets and household members

mpmasdistoffers a number of predefined distribution algorithms in order to allow flexible random assignments of assets, attributes or characteristics.

6.4.1 Distributing a list of observations

The most basic algorithm named 'distribute_observation_list' randomly allocates a list of observations among agents. Let us assume we have observed a number of fruit plantation in the area and gathered this information in the following table, which was loaded to memory as 'fruit_plantations' with one key column in the TABLES section.

id	crop	size	quantity
1	apple	12	2
2	pear	8	1
3	apple	34	1
4	apple	5	3
5	kiwi	13	1
6	kiwi	1	10
...			

According to the table, we want to have two apple plantations of size 12 (let's say ha) in our population, one 8 ha pear plantation and so on. With the following code, we would tell mpmasdist to randomly distribute all the observed populations among all agents in memory. Each agent has the same probability of receiving a plantation.

```
STEP 120 {
  DESCRIPTION = "Distribute fruit plantations"
  TYPE  =  distribute_observation_list
  AGENTS {
  }
  OBSERVATIONS #table(fruit_plantations, 0, []) {
     fruit    #=    #table(fruit_plantations, 2, [#this(observation,
        id)])
     size     #=    #table(fruit_plantations, 3, [#this(observation,
        id)])
     DEL_count   #=    #table(fruit_plantations, 4,
        [#this(observation, id)])
  }
  ASSIGN {
      #this(observation, fruit) "_plantation"  #= #this(observation,
         size)
  }
}
```

(Note: each statement needs to go on one line, line breaks in the example only for readability in this manual.)

The OBSERVATIONS section of the step generates the list of observations. Between the OBSERVATIONS keyword and the open curly bracket, mpmasdist expects a list of observation IDs, which is in this case simply read from the key column of the fruit_plantations table. Observations have attributes that are defined in the OBSERVATIONS section in a similar way as agent attributes. The local variable #this(observation, ...) allows the use of the observation ID or previously defined observation

attributes. In our case, we read the information in the further table columns an store them in attributes named fruit, size and DEL_count. The DEL_count attribute is a reserved attribute that tells `mpmasdist` how often an observation with an ID should be distributed.

In the example above, assets with a name formed from the content of the 'fruit' attribute of the observation and the suffix '_plantation' are directly assigned to the asset list of the agent selected by the distribution algorithm. The number of assets assigned is set according to the size attribute of the observation.

We could alternatively or additionally manipulate attributes of the selected agent (using an ON_ASSIGN section) or assign household members (using a MEMBERS section).

Exclusiveness and accumulation

The default behavior of the 'distribute_observation_list' algorithms assumes that an agent is eligible only once for assignment of an object with the same observation ID, but that it can receive several observations in the observation list as long as they have different IDs. So in our example, an agent could receive only one 12- ha-apple plantation, but it could well receive a 5-ha-apple plantation or a pear plantation in addition.

This behavior can be changed in both directions. If you assign the attribute 'DEL_exclusive' to an observation and set it to one, an agent that has been selected for an observation cannot be selected for another observation after that in the same step.

```
OBSERVATIONS #table(fruit_plantations, 0, []) {
   fruit    #=    #table(fruit_plantations, 2, [#this(observation,
      id)])
   size     #=    #table(fruit_plantations, 3, [#this(observation,
      id)])
   DEL_count   #=    #table(fruit_plantations, 4,
      [#this(observation, id)])
   DEL_exclusive  #= 1
}
```

(Note: each statement needs to go on one line, line breaks in the example only for readability in this manual.)

Conversely, if you define the attribute 'DEL_accumulate' and set it to one agents may receive several observations with the same ID. Be aware that any assignment in the ASSIGN or MEMBERS section is cumulative, i.e. if an agent already has 12 assets of type apple_plantation and additionally receives 5, it will have 17 afterwards. This is different in the ON_ASSIGN section as you do not add asset-s/members to a list, but manipulate attributes and need to define yourself whether the value is added to the attribute or replaces the old value. For an addition, you could for example write the following (make sure that the agent attribute has been defined before):

```
ON_ASSIGN {
    "agent." #this(observation, fruit) "_plantation"  #=
       #this(observation, size) + #this(agent, #this(observation,
       fruit) "_plantation" )
}
```

(Note: each statement needs to go on one line, line breaks in the example only for readability in this manual.)

More complex compatibility relations can be defined using the DYNAMIC_CONSTRAINTS section explained in more detail in the next section. For example, if we wanted to make sure agents are assigned only one plantation of each fruit type, we could introduce auxiliary attribute 'has_<fruit>' that is zero in the beginning and set to one as soon as the agent gets an asset of that type. The constraint

then makes sure that the agent is not eligible for an observation of a type if the corresponding attribute has been set to one.

```
STEP 120 {
  DESCRIPTION = "Distribute fruit plantations"
  TYPE  =  distribute_observation_list
  AGENTS {
     "has_" [ apple, pear, kiwi]   #= 0
  }
  OBSERVATIONS #table(fruit_plantations, 0, []) {
     fruit    #=    #table(fruit_plantations, 2, [#this(observation,
        id)])
     size     #=    #table(fruit_plantations, 3, [#this(observation,
        id)])
     DEL_count   #=    #table(fruit_plantations, 4,
        [#this(observation, id)])
  }
  DYNAMIC_CONSTRAINTS {
     #this(agent,  "has_" #this(observation, fruit)) == 0
  }
  ASSIGN {
      #this(observation, fruit) "_plantation"  #= #this(observation,
         size)
  }
  ON_ASSIGN {
     "agent.has_" #this(observation, fruit)    #= 1
  }
}
```

(Note: each statement needs to go on one line, line breaks in the example only for readability in this manual.)

One-sided constraints and distribution order

Often not all agents will be eligible for a certain type of observation and the distribution algorithm should be subjected to restrictions. `mpmasdist` distinguishes two types of constraints: FIXED_CONSTRAINTS are checked only once at the beginning of a distribution, i.e. if they contain any #this(observation, . . .) variable before the distribution for an observation is started, or if they do not contain observation specific information even only once at the beginning of the whole step.

For example, if only agents with the sector attribute set to two should be able to receive fruit plantations, the following fixed constraint ensures that only these are included in the list of candidate agents for an assignment.

```
FIXED_CONSTRAINTS {
   #this(agent, sector)    == 2
}
```

(Note: You can formulate any number of constraints in separate lines. Each line has to evaluate to true for the set of constraints to be fulfilled.)

DYNAMIC_CONSTRAINTS are checked before each single random allocation. DYNAMIC_CONSTRAINTS should only be used if the evaluation result of the condition might change during the distribution process, otherwise the repeated evaluation unnecessarily slows down the distribution process.

One example for a dynamic constraint has already been given in the previous section. Another example is ensuring that the agent does not receive more fruit_plantations than the amount of area it

owns. In teh following example the auxiliary agent attribute ‘land_for_fruits’ is introduced and set to the value contained in the land attribute of the agent in the beginning. Every time the agent receives a plantation the attribute is reduced by the corresponding amount of area. reduced

```
STEP 120 {
   DESCRIPTION = "Distribute fruit plantations"
   TYPE  =  distribute_observation_list
   AGENTS {
      land_for_fruits   #= #this(agent, land)
   }
   OBSERVATIONS #table(fruit_plantations, 0, []) {
      fruit    #=    #table(fruit_plantations, 2, [#this(observation,
         id)])
      size     #=    #table(fruit_plantations, 3, [#this(observation,
         id)])
      DEL_count   #=    #table(fruit_plantations, 4,
         [#this(observation, id)])
   }
   DYNAMIC_CONSTRAINTS {
      #this(agent, land_for_fruits)    >= #this(observation, size)
   }
   ASSIGN {
       #this(observation, fruit) "_plantation"  #= #this(observation,
          size)
   }
   ON_ASSIGN {
      "agent.land_for_fruits" #= #this(agent, land_for_fruits) -
         #this(observation, size)
   }
}
```

(Note: each statement needs to go on one line, line breaks in the example only for readability in this manual.)

If you formulate such a constraint, you will want to make sure that the distribution proceeds in a suitable order, usually starting with the observation that is hardest to distribute. Otherwise the only agent with enough land to accommodate the 200 ha apple plantation may not be eligible any more at the time this plantation is distributed, because it already received a 50 ha apple plantation and a 70 ha kiwi plantation.

You can determine the order in which observations will be treated using the special observation attribute ‘DEL_priority’. The lower the value of this attribute the earlier the observation will be distributed (irrespective of its place in the observation list).

In the example above, it would be a good idea to start with the largest plantations, which could be achieved by adding the following line to the OBSERVATIONS section.

```
DEL_priority   #= -1 * #this(observation, size)
```

Complex constraints and matching

While in the presence of a one-sided constrained, ordering observations is usually enough to ensure that as many observations as possible are distributed, more complex constraint structures require the use of a matching algorithm, otherwise receiving a suitable allocation by random allocation may be very unlikely.

`mpmasdist` offers the ‘match’ type of distribution which currently offers matching algorithms based

on the Hungarian method or mixed integer programming (MIP).[1] In both cases, all observations are matched to agents simultaneously, such that an overall likelihood measure is maximized. Likelihoods have to be defined for the agent for each combination of observation and agent in a special section called LIKELIHOODS. This section can consists of any number of lines with a label part and a likelihood value part separated by '#='. Each likelihood value part is evaluated and its value is combined with the likelihood values of other lines by multiplication.

As an example, let us assume that we expect farmers would generally plant around 50% of their area with fruit plantations and that of course the total fruit plantation area cannot be larger that the total land area of the agent.

```
STEP 120 {
  DESCRIPTION = "Distribute fruit plantations"
  TYPE  =  match_cdf
  MATCHING = match_hungarian
  AGENTS {

  }
  OBSERVATIONS #table(fruit_plantations, 0, []) {
     fruit    #=    #table(fruit_plantations, 2, [#this(observation,
        id)])
     size     #=    #table(fruit_plantations, 3, [#this(observation,
        id)])
     OBS_multiply   #=    #table(fruit_plantations, 4,
        [#this(observation, id)])
  }
  FIXED_CONSTRAINTS {
     #this(agent, sector)    ==    2
  }
  LIKELIHOODS {
        land_restr  #= #this(agent, land) >= #this(observation, size)
        50perc      #= 1 / #abs(0.5 *  #this(agent, land) -
           #this(observation, size) ) + #random_number(0.00001)

  }
  ASSIGN {
      #this(observation, fruit) "_plantation"  #= #this(observation,
         size)
  }

}
```

The likelihood part labeled 'land_restr' – and due to the multiplication the whole likelihood – is zero for all agent-observation combinations, in which the plantation is larger than the agent's land. In all other cases, the 'land_restr' part is one and the overall likelihood is determined by the second part labeled '50perc', which results from dividing one by the absolute difference between plantation size and half of the agent land and adding a small random number. This random component ensures that the algorithm arrives at a unique solution even if two combinations of agent and observation have the same absolute difference. The magnitude of the random component compared to the difference quotient determines how important the restriction is.

Some further notes to the above example: In the matching algorithm, each observation ID can be allocated only once, so the DEL_count attribute cannot be used. Instead, the OBS_multiply attribute clones the observation as many times as requested giving each clone a separate ID.

While using non-observation specific FIXED_CONSTRAINTS is possible, DYNAMIC_CONSTRAINTS

[1] MIP is still experimental and not further described here.

sections will be ignored in matching algorithms.

A likelihood of zero does not set a hard constraint if the number of observations is smaller or equal to the number of agents. The algorithm will distribute all observations and if necessary choose a combination with zero likelihood in case there is no other better option available. To avoid this you can e.g. include observations of size zero and making sure they have a small positive likelihood.

If the number of observations is larger than the number of agents, the user can use the field OBS_NONUSE_PENALTY to name an observation attribute that defines the likelihood of an observation not being assigned. mpmasdist will include artificial non-use agents and assign a non-use penalty equal to the value of the chosen attribute to the combination of the observation with a non-use agent.

```
STEP 120 {
   DESCRIPTION = "Distribute fruit plantations"
   TYPE  =  match_cdf
   MATCHING = match_hungarian
   OBS_NONUSE_PENALTY_ATT  =  nonusepenalty
   ...
   OBSERVATIONS #table(fruit_plantations , 0, []) {
   ...
      nonusepenalty  #= 1000000000 + random_number(1000)
   ...
   }
   ...
 }
```

Internally, the algorithm minimizes negated log-likelihoods, i.e. it transforms the values specified in LIKELIHOODS by taking logarithms and multiplying with minus one. The penalty is taken as is and directly corresponds to the negated log-likelihood, i.e. to make non-use a matter of last resort, it should probably be a very high positive number.[2]

Remaining observations

If not all observations could be distributed in a 'distribute_asset_list' step (e.g. due to unfulfilled constraints or less agents than observations with exclusive assignments), mpmasdist will inform the user printing a message on the screen and add the unassigned observations to a special list. If desired, the user can then make a second attempt to distribute these observations in a subsequent step, e.g. relaxing constraints or assigning to a different group of agents.

This is achieved by using the keyword NOTASSIGNED as observation list.

```
 OBSERVATIONS NOTASSIGNED {

 }
```

Alternatively, the list can be stored at the end of a step using STORE_NOTASSIGNED <filename> in the OUTPUT section, and restored at a later point of time using RESTORE <filename> as observations list. (Saving in human-readable format is also possible using the NOTASSIGNED <filename> keyword in the OUTPUT section.)

[2]Note: By default the algorithm uses a penalty of plus infinity as pseudo-log likelihood for agent-observation combinations with zero likelihood. This can be adapted assigning a USE_ZEROL_PENALTY field of the distribution step, so allowing a balancing between non use and violations of zero likelihoods.

6.4.2 Assignment following a probability distribution

To assign assets, attributes or household members by Monte Carlo sampling from a distribution function, `mpmasdist` offers the 'assign_by_cdf' distribution type.

Specification of the distribution function

Probability distribution functions are defined in the OBSERVATIONS section. Each distribution function is understood as a separate observation, from which each agent receives only one value (so exclusiveness is ensured by default).

Distribution functions can be specified in the form of frequency distributions, cumulative distribution functions or quantile functions (= inverse cumulative distribution functions). `mpmasdist` will internally transform them into quantile functions for Monte Carlo sampling.

As a frequency distribution The following table with two key columns named fruit_plantations contains three frequency distributions, one for apple plantations, one for pear plantations one for kiwis.

crop	size	quantity
apple	12	2
apple	34	1
apple	5	3
pear	8	1
kiwi	13	1
kiwi	1	10
...		

The frequency distribution could then be specified as follows:

```
OBSERVATIONS #table(fruit_plantations, 0, []) {
    CDF_ftype   #= freq
    CDF_freq_for_value_#table(fruit_plantations, 0,
       [#this(observation,id)])   #= #table(fruit_plantations, 3,
       [#this(observation,id), #this(attribute, 1)])

}
```

(Note: each statement needs to go on one line, line breaks in the example only for readability in this manual.)

As a cumulative distribution The following table with two key columns named fruit_plantations contains three cumulative probability distributions, one for apple plantations, one for pear plantations one for kiwis.

The cumulative distribution could then be specified as follows :

```
OBSERVATIONS #table(fruit_plantations, 0, []) {
    CDF_ftype   #= cdf
    CDF_bin_for_value_#table(fruit_plantations, 0,
       [#this(observation,id)]) #= #table(fruit_plantations, 3,
       [#this(observation,id), #this(attribute, 1)])
```

crop	size	cumulative_percentage
apple	0	0
apple	5	50
apple	12	83.33
apple	34	100
pear	0	0
pear	8	100
kiwi	0	0
kiwi	1	91
kiwi	13	100
...		

```
}
```

(Note: each statement needs to go on one line, line breaks in the example only for readability in this manual.)

As a quantile function The following table with two key columns named fruit_plantations contains three cumulative probability distributions, one for apple plantations, one for pear plantations one for kiwis.

crop	cumulative_percentage	size
apple	0	0
apple	50	5
apple	83.33	12
apple	100	34
pear	0	0
pear	100	8
kiwi	0	0
kiwi	91	1
kiwi	100	13
...		

The quantile function could then be specified as follows :

```
OBSERVATIONS #table(fruit_plantations, 0, []) {
     CDF_ftype   #= cdf
     CDF_value_for_bin_#table(fruit_plantations, 0,
        [#this(observation,id)]) #= #table(fruit_plantations, 3,
        [#this(observation,id), #this(attribute, 1)])
}
```

(Note: each statement needs to go on one line, line breaks in the example only for readability in this manual.)

Interpolation and discreteness

The distribution function is usually given as an empirical function indicating only a few points on the function. For sampling, it has to be interpolated. By default, `mpmasdist` uses linear interpolation (code: 'linear'). By adding the attribute 'CDF_interpolation' to the OBSERVATIONS section, you can choose

between stepwise interpolation (code: 'step') or cubic Hermite-spline interpolation (code: 'pmch'). E.g. for the later:

```
OBSERVATIONS #table(fruit_plantations, 0, []) {
    CDF_interpolation #= pmch
}
```

If you want to ensure only integer values are assigned also after interpolation, you can add the CDF_discrete attribute and set it to one.

```
OBSERVATIONS #table(fruit_plantations, 0, []) {
    CDF_discrete   #= 1
}
```

Assignment

The result of the sampling is saved in the local variable #this(observation, CDF_value_assigned), which can be used to make the assignment, e.g.

```
  ...
   TYPE = assign_by_cdf
  ...
  OBSERVATIONS #table(fruit_plantations, 0, []) {
      CDF_ftype   #= cdf
      CDF_value_for_bin_#table(fruit_plantations, 0,
          [#this(observation,id)]) #= #table(fruit_plantations, 3,
          [#this(observation,id), #this(attribute, 1)])
      fruit       #= #this(observation, id)
  }
 ASSIGN {
    #this(observation, fruit) "_plantation"  #= #this(observation,
        CDF_value_assigned)
 }
...
```

Constraints

FIXED_CONSTRAINTS and DYNAMIC_CONSTRAINTS can be added to the model in a similar way as for the 'distribute_observation_list' type. The local variables #this(observation, CDF_value_assigned) can be used to in the constraint formulation to test the drawn value.

E.g.

```
DYNAMIC_CONSTRAINTS {
  #this(agent, land) >= #this(observation, CDF_value_assigned
}
```

After adding constraints the sampling result will probably not correspond to the original probability distribution anymore, because the constraints lead to selective truncation of the sampled distribution for each agent.

To avoid this, you can add the attribute 'CDF_ensure_full' to the observation and set it to one. `mpmasdist` will then first draw as many values from the distribution as there are agents and then use the ordered 'distribute_observation_list' algorithm to try to distribute all the full range of the distribution function among the agents. The value of the attribute 'CDF_ensure_full_sorting' is used to

determine whether the distribution should start with the lowest (if set to -1) or the highest (default or set to 1) value drawn.

The attribute CDF_priority determines, which observation (i.e. which of several distribution functions) is used first.

For more complex constraints, the distribution type 'match_cdf' combines the assign_by_cdf type with the 'match' distribution type explained above. It samples as many values from a specified distribution function as there are agents and then finds the matching that maximizes the likelihood. The syntax is similar to the syntax of the 'match' type except that the OBSERVATIONS section uses the syntax for probability distributions explained for 'assign_by_cdf'.

Drawing only from parts of the distribution function

In some cases, you may want the distribution algorithm to draw only from a part of the distribution function, e.g. because you used a copula to define a joint distribution function. In this case, you can add the attributes 'CDF_part_lb_attribute' and/or 'CDF_part_ub_attribute' to the OBSERVATIONS section to indicate the name of an agent attribute that contains an lower, respectively upper quantile of the distribution function between which the agent should be located.

For example, the following code would tell `mpmasdist` to look for the 'apple_lb' and 'apple_ub' attributes of the agent, read them, and if they contained for example the values 40, respectively 60, it would draw a random value from the third quintile of the qunatile function.

```
 OBSERVATIONS [apple] {
   CDF_part_lb_attribute   =  #this(observation, id) "_lb"
   CDF_part_ub_attribute   =  #this(observation, id) "_ub"
}
```

6.5 Spatial distribution

`mpmasdist` currently offers two algorithms to create plot maps for MPMAS. The first is a quick version for cases, where spatial relations are not really important and you just want to create maps in order to comply with the MPMAS format for representing the soil endowments of agents (as well as cluster, network etc.). The second one is a more sophisticated distribution algorithm, which use existing land use maps to allocate farmsteads and plots in a realistic pattern.

6.5.1 Quick maps

For quick maps, `mpmasdist` offers the distribution type 'maps_artificial', which simply fits the plots of agents into the smallest rectangle possible: It groups all plots of all agent sequentially and breaks them up into as many rows as required to fit the input into a map of a user-defined North-South extension.

```
 STEP 40 {
   DESCRIPTION =  "Make artificial property, farmstead and soil maps"
   TYPE  =  maps_artificial
   AGENTS {
   }
   SOILCODES {
      black_soil    #= 0
      clay_soil           #= 1
      sandy_soil          #= 2
   }
```

```
    MAPHEIGTH = 10
    CELLSIZE  = 100
    OUTPUT {
       FARMSTEADMAP   "gis/CatchMap00Farm.txt"
       PROPERTYMAP    "gis/CatchMap00Prop.txt"
       SOILMAP     "gis/CatchMap00Soil.txt"
    }
}
```

(Note: each statement needs to go on one line, line breaks in the example only for readability in this manual.)

The SOILCODES section defines, which attributes of the agents shall be transformed to plots and by which code these shall be represented in the map. `mpmasdist` rounds the values contained in the attributes to full integers. So, for example, an agent, whose 'black_soil' attribute contains the value 18.3 would receive 18 cells with code 0 in the map. The MAPHEIGTH field defines the North-South extension of the map in cells, and the CELLSIZE field defines the sidelegnth of the cell in meters.

The keywords FARMSTEADMAP, PROPERTYMAP, and SOILMAP tell `mpmasdist` to save the corresponding map in the indicated file.

6.5.2 More realistic maps

The more sophisticated distribution algorithm of `mpmasdist` requires two distribution steps. In the first step, the farmsteads of the agents are distributed over the map, and in a second step, the plots are allocated more or less closely around the farmsteads. In both steps, the distribution is based on an existing suitability map that provides the map extents and whose cell values indicates, which cells are suitable for farmsteads, respectively certain types of plots.

Distributing farmsteads

The distribution type for distributing farmsteads is 'maps_assign_farmstead'. It expects only two fields: The MAP field indicates, which of the maps loaded in the MAPS section of the control file should serve as the basis for the distribution, and the SUITABLE field that indicates the list of cell values suitable for farmsteads.

```
STEP  20 {
    DESCRIPTION = "Assign farmsteads"
    TYPE  =  maps_assign_farmstead
    AGENTS  {

    }
    MAP   =  land
    SUITABLE =  [5, 9]
    OUTPUT {
       FARMSTEADMAP   "/gis/CatchMap00Farm.txt"
    }
}
```

(Note: each statement needs to go on one line, line breaks in the example only for readability in this manual.)

If a farmstead map already exists in memory from a previous step, it is going to be updated allowing the user to split farmstead distribution over several steps.

An ON_ASSIGN section can be added to update agent attributes when the farmstead is set, e.g. one could record the soil type at the location of the farmstead using the #mapcell function and the special agent attributes ‘farm_x’ and ‘farm_y’ from a user-loaded map called ‘soil_in_studyarea’.

```
 ON_ASSIGN {
   "agent.soil_at_fstd" #= #mapcell(soil_in_studyarea, #this(agent,
      farm_x), #this(agent, farm_y) )
}
```

Distributing plots

The plot distribution algorithm of `mpmasdist` is designed to ensure a random distribution of plots to agents, while taking account of the fact that plots of farmers are usually agglomerated and not completely randomly distributed over the maps. Plots in reality are usually larger than one grid cell and `mpmasdist` tries to form contiguous stretches of land belonging to the same farmer. On the other hand usually not all land of a farmer forms one contiguous plot, so the algorithms tries to find a balance. (This reflects the typical central European situation, for other situations an adaptation of the algorithm should be discussed with the `mpmasdist` developer team.)

Contrary to the quick algorithm, the more sophisticated algorithm does not transform attributes into cell values, but looks for suitable cells in an existing suitability map for each type of plot linked to an agent attribute.

What the algorithm does: The algorithm loops over plot types distributing area of one type first and then the next, so the following is repeated for each plot type:

1. Reclassify the landscape map into suitable and non-suitable cells according to the specified condition.
2. Mark all cells containing a farmstead and all those which have already be assigned as property in previous steps as used.
3. Go through the map and identify polygons, i.e. neighbouring cells which are suitable and unused, which could form a plot. Make a list of these polygons, giving them an ID and record their location (i.e. one handle cell) and size, and make a polygon map, i.e. a map which contains the IDs of the polygon a specific cell belongs to.
4. Order the agents by a prespecified priority criterion (e.g. start with the largest or smallest farm) and form a queue with the farm with the highest priority in front
5. Take the first agent from the queue and randomly draw a plot size smaller or equal to the remaining area of the current plot type still to be distributed for this agent and smaller than the largest polygon still available in the map (distribution to be drawn from can be changed by the user).
6. Look for a suitable polygon of size equal to or larger than the randomly drawn plot size in the map
    - Start at the agents farmstead look at the ring of 8 neighbouring cells to see if any of them belongs to a polygon large enough. If yes select that polygon and go to 7. If not, look at the 16 cells around this, and so on making the ring wider and wider
    - If after 20 widening iterations a suitable cell has not been found, check if the agent already has a plot and start the same procedure at a cell of this plot.
    - If there is no plot or all plots have already been checked and still there is no suitable polygon, draw one randomly from the list of polygons (wherever it is in the map)

7. Assign the required number of contiguous cells, starting with the cell found, using a random walk to neighbouring suitable and unused cells. Update the property map to record ownership of these cells.

8. If after that no cell is left in the polygon, delete the polygon from the polygon list. If cells are left, check whether the polygon has been split into two polygons. Adapt the polygon size in the list, and add a new polygon to the list if necessary.

9. If the agent has more area to be distributed, re-append it to the end of the agent queue.

10. The algorithm stops if there is no agent left in the queue, or (with a warning) when no suitable cells are left in the map.

Implementation in the control file: The `mpmasdist` distribution type for plot distribution is 'maps_assign_plots'. It expects that a farmstead map already exists in memory. This can either be created with the 'maps_assign_farmstead' distribution type or loaded from file in the [MAPS] section of the control file assigning it the reserved name FARMSTEADMAP.

Already existing property maps in memory are updated allowing the user to split plot distribution over several steps. An existing property map can be loaded from file in the [MAPS] section of the control file assigning it the reserved name PROPERTYMAP.

```
STEP 30 {
   DESCRIPTION = "Assign_plots"
   TYPE  =  maps_assign_plots

   MAP   =  land
   AGENTS {

   }
   SUITABLE {
       black_soil #= [112, 114, 156]
       clay_soil     #= [202, 297, 231]
       sandy_soil #= [14, 34, 89, 70]
   }
   AVGPLOTSIZE = 10
   DISTRIBUTE_ORDER  =  1

   OUTPUT {
      PROPERTYMAP    #var(LOCATION)"/gis/" #var(SEED)
         "/CatchMap00Prop.txt"
   }
}
```

(Note: each statement needs to go on one line, line breaks in the example only for readability in this manual.)

The MAP section indicates the user-loaded suitability map and the SUITABLE section list for each agent attribute that represents a type of land the cell values indicating suitability for this land type in the suitability map.

AVGPLOTSIZE controls the tentative number of cells that should be agglomerated to one plot.[3] The optional field DISTRIBUTE_ORDER controls whether the algorithm starts with the agents with largest (if set to 1, default) – or smallest (if set to 2) value for the plot type to be distributed.

[3] The plot size is randomly drawn from a probability distribution which has AVGPLOTSIZE as its mean. If the optional field PLOT_PDF is set to one (default) this distribution is a normal distribution with $\sigma = 2 * AVGPLOTSIZE$, if it is two, it is a uniform distribution.

Note: For many agents and large farm sizes, the algorithm can take very long (several days) to complete the task.

6.5.3 Making population, cluster, network, sector and catchment maps

Once farmstead and property maps have been created mpmasdist can automatically create the other MPMAS input maps (except soil maps). The distribution type 'make_secondary_maps_for_mpmas' requires only the special agent attributes 'MASPopulation', 'MASNetwork', etc. to be defined for each agent. The file names for the maps are defined using the corresponding keywords in the OUTPUT section.

```
 STEP 40 {
   DESCRIPTION =  "Make other maps for MPMAS"
   TYPE  =  make_secondary_maps_for_mpmas
   AGENTS {
      MASPopulation  #= 0
      MASCluster  #= #this(agent, cluster)
      MASNetwork  #= 0
      MASSector   #= #this(agent, sector)
      MASCatchment   #=    0
   }

   OUTPUT   {
      POPULATIONMAP      "/gis/CatchMap00Pop.txt"
      CLUSTERMAP      "/gis/CatchMap00Clu.txt"
      NETWORKMAP      "/gis/CatchMap00Netw.txt"
      SECTORMAP       "/gis/CatchMap00Sector.txt"
      CATCHMENTMAP       "/gis/CatchMap00Catchment.txt"
   }
}
```

(Note: each statement needs to go on one line, line breaks in the example only for readability in this manual.)

6.6 Varying agent populations

As mentioned in section 6.1.2, the values of user-defined variables may be overridden when running mpmasdist by using the command line option --set.

This may be used to generate variations of the agent population for uncertainty analysis, e.g. by varying the random seed or a constraint.

For example, the following setup would allow to create different distributions of fruit plantations among the agents by repeatedly running mpmasdist. The setup also makes sure that the output is saved in different folders named after the seed and that this folder is created if it does not exist.

```
...
[VARIABLES]
SEEDFRUITS     =  2761
outfolder         =  "out/" #var(SEEDFRUITS) "/"
...
[CONTROL]
MKPATHS  =  ["out/" #var(SEEDFRUITS) "/"]
```

```
...
[DISTRIBUTION]
...
 STEP 120 {
   DESCRIPTION = "Distribute fruit plantations"
   TYPE  =  distribute_observation_list
   AGENTS {
   }
   SEED  =  #var(SEEDFRUITS)

   OBSERVATIONS #table(fruit_plantations, 0, []) {
      fruit    #=    #table(fruit_plantations, 2, [#this(observation,
         id)])
      size     #=    #table(fruit_plantations, 3, [#this(observation,
         id)])
      DEL_count   #=    #table(fruit_plantations, 4,
         [#this(observation, id)])
   }
   ASSIGN {
       #this(observation, fruit) "_plantation"  #= #this(observation,
          size)
   }
   OUTPUT {
      ASSSIGNMENTS   #var(outfolder) "agent_assets.tab "
   }
 }
```

(Note: each statement needs to go on one line, line breaks in the example only for readability in this manual.)

If the setup above was contained in the control file 'example.dst', repeated calls to `mpmasdist` could look as follows:

```
mpmasdist example.dst --set "SEEDFRUITS=123"
mpmasdist example.dst --set "SEEDFRUITS=3456"
mpmasdist example.dst --set "SEEDFRUITS=789"
```

Bibliography

Schreinemachers, P. [2006], *The (Ir)relevance of the Crop Yield Gap Concept to Food Security in Developing Countries. - With an Application of Multi-Agent Modeling to Farming Systems of Uganda.*, Göttingen.

Appendix A

Command line options

A.1 mpmasql

```
mpmasql {<options>} <control file>  {<options>}
```

Table A.1: Command line options for `mpmasql`

Option	Description
-c, --compact	use compact matrix format (only for use with mqlmatrix)
-h, --help, -?	print this help
-n, --nomas	do not check for compatibility with MP-MAS format
-O <directory>	override output directory set in .ini file (for both uncommented and commented files)
-S	single matrix mode
-v	print version information
-x, --nocomment	suppress writing of commented files
-r, --runonly	create only run script
--set "<variable>=<value>"	change value of .ini variable
Additional output	
-a1	report on matrix consistency
-a2	produce files for EDICMatlab routines
Verbosity flags:	
-l1	show conversion control settings
-l2	show model configuration
-l3	verbose when reading model structure file
-l4	verbose when processing scenario adaptations
-l5	verbose when binding variables and tables
-l6	verbose when preparing model elements
-l7	verbose when parsing field and value definitions

-l8	verbose when sorting matrix elements
-l9	verbose when creating input files

Hint: Especially when using verbosity flags and converting many scenarios, mpmasql prints a lot of text to STDOUT and your shell environment might be flooded with text. You might not be able to read all text in your terminal, because you are not able to scroll back as far as necessary. To check all screen messages, you can pipe the output into a file, you can later open with an editor, e.g.

```
mpmasql <inifile> <flags>   > dbg.txt
```

Under Linux, you can of course also use a pipe to the `less` or `most` text pagers:

```
mpmasql <inifile> <flags>  | less
```

A.2 mqlmatrix

```
mqlmatrix <name\_of\_matrix\_file> <file format option> <other
    options>
```

Table A.2: Command line options for `mqlmatrix`

File format:		
(specifying one of these formats is obligatory)		
-F		expect matrix in failed format (.mip/.mpx)
-S		expect matrix in single agent format (.mtx)
-D		expect matrix in debug format (.mip/.mpx)
-V <subformat>		expect matrix in mpmas input file format (.dat)
	<subformat>	
	0:	basic format
	1:	with second production stage ("harvest milp")
	2:	with advanced consumption model
	11:	TSPC, with second production stage (not implemented)
	12:	TSPC, with advanced consumption model (not implemented)
--compact, -c		expect tableau in compact format (only in combination with -V and -S)
Override default settings:		
-A <filename>		specify file containing activity description
-C <filename>		specify file containing constraint description
-O <directory>		override output directory
-I <directory>		specify directory containing description files (if not specified, the directories are read from mqlmatrix.conf in the working directory and names of description files are guessed from the matrix file name)

-M <filename>	filename of MilpCheck executable (default: /usr/local/bin/MilpCheck)
-XP <command>	set program for paging (defaults: most | less)
-XS <command>	set spreadsheet program for table views (defaults: gnumeric | oocalc)
Others:	
-p	print matrix in single agent format (.mtx) and exit
-R	import solution vector
-h, -?, --help	print this help
-v	print version information
--risk_aversion <value>	set risk aversion coefficient for evaluation of imported quadratic solution to <value>

A.3 mpmasdist

```
mpmasdist {<options>} <control file>  {<options>}
```

Table A.3: Command line options for `mpmasdist`

Option	Description
-v	print version information
--set "<variable>=<value>"	set <variable> defined in the control file to <value> for this run

A.4 mpmas

Hint: Contrary to the other programs and most other standard Linux applications, there is no space between the mpmas options and their argument. E.g. its has to be `-Ntest_` and not `-N test_`.

```
mpmas   -N<name\_of\_scenario\_file>  <other options>
```

Table A.4: Command line options for `mpmas`

Required:	
-N	is followed without blank by the simulation name
Optional:	
-A	spatial maps ... see BasicData (Second sheet)
-C	set filename of canal efficiencies (EDIC)
-D	save maps for debugging when coupled with external crop model
-E	set filename of lambda expectation parameters
-H	set water surplus factor (EDIC)
-I	is the path to the input folder, which must contain two subfolders /dat and /gis
-M	export agent LP matrices in compressed format (.mpx)
-O	is the path to the output folder, which must contain subfolders /out and /out/test
-P	set path to GIS input (overwrites default folder name)

-R	set digits on the "right" side of decimal point, used for rounding floating points
-S	assign seed value for random events after lottery (integer value)
-T	specify specific test runs (see list below)
-U	set upper bound (maximum absolute floating point in MILP)
-X	set path to xml files
-Y	set number of simulation periods (i.e. complete earlier than specified in input files)
-Z	specify name of file for dynamic switches (previously flag -D)

Debugging:

-T1	Write all input files back to /out/test folder
-T2	Print detailed water-related info to screen and file (wAg-files)
-T3	Wait for user input at end of year
-T4	Save expected and actual water-supply data for all agents in population (cannot be used together with T9)
-T5	Save expected and actual water-demand data for all agents in population (cannot be used together with T9)
-T6	Write agents' water data to file – if activated in SAVE_AGENT(fstd)
-T7	Save expected and actual yields for all agents in population, before and after estimating yields of missing crops
-T8	Write all random number that were generated to file in test folder
-T9	Save expected and actual yields for all agents in population (cannot be used together with T4)
-T10	Allocate water demand spatially
-T11	Save expected and actual prices for all agents in population
-T12	Testing input routine, trace model run
-T13	Exchange maps with external crop model based on files (and not TDT - "Suppress coupling")
-T14	Write endogenous prices to files
-T15	Basicdata: reading translation table
-T16	Allocate crops to maps but read crop yields from external file ("Look-up table")
-T17	Save agent crop mixes and solution vectors (before and after spatial correction for maps)
-T18	Ignore sub-pixel crops in land-use maps but keep them in production solution
-T19	Debug information for exchanging land-use and crop yield look-up tables
-T20	Raster2D: exporting irrigation table
-T25	Landscape - construction of sectors, landscape, cells
-T28	Outputs for spatial allocation to cells
-T29	De-activate sectors during initialization with special input file (for use in Mpmas stand-alone)
-T30	For coupling, reads some data with ending "_coupl.dat" instead of "dat"
-T31	Evaluate Edic - Routing, etc
-T32	Export sector-wise land use data (using InputData class)
-T33	Crop-water module: simple checks for plant water demands, irrigation water and water deficits
-T34	Read input data - reading out all Dat and Gis files
-T35	Prints a dot on screen after each 20th agent
-T36	Suppress all outputs
-T37	Medium level of verbosity for OSL messages (-T1 is highest level of verbosity)
-T38	Used in deactivated Wasim coupling code

-T39	Dissecting maps into smaller, sector-wise maps writing sector Gis to file
-T41	Debug shrinking of maps
-T42	Debug info for land rental market
-T43	Debug info for producer organizations
-T50	Linux debugging - verbose, not recommended
-T51	Edic outputs extended (not recommended)
-T52	Translation of land-use grid / joining into larger map
-T53	Transforming units of inflows from Wasim routing model into MP-MAS
-T55	Minmax
-T56	Communicate all data
-T57	Table of Etr
-T60	Livestock
-T65	Prints detailed information about the updating of household composition at the end of the period onto the screen
-T66	Saving image of livestock list (still under development)
-T67	Write results from water assignment to catchment file
-T68	Save results of asset assignment (detailed log-file of lottery plus .lt1 and .lt2 files)
-T69	Exogenous land-use scenarios (EDIC calibration run, Land uses fixed by sector)
-T70	Save agent income data
-T71	Print entries for “Forced Solution” to screen
-T72	Always stop after random assignment
-T73	Stop or continue after random assignment (Waits for user input)
-T74	Stop after first Investment Lp solved
-T75	Stop or continue after first Investment Lp solved (Waits for user input)
-T76	Stop after first Production Lp solved
-T77	Stop or continue after first Production Lp solved (Waits for user input)
-T78	Save all MILP as MPS files
-T79	Write agent objective values in Investment and Production LPs into files
-T80	Do wasim control (Read external land-use maps and irrigation key file) and quit then
-T81	Suppress initialization of Wasim coupling water constraints and expectations set to 9999 (not constraining)
-T82	Print header lines to MP-MAS output files
-T88	Test consistency of maps (Chile hard-coded)
-T89	Create sector maps according to inflowIDs of ActWaterRights (Ghana hard-coded)
-T90	Catch segmentation faults in OSL when in solving mode 4 (only for Linux)
-T91	Save updated ’original’ LP matrix for producer organizations
-T92	Save updated ’original’ LP matrix for farm household and other farm agents
-T93	Suppress saving pre-loaded MILPs after failure
-T94	Suppress saving OSL-loaded MILPs after failure
-T98	Reads LP input file ignoring text (not sure if this really works)
-T99	Interrupts model run and waits for user input

Note: A useful set of flags to debug input files is: -T1 -T34; sometimes -T99 might be of help. Note that -T1 writes all input dat files to the /out/test folder as tst files (very useful when checking what the

program actually reads from files).

Appendix B

mpmasql file reference

This chapter gives an overview of the structure and a reference of all possible entries in the `mpmasql` control and model design files.

Comments In all of the files, a line starting with an apostrophe (') is ignored and can be used for comments. You can also stretch comments over several lines with block comments. Block comments start with `/*` and end with `*/`.

```
 ' this is a comment line and will be ignored by mpmasql

/*
This is also a comment, which
spans over several lines
*/
```

MFL expressions At many locations in the files, you can use `mpmasql` function language (MFL) to write expressions to specify that a certain value depends on another value, e.g. the scenario, the specific instance of a class or another variable. The reference of all available MFL functions and operators is given in chapter C.

B.1 Control file

The control file contains settings that instruct `mpmasql` which model design and data sources to use to build the model, where to write and how to name the output, and which scenarios to create. It contains a number of entries of the form:

```
<control parameter>  = <scalar/list: MFL expression>
```

The SCENDEF, RHSDEF, SCENLIST, RHSLIST and RHS_SOILTAB entries are special, because they expect a table definition.

```
tablename   = (tabletype) {   tablesource:
                    an SQL statement for type sql,
                    a filename for type ascii,
                    or a comma separated list for type list
                      }
```

All of the tables can be of type sql or ascii. SCENLIST and RHSLIST can also be of type list. For RHSDEF you can specify a type `transform_assets` with empty curly brackets, indicating that you want right hand sides to be automatically created from the ASSET_ENDOWMENT, HOUSEHOLD_COMPOSITION and RHS_SOILTAB tables.

Apart from the predefined control parameters listed in table B.1, you can also define own user variables (with the same syntax), which can be used in MFL expressions defining the tables for SCENDEF, SCENLIST, RHSDEF and RHSLIST using the `#var()` function. All control parameters and user variables of the .ini file can be overwritten at program call using the `--set` flag (see section A.1). In this way you can generate various scenario sets using a single .ini file.

Table B.1: Parameters in the control file

Entry		
[INPUT]	optional	Section heading
CONFIGURATION	required	Name and path of the configuration file (e.g.: tutorial.cfg)
STRUCTURE	required	Name and path of the structure file (e.g.: tutorial.cll)
DATA	required	Name and path of the data file (e.g. tutorial.var)
DBTYPE	required	Type of the database, or more specifically of the perl DBD to be used (e.g. mysql).
DB	required	Name of the database to connect to (e.g.: mpmasql_example_db:localhost)
DBUSER	required	Username for the database
DBPWD	optional	Password for the database
[OUTPUT]	optional	Section heading
OUTDIR	optional	directory the input files shall be written to (ending with a / under Linux). By default testoutput/ is used.
OUTDIR_CM	optional	directory the commented input files shall be written to (ending with a / under Linux). By default testoutput/ is used.
OUTDIR_S	optional	directory where standalone matrices shall be written to (ending with a / under Linux). By default testoutput/ is used.
OUTDIR_S_CM	optional	directory where the commented standalone matrices shall be written to (ending with a /). By default testoutput/ is used.
[RUN SCRIPT]	optional	section heading
RUNSCRIPTDIR	optional	directory where MP-MAS invocation scripts should be written to
EXEC	optional	path and name of the MP-MAS executable to be used (default: mpmas, i.e. the standard executable that has been installed into /usr/local/bin/ when installing/updating mpmasql.)

RUNINDIR	optional	value for the -I flag of MP-MAS
RUNOUTDIR	optional	value for the -O flag of MP-MAS
RUNTDTDIR	optional	value for the -X flag of MP-MAS
RUNCNLDIR	optional	Directory where EDIC canal files are written
FLAGS	optional	other flags for MP-MAS
[SCENARIOS]	optional	section heading
PREFIX	required	A prefix for all MP-MAS input file names for all scenarios (Note: required entry, but assignment of empty value is possible)
SCENLIST	optional	list of scenarios the conversion script has to convert input files for
SCENDEF	optional	table with scenario definitions
[RHS]	optional	section heading
RHSLIST	optional	list of RHS scenarios for single matrix mode
RHSDEF	optional	table with RHS scenario definitions for single matrix mode
RHS_SOILTAB	optional	table with soil information if RHSDEF (transform_assets) is used
RHS_MAPDIR	optional	directory with MP-MAS input maps if RHSDEF (transform_assets) is used
[DEBUG]	optional	section heading
SKIP	optional	list of input files to be skipped
DBGMILPS	optional	list of agents, for which debugging MILPs shall be saved by MP-MAS
[MASPIPE]	optional	section heading
MASPIPESERVER	optional	LUE Group Simulation Server to use for remote simulation
MASPIPEUSER	optional	LUE Group Simserv User for remote simulation
MASPIPEEXEC	optional	LUE Group Simserv Executable identifier for remote simulation
MASPIPEVERSION	optional	Model Version Identifier for remote simulation
[STATA SCRIPT]	optional	section heading
STATA_ROOT	optional	root directory for stata scripts (Default ./)
STATA_MEM	optional	argument for ‘set mem’ command in stata script (Default: 1000M)
STATA_MAXVAR	optional	argument for ‘set maxvar’ command in stata script (Default: 5000)
STATA_MPMASOUT	optional	directory where MPMAS output is located (Default: RUNOUTDIR/out/)
STATA_MPMASLAB	optional	directory where labeling and group variable files are located (Default: RUNINDIR/input/dat/)
[R SCRIPT]	optional	section heading
R_ROOT	optional	root directory for R scripts (Default ./)

R_MPMASOUT	optional	directory where MPMAS output is located (Default: RUNOUTDIR/out/)
R_MPMASLAB	optional	directory where labeling and group variable files are located (Default: RUNINDIR/input/dat/)
R_TO_WORKSPACE	optional	whether data frames shall be kept in the workspace for each scenario (0/1, default: 0)
R_TO_TAB	optional	whether data frames shall be exported as tab-separated text files (0/1, default: 1)
R_ALL	optional	whether all raw output shall be kept/exported (0/1, default: 0)
R_JOIN_RAW	optional	whether a combined data frame of raw output (incl. performance, rhs, lhs and solution shall be generated) (0/1, default: 1)
R_NO_USERVARS	optional	omits generation of data.frame with aggregated columns according to user-defined resultgroups (0/1, default: 0)
R_NO_USERSET	optional	omits generation of a combined data.frame of aggregated columns according to user-defined resultgroups (0/1, default: 0)
R_FILTER	optional	list of variables to be used to create a subset of the combined data.frame of user variables

B.2 Model configuration

The model configuration controls the general setup of the MP-MAS model. It contains section headings and entries. Section headings are used to improve readability, but do not have any specific function. You are free to choose your own.

```
<configuration parameter>  = <scalar/list: MFL expression>
```

Entries in the configuration file can be assigned scalar values and lists and `mpmasql` function language expressions can be used to do so. These expressions may refer to variables in the [GLOBALS] section of the data file, not to [USER VARIABLES] . Local variables are not defined.

The following entries are defined:

Table B.2: Entries in the model configuration file

Field	Description	Required	Default
[TIME]			
STARTYEAR	Start year of simulation (integer)	required	-
ENDYEAR	End year of simulation (integer)	required	-
STARTMONTH	Start month of simulation (integer)	required	-
ENDMONTH	End month of simulation (integer)	required	-
STARTDAY	Start day of simulation (integer)	required	-
ENDDAY	End day of simulation (integer)	required	-
SEASONSTART	Start month of the cropping season (integer)	required	-

SEASONEND	Last month of cropping season (integer)	required	-
IRRIGMONTH	Number of irrigation months (integer)	optional	0
NORTHERN	Is the model location on the Northern Hemisphere (0-1)	optional	1
SPINUPS	Number of phasing in periods (integer)	optional	0
[LANDSCAPE]			
MAPS	directory with input maps (relative to input/)	optional	gis/
NSOIL	number of soil types(integer)	required	-
CELLSIZE	Size of gridcell in the map (number)	optional	1
[FINANCIAL]			
CURRENCYSCALE	if the currency is given in a different unit than 1, e.g. in 10,000 CHP. Influences the precision of calculations	optional	1
[AGENTS]			
POPULATIONS	Number of agent populations (integer)	optional	1
CLUSTERS	Number of clusters within each population (integer)	optional	1
AGENT_INFO_FROM_MAPS	Whether an explicit AGENT_INFO table is given, or whether the corresponding information should be read from the maps	optional	0
[ASSETS]			
ASSETDISTTYPE	Initial allocation of assets (0: random assignment by MP-MAS, 1: supplied by user)	optional	1
LOTTERYLOOPS	Number of lottery loops	optional	30
NFARMASSETS	Tentative number of farm assets (integer)	optional	10
SAFEROUND	Safe rounding factor for indivisible assets	optional	1
MINPERM	Minimum investment size, amounts smaller than this will not change the endowment of the agent	optional	.1
MINWATER	Minimum share of plant water demand that has to be fulfilled for assignment of irrigated perennials in the lottery (fraction)	optional	1
[CONSUMPTION]			
HHNUTRIENTS	List of household nutrients	required for advanced consumption model	-
[SUBMODELS]			
CONSTYPE	Type of consumption model (0: simple, 1: advanced)	optional	0
CROP_MODEL	Type of crop growth model (0: none, 1: TSPC, 2: CropWat, 3: External)	optional	0

BIOVERSION	Version of biophysical model (0: standard; 1: CropWat Chile; 2: CropWat with flood factor; 3: External yields from lookup table; 4: External with internal coupling in memory)	optional	0
EXTERNAL_CROP_MODEL	If and which link files for an external crop growth model are to be created by mpmasql (currently available: EXPERT-N).	optional	-1
XNDBGFILES	Whether debug yield and crop activity id files are to be created	optional	0
LANDSCAPE_MODEL	Type of landscape model (0: none, 1: EDIC, 2: ,3: External)	optional	0
IRRIGATION	Use irrigation (with landscape model 1, 1	optional	0
LANDMARKET_MODEL	Turn on land market model (0: No land transfers, 1: Land to "Other Farm Agent", 2: Rental market, 3: Long term market)	optional	0
PERMCROP_MODEL	Use permanent crops	optional	0
POLICY	Simple infrastructure subsidy policy model	optional	0
COUNTRY	Activate certain study specific modifications	optional	0
HARVEST_DECISION	Re-run production decision with updated yields and fixed land area after crop growth model results	optional	0
DEMOGRAPHY_MODEL	Type of demography model : 0 - basic demography model, 1- advanced demography model (currently, only for COUNTRY = 5)	optional	0
[DIFFUSION]			
NETWORKS	Number of Innovation Networks (integer)	required	-
COMTYPE	Type of communication	optional	2
CUMADOPT	Type of diffusion model	optional	1
THRESHOLDS	list of fractions that denote the lower bound of innovation segments	optional	[0]
[HYDROLOGY]			
NINFLOWS	Number of inflows	optional	0
VINFLOWS	Values supplied per inflow (set to 2 if EDIC is used)	optional	0
[CROP GROWTH			
XNSEASONS	External crop growth models: Number of cropping seasons per year	optional	1
[DYNAMICS]			
OFFLIQEQ	Switch off updating of liquidity and equity (0-1)	optional	0
OFFHHAGING	Switch off aging of households (0-1)	optional	0
OFFLIVEAGE	Switch off aging of livestock (0-1)	optional	0
OFFASSAGE	Switch off aging of assets (0-1)	optional	0

OFFINV	Switch off investments (0-1)	optional	0
OFFINVDELAY	Delay turning off investments by x periods	optional	-1
OFFDYNSOIL	Switch off soil dynamics (0-1)	optional	1
OFFDYNFLOW	Switch off dynamic inflows (0-1)	optional	1
[SOLVER]			
MINOBJ	Minimum objective value (MP-MAS issues a warning if an agent obtains an objective value lower than this	optional	0
OSL_MAXSOLVETIME	Maximum solving time	optional	5.5
OSL_MAXITERATE	Maximum number of iterations	optional	999 999 999
OSL_MAXNODES	Maximum number of nodes	optional	99 999 999
OSL_SEQUENCE	OSL sequencing (0-4): 0: start in mode 1; 1 or 2: start in mode 2; 3: start in mode 3; 4: start in mode 4	optional	0
OSL_M4_ATTEMPTS	number of attempts in mode 4	optional	2
OSL_M4_ITHRESH	Mode 4: max. number of integers for first attempt	optional	-1
OSL_M4_NTHRESH	Mode 4: max. number of nodes processed in all but last attempts	optional	-1
OSL_M4_STHRESH	Mode 4: max. number of solutions to be found in all but last attempts	optional	-1
OSL_M4-_LP_PRESOLVE	Mode 4: first attempt to use LP presolving	optional	1
OSL_M4_LP-_PRESOLVE_ONFAIL	Mode 4: use LP presolving if previous attempt failed	optional	1
OSL_M4_SCALE	Mode 4: first attempt to use scaling	optional	1
OSL_M4_SCALE-_ONFAIL	Mode 4: use scaling if previous attempt failed	optional	1
OSL_M4_CRASH	Mode 4: first attempt to use crashing	optional	1
OSL_M4_CRASH-_ONFAIL	Mode 4: use crashing if previous attempt failed	optional	1
OSL_M4-_CRASHMODE	Mode 4: crash mode (1-4); 1: dual feasibility may not be maintained; 2: dual feasibility is maintained, if possible, by not pivoting in variables that are in the objective function; 3: dual feasibility may not be maintained, but the sum of the infeasibilities will never increase; 4: dual feasibility is maintained, if possible, by not pivoting in variables that are in the objective function. In addition, the sum of the infeasibilities will never increase.	optional	1
OSL_M4-_I_PRESOLVE	Mode 4: first attempt to use LP presolving	optional	2
OSL_M4-_I_PRESOLVE-_ONFAIL	Mode 4: use MIP presolving if previous attempt failed	optional	1
OSL_M4-_I_STRATEGY	Mode 4: Strategy for MIP solving and presolving	optional	1
OSL_M4-_I_HEURPASS	Mode 4: number of heuristic passes in MIP solving/presolving	optional	1

OSL_M4_ACCEPT-_DIFF	Mode 4: accept attempt if absolute difference to best estimated solution is smaller than this	optional	0
OSL_M4_ACCEPT-_RELDIFF	Mode 4: accept attempt if relative difference to best estimated solution is smaller than this	optional	0
OSL_M4_CONT-_ONFAIL	Mode 4: continue after failed attempt (yes/no)	optional	1
OSL_M4_MODE3-_ONFAIL	Mode 4: continue in mode 3 if all attempts failed (yes/no)	optional	0
OSL_M4_I_PRE-_ADDROWS	Mode 4: max. number of rows to be added for integer presolving	optional	0
OSL_M4_LOG	Mode 4: log solving process	optional	0
[SHADOW PRICE CALCULATION]			
SHDW_SWITCH	Activate calculation of shadow prices for every agent (0-1)	optional	0
SHDW_LPMODE	Shadow prices for production or investment LP (production or investment)	optional	-
SHDW_FIRST	Identifier of first constraint for which shadow prices are to be calculated	optional	-
SHDW_NUM	Number of constraints for which shadow prices are to be calculated	optional	-

B.3 Model data file

The data file has six sections:

[GLOBALS] Here you can define global variables and lists, which you can use in `mpmasql` function language expressions in all of the model design files (except in other globals defined above a specific global).

[MPMAS PARAMETERS] In this section, parameters for the dynamics of the MP-MAS model and its submodules are set. You can use `mpmasql` function language to do so, referring only to global variables.

[MPMAS TABLES] In this section, you specify database queries for MP-MAS parameters or exogenous variables expected by `mpmasql` in a pre-defined table format. You can use `mpmasql` function language to do so, referring only to global variables.

[USER TABLES] In this section, you can define your own tables and the database queries to fill them, which you can then refer to in the definition of the model structure (.cll), the user variables and the instance lists. You can use `mpmasql` function language to define the database queries, referring only to global variables, configuration entries, and MP-MAS parameters.

[USER VARIABLES] In this section, you can define and set your own variables and lists, which you can then refer to in the definition of the model structure (.cll) and the instance lists. You can use `mpmasql` function language to set variable values, referring to global variables, configuration entries, MP-MAS parameters and user defined tables.

[INSTANCES] Here you provide the lists of class instances, to fill your model structure with data. You can use `mpmasql` function language to set variable values, referring to global variables, configuration entries, MP-MAS parameters, user defined tables and user defined variables.

B.3.1 [GLOBALS]

Globals are variables that you can refer to anywhere else in the model design files, i.e. in the model configuration, the model data file and the model structure. The only place where you cannot refer to them, is in global definitions that come before the one in which they are defined. Their main use is to implement scenarios that change MPMAS parameters or table queries.

The definition syntax is simple:

```
<globalname>   = <value or list: MFL expression>
```

and you refer to them later by using `#var()` for scalar variables, respectively `#foreach()`, `#list` or `#qlist` for list variables.

Examples:

```
FIRSTYEAR   =  2000
LASTYEAR =  2010
YEARS    =  #steps(#var(FIRSTYEAR), #var(LASTYEAR),1)
EQSH     =  .25
```

B.3.2 [MPMAS PARAMETERS]

In the MPMAS PARAMETERS section, you supply values for scalar parameters for the dynamics of the MP-MAS model and its submodules.

The definition syntax is similar to the definition of configuration entries or globals:

```
<parametername>   = <value or list: MFL expression>
```

MFL expressions can refer to global variables only. Examples:

```
LIQUIDITY   = 10000
EQSHARE  = #var(EQSH)
LEVERAGE = (1 - #var(EQSH) ) / #var(EQSH)
```

The following table lists all defined MPMAS parameters:

Table B.3: MPMAS parameters

Field	Description	Required	Default
ASSETMAXAGE	Maximum age of assets at simulation start (integer)	optional	7
ASSETMINAGE	Minimum age of assets at simulation start (integer)	optional	1
DURPOLICY	Policy model: Duration of policy measure (integer)	optional	0
EFFRAIN _TYPE	CropWat Model: Effective rainfall (USDA: USDA formula, Regression: regression approximation)	optional	USDA
EFFRAIN_CONST	CropWat Model: Effective rainfall, Constant for Regression	optional	1.3E+00
EFFRAIN_CWR	CropWat Model: Effective rainfall, Coefficent of CWR for Regression	optional	-3.5E-02

EFFRAIN_CWR2	CropWat Model: Effective rainfall, Coefficent of CWR squared for Regression	optional	1.7E-04
EFFRAIN_PREC	CropWat Model: Effective rainfall, Coefficent of PREC for Regression	optional	6.4E-01
EFFRAIN_PREC2	CropWat Model: Effective rainfall, Coefficent of PREC squared for Regression	optional	-4.8E-04
EFFRAIN_CWRPREC	CropWat Model: Effective rainfall, Coefficient of the interaction term of PREC and CWR for Regression	optional	1.0E-03
EQSHARE	Minimum share of equity financing for investments (default value, can be overridden by segment specific values in SEGINFO and by asset-specific values in the model structure file)	optional	1
EXPECTATIONS	Type of expectations (0:constant, 1:naive, 2:adaptive, 3: foresight), (default value, can be overridden by agent specific values)	optional	0
INCTRANS	Income transfer (migration submodel) (default value, can be overridden by segment specific values in SEGINFO)	optional	0
INTEREST_LGCRD	Interest rate on long-term credits (default value, can be overridden by segment specific values in SEGINFO)	required	-
INTEREST_SHCRD	Interest rate on short-term credits (default value, can be overridden by segment specific values in SEGINFO)	required	-
INTEREST_SHDEP	Interest rate on short-term deposits (default value, can be overridden by segment specific values in SEGINFO)	required	-
INVESTMENT _HORIZON_AFFECT _LIFETIME	Advanced demography with investment horizon: If set to one, this constraint makes the agent use his expected remaining household lifetime to calculate investment costs of new assets, whenever it is smaller than the lifetime of the asset. (Note: currently not yet implemented for perennial crops and livestock.)E.g if a an asset has a lifetime of 8 years, but the lifetime is expected to remain in farming only 5 years more, the annuity would be calculated based on 5 instead of 8 years.	optional	0

INVESTMENT _HORIZON_MODEL	Advanced demography model: Turn on investment horizon model If set to one, this allows you to make investment behavior depend on the expected remaining lifetime of the household, using the corresponding parameters	optional	0
INVESTMENT _HORIZON _PTHRESHOLD	Advanced demography with investment horizon: MPMAS will go through the retirement probabilities of each relevant household member (who these are is determined by the INVESTMENT _HORIZON _WHO setting). The age where the retirement probability first surpasses the value given here is identified as the probable end of household lifetime. Expected remaining household lifetime if this member is then calculated by subtracting the current age of the household member from this age. The actual expected remaining household lifetime is the maximum one found among all relevant household members	optional	2
INVESTMENT _HORIZON_WHO	Advanced demography with investment horizon: This indicates who is considered relevant for the remaining household lifetime: 0 - only the current household head; 1 - the current household head and the household member who would become household head if the current household head retired today; 2 - as in 1, plus a potential future household head, which is not old enough to become household head today, but will become available as household head before the current one needs to retire.	optional	0
LEVERAGE	Leverage at the start of the simulation (1 - EQSHARE) / EQSHARE (default value, can be overridden by agent specific values)	optional	0
LIQRESERVE	Liquidity Reserve: percentage of expected end-of-year cash demand that is not entered into the MILP (fraction)	optional	0
LIQUIDITY	Liquidity at the start of the simulation (default value, can be overridden by agent specific values)	optional	0
LMMINBIDREL	Land market: Minimum bid (relative to average shadow prices)	0.2	
LMMINBIDABS	Land market: Minimum bid (absolute value)	20	

LMMAXBIDREL	Land market: Maximum bid (relative to average shadow prices)	2	
LMMAXBIDABS	Land market: Maximum bid (absolute value)	2000	
LMMKUPOUT	Land market: Markup for renting-out decisions	1.3	
LMMKUPIN	Land market: Markup factor for renting-in decisions	0.7	
LMTRYOUT	Land market: Number of renting-out attempts	3	
LMTRYIN	Land market: Number of renting-in attempts	6	
LMMAXDIST	Land market: Maximum distance for placing bid in grid cells)	30	
LMDURATION	Land market: Duration of rental contract (years)	1	
LMSHPAYEND	Land market: Share of rental payment paid after harvest	0.0	
MAXAGEPERM	Policy model: Maximum age of subsidized permanent crops (integer)	optional	0
MIGRATIONPULL	Migration pull factor (default value, can be overridden by segment specific values in SEGINFO)	optional	0
OPPWAGE	Opportunity wage of the household head (default value, can be overridden by segment specific values in SEGINFO)	optional	0
OVERLAP	Overlap of network thresholds	optional	1
PAREX	Parameter for adaptive expectations, see Brandes et al. (1997: 382)	optional	0.5
RESELL	Share of acquisition cost that can be recovered when reselling the investment (default value, can be overridden by segment specific values in SEGINFO)	optional	0
SCONEXTRA	Simple consumption model: Extra consumption percentage if income surpasses minimum consumption (fraction)	required	-
SCONMIN	Simple consumption model: Minimum consumption: money that is consumed at least if the liquid means suffice (number)	required	-
SCONRED	Simple consumption model: Essential share of minimum consumption that has to be maintained even if liquidity does not suffice (fraction)	required	-

SUCCESSION_AD _MINCONS_FACTOR	Advanced demography model: share of household minimum consumption to be covered for succession at death	optional	-9999
SUCCESSION_AD _MINIMUM_PROFIT	Advanced demography model: minimum income to be reached for succession at death	optional	-999999999
SUCCESSION_AD _DECISION_ACTIVITY	Advanced demography model: index of a MILP activity, whose solution value needs to be larger than zero, for succession at death to occur	optional	-
SUCCESSION_VR _MINCONS_FACTOR	Advanced demography model: share of household minimum consumption to be covered for voluntary retirement and succession	optional	-9999
SUCCESSION_VR _MINIMUM_PROFIT	Advanced demography model: minimum income to be reached for voluntary retirement with succession	optional	-999999999
SUCCESSION_VR _DECISION_ACTIVITY	Advanced demography model: index of a MILP activity, whose solution value needs to be larger than zero, for voluntary retirement and succession to occur	optional	-
SUCCESSION_VR _MINIMUM_RENTAL	not used yet	optional	0
SUCCESSION_OR _MINCONS_FACTOR	Advanced demography model: share of household minimum consumption to be covered for succession after obligatory retirement	optional	-9999
SUCCESSION_OR _MINIMUM_PROFIT	Advanced demography model: minimum income to be reached for succession after obligatory retirement	optional	-999999999
SUCCESSION_OR _DECISION_ACTIVITY	Advanced demography model: index of a MILP activity, whose solution value needs to be larger than zero, for succession after obligatory retirement to occur	optional	-
TCOST	Transport cost for one gridcell (number) (for land markets)	optional	0
XC_SAV_INCSEGS	Advanced consumption model: List with the width of the income segments (except first) of the savings function	required for advanced consumption	-
XC_SAV_COEF_INC	Advanced consumption model: Estimated coefficient of income in the savings function	required for advanced consumption	-
XC_SAV_COEF_INC2	Advanced consumption model: Estimated coefficient of squared income in the savings function	required for advanced consumption	-
XC_SAV_COEF_HH	Advanced consumption model: Estimated coefficient of household in the savings function	required for advanced consumption	-

XC_SAV_COEF_CONST	Advanced consumption model: Estimated constant in the savings function	required for advanced consumption	-
XC_FNF_COEF_CONST	Advanced consumption model: Estimated constant in the food-non-food expenditure function	required for advanced consumption	-
XC_FNF_COEF_EXP	Advanced consumption model: Estimated coefficient of total expenditure in the food/non-food expenditure function	required for advanced consumption	-
XC_FNF_COEF_HH	Advanced consumption model: Estimated coefficient of total expenditure in the food/non-food expenditure function	required for advanced consumption	-
XC_FNF_EXPSEGS	Advanced consumption model: List with the width of the total expenditure segments in the food/non-food expenditure function	required for advanced consumption	-
XC_FOO_FEXSEGS	Advanced consumption model: List with the width of the food expenditure segments in the food category expenditure function	required for advanced consumption	-

B.3.3 [MPMAS TABLES]

In the MPMAS TABLES section, you specify the data sources for all tables that have a fixed format required by `mpmasql`. The syntax for MPMAS table definitions is

```
tablename   = (tabletype) {   tablesource: MFL expression that can
                      span several lines
                        }
```

The table type can be either sql or ascii. For tables of types sql, the expression in the curly brackets should result in an SQL SELECT statement that is being send to the database specified in the control file and results in a table of the required format. E.g.

```
ASSET_ENDOWMENTS  = (sql) {   "SELECT farmstead_id, asset, quantity
                       FROM example_db.tbl_assets
                       WHERE scenario = " ~ #var(ASSETSCEN)
                        }
```

For tables of type ascii, the expression in the curly brackets should result in a filename of a text file with values in tab-separated format (experimental). E.g.

```
ASSET_ENDOWMENTS  = (ascii) { "data/assets.txt"  }
```

List of MPMAS data tables

Table B.4: MPMAS data tables

Field	Description	Required	Default

AGENT_INFO	Table that contains agent info like internal MP-MAS agent_id, cluster, population etc. from each agent	required if ASSET-DISTTYPE = 1 and not AGENTINFO_FROM_MAPS = 1	-
ASSET_CDF	Table that lists cumulative distribution functions for assets for each cluster	required if ASSETDIST-TYPE = 0	-
ASSET_ ENDOWMENTS	Table that lists asset endowments for each household	required if ASSETDIST-TYPE = 1	-
CROPWAT_PRECIPEXPECT	CropWat Model: Table with initially expected monthly precipitation for each sector	required	-
CROPWAT_PRECIPACTUAL	CropWat Model: Table with actual monthly precipitation for each sector and simulation year	required	-
CROPWAT_EVAPOEXPECT	CropWat Model: Table with initially expected potential reference evapotranspiration for each sector	required	-
CROPWAT_EVAPOACTUAL	CropWat Model: Table with actual monthly potential reference evapotranspiration for each sector and simulation year	required	-
EDICROUTING	EDIC Model: Table with surface and subsurface routing coefficients between sectors	required	-
EDICINFLOWEXPECT	EDIC Model: Table with expected monthly river flows	required	-
EDICINFLOWACTUAL	EDIC Model: Table with actual monthly river flows for every simulation year	required	-
EDICAGENTRIGHTS	EDIC Model: Table with water rights of agents as a share of sector share of inflow	required	-
EDICSECTORRIGHTS	EDIC Model: Table with sector shares of rights to each inflows	required	-
EDICIRRIGMETHODS	EDIC Model: Table with efficiency characteristics for each irrigation method	required	-
FIXCOSTS	Table of fix costs the agent have to face (additional to debt service for assets and depreciation), cluster specific	optional	0

HOUSEHOLD_CDF	Table that specifies the household composition in terms of sex and age categories and population as cumulative distribution function for each cluster	required if ASSETDIST-TYPE = 0	-
HOUSEHOLD_COMPOSITION	Table that specifies the household composition in terms of household member categories and population for each household	required if ASSETDIST-TYPE = 1	-
HOUSEHOLD_DYNAMICS	Table specifying characteristics of population members at different ages (career)	required	-
HOUSEHOLD_MEMBER_TYPES	Table that defines the household member categories used in HOUSEHOLD_COMPOSITION or HOUSEHOLD_CDF (table)	required	-
HOUSEHOLD_SWITCHING	Advanced demography model: Table indicating the probability that the partner/descendant of a person of career X gets career Y, respectively the probability of a household member of career X to switch to career Y if certain events occur	optional	-
INFPROJ	Policy model: Table describing the characteristics of infrastructure projects	optional	0
INFPROJPARTICIPANTS	Policy model: Table listing farmstead_ids of participants of infrastructure projects	optional	-
INNOVATIONS	Table specifying the availability of innovations to innovation segments	optional	-
INVESTMENT_HORIZON	Table to override any of the global INVESTMENT_HORIZON_...MPMAS parameters with population specific settings	optionl	-

INVESTMENT_HORIZON-_CONSTRAINTS	Advanced demography model with investment horizon: Table listing constraints which allow to directly influence the LP decisions of the household depending on expected household lifetime. You can define as many constraints as you like, by indicating the corresponding LP row, a minimum age and a right hand side value to insert if the household falls into the category. This right hand side value is then inserted into the constraint representing the current remaining expected lifetime of the household. The respective constraint is the one, whose minimum lifetime is surpassed, while the minimum lifetime of the next following constraint is not surpassed.	optional	-
PRODUCER_ORGANIZATIONS	List producer organizations and links them to the files containing their decision problem	optional	-
PRODUCER_ORGANIZATION-_SERVICES	describes the links between farm agent activities and decision problem of producer organizations	optional	-
REGION	a table describing the sectors of the model	required	-
SEEDS	Table that lists a seed for each sector to initialize the random distribution of assets and household members (table)	required	-
SEGBOUNDS	Table containing segment-specific restrictions on perennial investments to avoid over investment and bankruptcy due to gestation periods	optional	-
SEGINFO	Table providing segment-specific financial variables	optional	-
SUCCESSION_CONTROL	Table to override any of the global SUCCESSION_CONTROL_... MPMAS parameters with population specific settings	optional	-

Description of table structures

Table B.5: Structure of the AGENT_ INFO table

Column	Type	Description
farmstead_id	integer	GIS Id of the farm (key)
pop	integer	Population ID
clu	integer	Cluster ID
catchment	integer	Catchment
agent_id	integer	MP-MAS internal agent index
sec	integer	MP-MAS internal sector id (not gis_id)
numplots	integer	number of plots

Table B.6: Structure of the ASSET_CDF table

Column	Type	Description
pop	integer	Population ID (key)
clu	integer	Cluster ID (key)
asset	identifier	Identifier of networkobject (key)
upbound	number (0-100)	Upper bound of CDF segment (key)
value	number	number of members to assign for this CDF segment

Table B.7: Structure of the ASSET_ ENDOWMENTS table

Column	Type	Description
farmstead_id	integer	GIS Id of the farm (key)
asset	identifier	Identifier of networkobject (key)
value	number	endowment

Table B.8: Structure of CropWat tables

Column	Type	Description
	CROPWAT_PRECIPEXPECT	
gisid	identifier	Gis ID of sector
month	integer	calendar month (1..12)
precipitation	number	volume
	CROPWAT_PRECIPACTUAL	
gisid	identifier	Gis ID of sector
year	integer	year of flow
month	integer	calendar month (1..12)
precipitation	number	volume
	CROPWAT_EVAPOEXPECT	
gisid	identifier	Gis ID of sector
month	integer	calendar month (1..12)
evapotranspiration	number	volume

	CROPWAT_EVAPOACTUAL	
gisid	identifier	Gis ID of sector
year	integer	year of flow
month	integer	calendar month (1..12)
evapotranspiration	number	volume

Table B.9: Structure of EDIC tables

Column	Type	Description
	EDICROUTING	
from_gisid	integer	gisid of sector where flow originates
to_gisid	integer	gisid of sector where flow arrives
surface_coeff	share	surface flow coefficient
subsurface_coeff	subsurface flow coefficient	
	EDICINFLOWEXPECT	
inflow_id	identifier	ID of inflow
month	integer	calendar month (1..12)
flow	number	volume of flow
	EDICINFLOWACTUAL	
inflow_id	identifier	ID of inflow
year	integer	year of flow
month	integer	calendar month (1..12)
flow	number	volume of flow
	EDICPRECIPEXPECT	
gisid	identifier	Gis ID of sector
month	integer	calendar month (1..12)
precipitation	number	volume
	EDICPRECIPACTUAL	
gisid	identifier	Gis ID of sector
year	integer	year of flow
month	integer	calendar month (1..12)
precipitation	number	volume
	EDICAGENTRIGHTS	
catchment_	integer	Catchment ID (=0)
gisid	identifier	Gis ID of sector
farmstead_id	integer	Farmstead Id as used in maps
inflow_id	identifier	ID of Inflow
share_in_sector_flows	share	Agent's share of sector's share of inflow
	EDICSECTORRIGHTS	
gisid	identifier	Gis ID of sector
inflow_id	identifier	ID of Inflow
water_right_share	share	Sector share to inflow
max_flow_capacity	number	Maximum flow capacity of canals in sector with respect to this inflow
	EDICIRRIGMETHODS	
irrig_id	integer	ID of irrigation technology
irrig_txt	string	Name of irrigation technology
irrig_code	string	MP-MAS Code of Irrigation technology
night_sh	share	Loss through night time runoff
evapo_sh	share	evapotranspiration share
surface_sh	share	surface runoff share
subsurface_sh	share	subsurface runoff share

ground_sh	share	groundwater loss share

Table B.10: Structure of the FIXCOSTS table

Column	Type	Description
pop	integer	Population ID (key)
clu	integer	Cluster ID (key)
upbound	number	Upper bound of farm size segment (hectare) to which the following fix costs apply(key)
fix	number	Fixed cost that does not depend on farm size within the size class (in monetary units)
var	number	Additional fixed costs that depends on farm size (in monetary units per ha)

Table B.11: Structure of the HOUSEHOLD_CDF table

Column	Type	Description
pop	integer	Population ID (key)
clu	integer	Cluster ID (key)
kindsexageid	identifier	Sex-Age-Category
upbound	number (0-100)	Upper bound of CDF segment
value	integer	number of members to assign for this CDF segment

Table B.12: Structure of the HOUSEHOLD_ COMPOSITION table

Column	Type	Description
farmstead_id	integer	GIS Id of the farm (key)
sexagecat	identifier	Identifier of the sex-age-category (key)
value	number	number of members

Table B.13: Structure of the HOUSEHOLD_MEMBER_TYPES table

Column	Type	Description
For the basic demography model:		
code	identifier	ID of household member category (key)
career	identifier	ID of the career of this category
sex	0-1	0 - female, 1- male
lowage	integer	lower age bound
upage	integer	upper age bound
For the advanced demography model:		
code	identifier	ID of household member category (key)
career	identifier	ID of the career of this category
probably_married	0-1	if the household member is married at the beginning of the simulation (only if suitable partner available)
lowage	integer	lower age bound

upage	integer	upper age bound

Table B.14: Structure of the HOUSEHOLD_DYNAMICS table

Column	Type	Description
pop	integer	Population (key)
career	identifier	The Career Identifier (key)
upperyear	integer	The upper bound of the age segment the information is provided for.(key)
labgrp	identifier	Labour category the agent provides labour to in this age segment
labprov	number	Amount of labour per year the agent provides within this age segment
mortality	fraction	Probability of dying for every year within the age segment
fertility	fraction	Probability of giving birth for every year within the age segment
Additional columns for the advanced demography model:		
p_marriage	fraction	probability of finding a partner in case member has not got one yet
p_leaving	fraction	probability of leaving the household (with partner)
p_retiring	number	values between 0 and 1 indicate probability voluntary attempt to retire, values above one indicate sure (obligatory) retirement (Retirement for normal household members always succeeds, retirement of household head requires conditions to be met)
priority_hhead	number	priority in selection procedure for new household head: The lower the value, the higher the priority. Negative values indicate person cannot become household head, values above 5000 indicate person can only become household head when old one dies.
mincons	number	monetary minimum consumption of household member
Additional columns for the advanced consumption model:		
energy	number	Yearly energy need within this age segment (only)
nutrient2, ...	number	Further nutrient needs within this age segment (optional and only for advanced consumption models)

Table B.15: Structure of the HOUSEHOLD_SWITCHING table

Column	Type	Description
pop	integer	Population (key)
orig_career	identifier	The Career Identifier (key)

reason	integer	Reason for switching: 0 - Career of descendants, 1 - Career to switch to after marriage, 2 - Career of partner, 3 - Career to switch to when becoming household head, 4 - Career to switch to when partner becomes household head, 5 - Career to switch to when retiring
new_career	identifier	The Career Identifier (key)
probability	fraction	the probability to switch to/spawn the new career in case the reason applies

Table B.16: Structure of the INFPROJ and INFPROJPARTICIPANTS tables

Column	Type	Description
		INFPROJ
project_id	identifier	Project Identifier
year	integer	Year of Project Implementation
object	idientifier	Network object that is supplied by Project
size	number	Size of network object provided (e.g. ha)
total_cost	number	total cost of the projects over all participants
subsidized_share	share	Share that has been subsidized
		INFPROJPARTICIPANTS
project_id	identifier	Project Identifier
farmstead_id	integer	Farmstead id of participant
catchment_id	integer	Catchment Id of Farmstead
sector_id	integer	Sector Id of Farmstead

Table B.17: Structure of the INNOVATIONS table

Column	Type	Description
network	integer	network index (counting from 0) (key)
segment	integer	segment index (counting from 0) (key)
innogrp	identifier	innovation group identifier (key)
available	integer	model period the networkobjects of this innovation group become available to the agents of this segment (exact availability depends on COMTYPE !)
accessible	0-1	whether the networkobjects of this innovation group are accessible to the agents of this segment at simulation start or not

Table B.18: Structure of the INVESTMENT_HORIZON table

Column	Type	Description
pop	integer	Population (key)

parameter	parameter setting	any of the INVESTMENT_HORIZON_...parameters
setting	number	population specific value

Table B.19: Structure of the INVESTMENT_HORIZON_CONSTRAINTS table

Column	Type	Description
pop	integer	Population (key)
minlife	integer	lower bound of remaining lifetime segment
constraint	identifier	identifier of corresponding constraint
rhsvalue	number	value to be entered on the right hand side of the constraint

Table B.20: Structure of the PRODUCER_ORGANIZATIONS table

Column	Type	Description
PO	integer	Running PO number (key)
Description	text	Name of PO
File	path	Path to .cll file containing PO decision problem (relative to working directory)

Table B.21: Structure of the PRODUCER_ORGANIZATION_SERVICES table

Column	Type	Description
PO	integer	Running PO number (key)
POservice	identifier	Identifier of PO service (key)
member_activity	identifier	identifier of quantity activity in farm agent MIP
PO_qconstraint	identifier	identifier of quantity constraint in PO MIP
PO_qactivity	identifier	identifier of quantity activity in PO MIP
PO_profit	identifier	identifier of profit activity in PO MIP

Table B.22: Structure of the REGION table

Column	Type	Description
gisid	integer	ID of the village/sector in the GIS map (overall ID for the region)
code	string	Name for the Region
catchid	integer	Id of the Catchment/District
subsector	0-1	Is this a subsector ?
beingused	0-1	Is this sector to be used?
rank	integer	Rank of the sector
Additional columns if the EDIC hydrology model is used:		
reuse	number	Initially Expected Reuse Factor
canalefficieny	share	Canal Efficiency

watersystemefficiency	number	Water System Efficiency due to Overnight Storage
bcoeff	share	b calibration coefficient of the EDIC model
gcoeff	share	g calibration coefficient of the EDIC model
num_random_wr	integer	Number of random water rights to be allocated (0)

Table B.23: Structure of the SEEDS table

Column	Type	Description
gisid	integer	identifier of the sector (key)
seed	integer	random seed

Table B.24: Structure of the SEGBOUNDS table

Column	Type	Description
network	integer	network index (counting from 0)
segment	integer	segment index (counting from 0)
soil	integer	soil index (counting from zero)
physbound	number	physical bound for investments on that soil type
oppcost	number	estimated opportunity cost for investments on that soil type

Table B.25: Structure of the SEGINFO table

Column	Type	Description
network	integer	network index (counting from 0)
segment	integer	segment index (counting from 0)
variable	name of MP-MAS parameter to set	
value	segment specific setting	

Table B.26: Structure of the STANDARD_LAND_RENTS table

Column	Type	Description
network	integer	network index (counting from 0)
segment	integer	segment index (counting from 0)
soil	integer	soil index (counting from zero)
rent	number	price for renting out/selling of land if agent has shortage of liquidity when land market and perennial model are not used

Table B.27: Structure of the SUCCESSION_CONTROL table

Column	Type	Description

pop	integer	Population (key)
parameter	parameter setting	any of the SUCCESSION_CONTROL_... parameters
setting	number	population specific value

B.3.4 [USER TABLES]

In the USER TABLES section you can define tables for data you want to refer to in the part of the model structure, which you define yourself. As the table format is not predefined, you have to specify the number of columns, `mpmasql` should treat as key columns. The syntax for user table definitions is

```
tablename   = (tabletype, number of key columns) { tablesource: MFL
    expression that can
                     span several lines
                       }
```

The table type can be either sql or ascii. For tables of types sql, the expression in the curly brackets should result in an SQL SELECT statement that is being send to the database specified in the control file and results in a table of the required format. E.g.

```
yields   = (sql, 3) {   "SELECT crop_id, soil_id, intensity, quantity
                      FROM example_db.tbl_yields
                      WHERE weather = " ~ #var(WEATHERSCEN)
                       }
```

For tables of type ascii, the expression in the curly brackets should result in a filename of a text file with values in tab-separated format (experimental). E.g.

```
yields   = (ascii, 3) { "data/dry_year_yields.txt"  }
```

B.3.5 [USER VARIABLES]

In the USER VARIABLES section, you define your own variables and lists, which you can then use the model strcuture file and the INSTANCES section of the model data file. The definition syntax is similar to the definition of configuration and MPMAS parameters and global variables. MFL expressions can refer to global variables, configuration parameters, MPMAS parameters and user-defined tables.

```
<variablename> = <value or list: MFL expression>
```

Examples:

```
WEATHER = "dry"
MONTHS   = #steps(1,12,1)
PRODUCTS = #table(products,0, [ ])
```

B.3.6 [INSTANCES]

The INSTANCES section contains lists corresponding to classes in the model structure file. Each class of the model structure file is a blueprint describing all the model elements (like activities, assets, constraints, ...) that are to be created for every item in the corresponding lists. The list members have to correspond to the rules for identifiers, i.e. they can contain only letters of the English alphabet, digits and underscores. You can use MFL functions to specify these lists referring to all global and user variables, configuration and MPMAS parameters and user defined tables. The syntax is simply:

```
<classname> = <list: MFL expression>
```

Examples:

```
PRODUCTS = [wheat, barley, maize]
MACHINERY = #sql("SELECT machinery_id FROM tbl_machinery WHERE
    available_in_study_area = TRUE")
PERENNIALS = #table(perennials, 0, [])
```

B.4 Model structure file

The model structure file is subdivided into four sections. These section headings are syntactically meaningful, i.e. you cannot omit them, however you can rearrange their order if you like.

[ACTIVITY TYPES] Here you can define additional types of LP activities, apart from the pre-defined ones. Activity types are mainly used to sort the activities in the LP matrix, and to refer to groups of activities in `mqlmatrix`.

[CONSTRAINT TYPES] Here you can define additional types of LP constraints, apart from the pre-defined ones. Constraint types are mainly used to sort the constraints in the LP matrix, and to refer to groups of constraints in `mqlmatrix`.

[DEFAULTS] Here you can change default settings for all elements.

[MODEL] This is the main part, where you define your decision model and the links to submodels.

B.4.1 [ACTIVITY TYPES] and [CONSTRAINT TYPES]

These two sections look the pretty much the same. They just contain a comma-separated list of the additional types you want to define, in the order you want them to appear in the LP tableau. The lists can stretch over several lines.

Example:

```
[ACTIVITY TYPES]
grow, hire, default,
invest

[CONSTRAINT TYPES]
balance, default,
endowment
```

For matrix constraints and activities, types fulfill a double role. First, there are predefined types which denote special elements that have to be indicated to MP-MAS in the input file, e.g. the liquidity constraint, short term credit activity, household labor endowment, soil endowment and so on (see Section B.4.1) . These types are known to MP-MAS and should not be included into the ACTIVITY TYPES and CONSTRAINT TYPES section, as there position in the input file is fixed an should not be changed.

Second, these types are used to sort the matrix elements according to the sorting order specified by the user in the variables file. The user is free or better is advised to define additional types for those matrix elements which do not have to be marked as special elements in order to have control over the positioning of the elements in the matrix.

Activities and constraints are sorted first according to the type order you specify in the first two section inf the structure file and then alphabetically by activity identifiers (numbers receive special treating in alphabetical treatment, making sure that e.g. labor_in _month_2 is located before labor_in _month_12.

Special activity and constraint types

Table B.28: Special matrix element types

Activities

MASsell	Selling activity: need a Market.dat entry and are placed first in the matrix
MASbuy	Buying activity: also need a Market.dat entry and are placed after selling activities
MASfoodbuy	Advanced consumption model: Buying food, not considered in income, but in cash flow calculation
MASfoodconsume	Advanced consumption model: Consuming self-produced food, not considered in cash flow, but in income calculation
MASlabpin	a hiring permanent labour activity
MASlabpout	a selling permanent labour activity
MASlabtin	a hiring temporary labour activity
MASlabtout	a selling temporary labour activity
MASsav	savings activity
MASdep	short-term depositing activity
MASscred	short-term credit activity
MASfuture	future selling activity: have to go last and not counted for cash flow calculation, but for income
MASnutpenalty	Advanced consumption model: disutility for starving, not considered in income nor cash flow
MAStppenalty	Time preference penalty, not considered in income nor cash flow
MASpenalty	User defined disutility, not considered in income nor cash flow
MASzero	an activity not counted in cash income calculation, go even after future selling activities

Constraints

MASliq	Liquidity Endowment Constraint
MASlab	Household labour endowment
MASsoil	Soil Endowment
MAScas	Cash Financing Constraint
MASinv	Soil Constraint for Perennial Investments
MASly1	Liquidity in first year
MASong	Ongoing Liquidity
MASscl	Short-Term Credit Limit
MASzero	'Zero' Constraint: go last, static RHS value for all agents

B.4.2 [DEFAULTS]

Here you can change default settings for elements of all classes. E.g. if you would like to make all activities integer activities by default and set the default priority to 100, you're DEFAULTS section would look like this:

```
[DEFAULTS]
#activity activity {
    integ   #= 1
    priority   #= 100
}
```

If you do not want to specify any customary default settings, you can omit the whole section.

B.4.3 [MODEL]

The main model structure section contains class definitions, which themselves contain model element definitions, which then again contain field (or attribute) definitions and subelement field definitions.

```
#class <class1name> {
     #<elementtype> <element identifier>  {

          <field name: MFL expression>  #= <value: MFL expression>

          #<subelement>> <field name: MFL expression>  #= <value: MFL
              expression>
     }
     ...
}
#class <class2name> {
      ...
}
...
```

Model element types

There are seven types of model elements that can be included into a class (see Table B.29). Each of these have a number of predefined fields (attributes) and subelement fields, whose values must, respectively can be set in the model structure file. For LP activities, every other field is interpreted as the coefficient of this activity in the constraint with the identifier that corresponds to the attribute name. In all other elements unknown fields are ignored.

Table B.29 provides an overview of available types of model elements and the next subsections will describe the defined attributes for each type of model element.

Table B.29: Types of model elements

Type	Description
activity	An activity in the decision model
constraint	An constraint in the decision model
asset	A farm asset/resource that provide capacities to the agent, can be invested into, may be associated to a debt payment, ages and wears

perennial	A complex element that contains activities, constraints and an asset linked to a perennial crop type
livestock	A complex element that contains activities, constraints and assets linked to a livestock type
sos	A special ordered set, a set that groups some binary integer activities of which only one can be chosen
timepreferencepenalty	

Activity attributes

Table B.30: Attribute fields for activities

Field	Description	Values	Default
name	Name of the activity (for commented input files)	any string	"N/A"
unit	Unit of the activity (for commented input files)	any string	"N/A"
type	compare Section B.4.1	type identifier	default
account	Account for partitioning of total gross-margin	account identifier	not_specified
lbound	Lower Bound	any number	0
ubound	Upper Bound	any number	10^{31}
integ	Integer activity	0-1	0
priority	Branching priority for integer activities	0-1000	1000
pseudo_down	downside pseudo costs for integers	number	0.01
pseudo_up	upeside pseudo costs for integers	number	0.01
sos	SOS the activity belongs to	sos identifier	-1
sos_position	position in SOS	integer	0
orow	objective row coefficient for Initital Year and Single Matrix Mode	any number	0
orow_<modelyear>	actual objective row coefficient for year <modelyear>	any number	0
consfix	Fixed in end-of-year decisions ?	0-1	0
landfix	Fixed in land market decisions ?	0-1	0
cptype	Type of changing production costs (0: normal, 1: perennial, 2: no updating of liquidity and credit coefficients)	integer	0
account	When using the R output script, this field can be used to for a detailed decomposition of revenue and cost into different accounts	"not_specified"	

type_of_payment	Determines how the objective row coefficient of the activity is treated in the MPMAS income and cash flow calculation: 1 - cash earning (income + cash flow), 2 - earning in-kind (income), 3 - purchase of food (cash flow), 4 - consumption of self-produced food (income), 5 - production cost (income + cash flow), 6 - appreciation of assets (income), 7 - utilities (neither nor). The following activity types have predetermined values: MASsell (1), MASsellkind (2), MASfoodbuy (3), MASfoodconsume (4), MASfuture (6), MASzero (7), MASnutpenalty (7), MAStppenalty (7), MASpenalty (7). The default value for all other activities is 5 and can be overriden here	integer	5

Constraint attributes

Table B.31: Attribute fields for constraints

Field	Description	Values	Default
name	Name of the constraint (for commented input files)	any string	"N/A"
unit	Unit of the constraint (for commented input files)	any string	"N/A"
eqtype	Equation operator (default is 1 ($\leq$)	digit (1: $\leq$; 2: $\geq$; 3: $=$)	1
epsilon	Precision for equality constraints	number	0
range	Lower bound for LE and EE, upper bound for GE constraints (only very seldom required to be set)	number	$\leq$: -10^{31}; $=$: 0; $\geq$: 10^{31}
type	compare Section B.4.1	type identifier	
value	for `MASzero` constraints (cf. Sect.B.4.1) only: the value to be set	any number (default: 0)	
labgrp	for `MASlab` constraints (cf. Sect.B.4.1) only: the labour group this labour constraint belongs to	labour group identifier	

Asset attributes and subelements

Table B.32: Attribute fields for assets

Field	Description	Values
name	Name of the asset (for commented input files)	any string
innogrp	Innovation group the asset belongs to (cf. Sect. 4.12)	innovation group identifier, optional
type	Asset type (0: Asset with land demand, 1 : Asset without land demand, 2: Symbolic asset, > 2: Asset types defined by special country versions)	integer

div	Asset is divisible	0-1
acqcost	Acquisition cost of asset	any number,
lifetime	life time of the asset	any positive integer
terrain	soil type the asset requires (perennials)	integer
mininvest	Minimum investment size	0
act	The related investment activity	activity identifier
con	The related endowment constraint	constraint identifier
perm	the related yield constraint (for perennials)	constraint identifier or -1
multiplier	value passed into the rhs endowment, when agent invests into one unit of the asset	any number
innovativeness	Innovativeness this object indicates	default
eqshare	Share of the acquisition cost that has to be financed from own capital	fraction between 0 and 1
irate	Interest rate on foreign capital used to buy the asset	fraction between 0 and 1
available_until_period	last period the object is available for investment (currently only for COUNTRY == 5)	integer
oname	Shortname for the asset (10 characters) passed to MP-MAS	10 Character-String
landreq	Does the object require land - for lottery	0-1
inflation_rate_past	Country switch Germany (5): Assumed average price increase in the past (i). Acquisition cost of assets in agents' inventory at simulation start are divided by $(1+i)^{age}$, to account for the fact that purchases prices were lower in the past	0
minimum_age_of_asset_at_simulation_start	Country switch Germany (5): Minimum age an agent's asset of this type can have at the beginning of the simulation	1
maximum_age_of_asset_at_simulation_start	Country switch Germany (5): Maximum age an agent's asset of this type can have at the beginning of the simulation	1000

Perennial attributes and subelements

Livestock attributes and subelements

SOS attributes and subelements

Table B.33: Attribute fields for special ordered sets

Field	Description	Values	Defaults
name	Name of the SOS (for commented input files)	any string	"N/A"

type	Type of sos (1: at maximum one of the activities in the set can be non-zero, 2: at maximum two of the activities in the set can be nonzero, and these have to be neighbors, 3: exactly one of the activities in the set has to be one, all others zero)	integer	3
priority	branch and bound priority of the set	0-1000	1000

Time-preference penalty attributes and subelements

Table B.34: Example: Results for #this(instance) and #this(by,<i>) in the respective element

activitiy id	instance	by,0	by,1	by,2
wheat_grow_0_high	wheat	0_high	0	high
wheat_grow_0_medium	wheat	0_medium	0	medium
wheat_grow_0_low	wheat	0_low	0	low
wheat_grow_1_high	wheat	1_high	1	high
wheat_grow_1_medium	wheat	1_medium	1	medium
wheat_grow_1_low	wheat	1_low	1	low
wheat_grow_2_high	wheat	2_high	2	high
wheat_grow_2_medium	wheat	2_medium	2	medium
wheat_grow_2_low	wheat	2_low	2	low

Use of `mpmasql` function language in class definitions

For field name and value definitions you can use `mpmasql` function language (MFL) expressions. You can also use MFL expressions to create complex element identifier, if you append `by` to the element type and provide a fixed element identifier part. The resulting element identifier will be <instance identifier>_<fixed identifier>_<result of MFL expression>.

```
#class <class1name> {
     #<elementtype>by <fixed identifier>  <MFL expression> {
           ...
     }
}
```

For model element identifiers and field names the result of any MFL expression is interpreted as a list, and consequently if the resulting list contains several items, one element/field for each of the list items is created, if the list is empty, no element/field is created at all. With a few exceptions, value definitions can only be given as MFL expressions that result in a scalar value. (If they receive a list, they usually use the first item in the list only).

You can refer to the identifier of the instance currently created with `#this(instance)` in any MFL expression in the whole class. You can refer to the current non-fixed part of the element identifier with `#this(by,0)` and to the current field created with `#this(field,0)`. Any number i greater than zero will refer to the current element of the ith array that was used in the element/field name definition. For example:

```
#class CROP {
   #activityby grow #foreach(SOIL)_#foreach(INTENSITY) {
      ...
      #table(crop_products,0,[#this(instance)])"_balance"  #= -1 *
         #table(yields,4,[#this(instance), #this(field,1),
         #this(by,1), #this(by,2))
      ...
   }
   ...
}
```

Let's assume the instance list CROP contained a 'wheat' instance, let's further assume the user-defined lists SOIL was [0, 1, 2] and the user defined list INTENSITY contained [high, medium, low], then the above element definition would return 9 activities. Then the values return by the local variables accessed through the `#this()` function would be as shown in Table B.34.

Further, let's assume that the table query `#table(crop_products,0,[#this(instance)])"_balance"` returned a list with two elements [wheat, straw] the each of the 9 activities would have a field wheat_balance

and straw_balance corresponding to the coefficients of the activity in the constraints of the same name. Then the `#this(field,1)` in the value expression would evaluate to "wheat" for the first and "straw" for the second field.

Reusing elements of other classes

You may have classes, which are pretty similar and differ only in a few elements. For example, you could have a class PRODUCTS and a class INTERMEDIATES. Both need a yield balance constraint and a selling activity (assuming there is a market for intermediate products). However, the class INTERMEDIATES additionally needs a buying activity and a transformation activity, which e.g. transforms hay into energy and raw fodder input for animal production.

To save you time and ensure that any changes you make to the elements in the PRODUCTS class also immediately apply to the INTERMEDIATES class, you can just let INTERMEDIATES use the selling activity and the yield balance constraint from the PRODUCTS class by adding an `#has () elements like` statement. Or, as syntactic sugar, if you think if your class name is in plural your statement should also be in plural, you might opt for `#have () elements like`, too. Or, if you prefer standard object-oriented language you can write `#inherits () from`.

Assuming you already defined a PRODUCTS class with a contraint 'balance' and an activity 'sell', you write the following, to include the same elements into the INTERMEDIATES class and additionally define a 'buy' and a 'transform' activity.

```
#class INTERMEDIATES {
      #has (balance, sell) elements like PRODUCTS

      #activity buy {
              ....
      }
      #activity transform {
             .......
      }
}
```

An asterisk (*) instead of a list of elements, uses all elements of the parent class. Thus, as `balance` and `sell` are all the elements in the class PRODUCTS, you could have also written:

```
have (*) elements like PRODUCTS.
```

Further, you can override attributes of a reused class. If - because of whatever reason - you have stored your intermediate product prices in a different table than your product prices (e.g. inputprices), you could still let INTERMEDIATE use the `sell` activity from PRODUCT and override only the `orow` attribute afterwards.

```
#class INTERMEDIATES {
      #has (balance, sell) elements like  PRODUCTS
      #activity sell {
              orow #= #table(inputprices, 3, [#thisinst])
      }

}
```

The resulting selling activity will have the same attributes as the one in the PRODUCT class, only the objective row attribute is defined differently. In this case it is, of course, important that you put the `has () elements like` statement before the overriding attribute definition. Otherwise, you will first define a new activity which is then overriden by the copied one.

Templates for standard classes

```
#class   MASSTANDARD {
   #constraint liqendow {
      name           #= "Liquidity endowment"
      unit              #= #var(CURRENCY)
      type           #=    MASliq
   }

   #constraint year1liq {
      name           #= "Year 1 liquidity"
      unit              #=    #var(CURRENCY)
      type           #= MASly1
     }
     #constraint  ongliq   {
      name           #= "Ongoing liquidity"
      unit              #=    #var(CURRENCY)
      type           #= MASong
     }
     #constraint  stcredit {
      name           #= "Short-term credit limit"
      unit              #=    #var(CURRENCY)
      type           #= MASscl
   }
   #activity   stcreditact {
      name           #= "Short-term credit"
      unit              #= #var(CURRENCY)
      type           #= MASscred
      orow           #= -1 * #var(INTEREST_SHCRD)
      #this(instance)_ongliq     #= -1
      #this(instance)_stcredit   #= 1
   }
   #activity   stdeposit {
      name           #= "Short-term deposits"
      type           #= MASdep
      unit              #=    #var(CURRENCY)
      orow           #= #var(INTEREST_SHDEP)
      #this(instance)_ongliq     #= 1
   }
   #activity   liqtrans {
      type           #= transfer
      name           #= "Transfer Liquidity"
      unit              #= #var(CURRENCY)
      #this(instance)_ongliq     #= -1
      #this(instance)_year1liq   #= -1
      #this(instance)_liqendow   #= 1
   }
}
```

```
#class   MASSTANDARD {
   #constraint liqendow {
      name           #= "Liquidity endowment"
      unit              #= #var(CURRENCY)
```

```
   type            #=    MASliq
}

#constraint year1liq {
   name            #= "Year 1 liquidity"
   unit               #=    #var(CURRENCY)
   type            #= MASly1
  }
  #constraint ongliq {
   name            #= "Pre-harvest liquidity"
   unit               #=    #var(CURRENCY)
   type            #= MASong
  }
  #constraint stcredit  {
   name            #= "Short-term credit limit"
   unit               #=    #var(CURRENCY)
   type            #= MASscl
}
  #constraint liqendofyear {
   name            #= "End of year liquidity"
   unit               #=    #var(CURRENCY)
   type            #= MASliqend
   eqtype             #= 3
  }
  #constraint stcredit_balance   {
   name            #= "Short-term credit balance"
   unit               #=    #var(CURRENCY)
   type            #= MASscbal
}
  #constraint stcredit_defyno {
   name            #= "Short-term credit potential default"
   unit               #=    #var(CURRENCY)
   type            #= MASscdef
}
#activity stcredit_repaid {
   name            #= "Short-term credit repaid"
   unit               #= #var(CURRENCY)
   type            #= MASscred
   orow            #= -1 * #var(INTEREST_SHCRD)
   #this(instance)_liqendofyear     #= 1 + #var(INTEREST_SHCRD)
   #this(instance)_stcredit_balance #= -1
}
#activity stcredit_taken {
   name            #= "Short-term credit taken"
   unit               #= #var(CURRENCY)
   type            #= MASscborrow
   consfix         #= 1
   #this(instance)_ongliq  #= -1
   #this(instance)_stcredit   #= 1
   #this(instance)_stcredit_balance #= 1
}
#activity stcredit_defaulted {
   name            #= "Short-term credit defaulted"
   unit               #= #var(CURRENCY)
   type            #= MASscdefault
   #this(instance)_stcredit_balance #= 0
```

```
      #harvest> #this(instance)_stcredit_balance   #= -1
      #this(instance)_defyno  #= 1
   }
   #activity remaining_cash {
      name           #= "Remaining cash end of year"
      unit              #= #var(CURRENCY)
      type           #= MASrcash
      type_of_payment      #= 0
      #this(instance)_liqendofyear #= 1
   }
   #activity stdeposit {
      name           #= "Short-term deposits"
      type           #= MASdep
      unit              #=    #var(CURRENCY)
      orow           #= #var(INTEREST_SHDEP)
      #this(instance)_ongliq     #= 1
      #this(instance)_liqendofyear  #= - (1 + #var(INTEREST_SHDEP) )
   }
   #activity liqtrans {
      type           #= transfer
      name           #= "Transfer start liquidity to pre-harvest
         liquidity"
      unit              #= #var(CURRENCY)
      #this(instance)_ongliq     #= -1
      #this(instance)_year1liq   #= -1
      #this(instance)_liqendow   #= 1
   }
   #activity liqtransend {
      type           #= transfer
      name           #= "Transfer remaining pre-harvest liquidity to
         end of year cash"
      unit              #= #var(CURRENCY)
      #this(instance)_ongliq     #= 1
      #this(instance)_liqendofyear  #= -1
   }
   #activity penalty_for_credit_default {
      type           #= MASpenalty
      name           #= "Utility penalty for defaulting on credits"
      unit              #= #var(CURRENCY)
      integ          #= 1
      ubound            #= 1
      orow           #= -1 * #var(UP_CREDIT_DEFAULT)
      #this(instance)_stcredit_defyno  #= -1 * #var(MAX_DEFAULT)
   }
}



#class SOILS {
   #constraint soil {
      name  #= "Soil of type " ~ #this(instance)
      unit     #=    ha
      type  #=    MASsoil
      eqtype   #=  3
   }
```

```
}



 #class LABOR {
   #constraint hhlabor {
      name  #= "Household labor of type: " ~#this(instance)
      unit     #=  "person year"
      type  #=  MASlab
      labgrp   #=  #this(instance)
   }
   #constraint labor_capacity {
      name  #= "Capacity of labor of type: " ~ #this(instance)
      unit     #= "person year"
   }
   #activity perlabin {
      name  #= "Hiring in permanent labor of type: " ~ #this(instance)
      unit     #= persons
      type  #= MASlabpin
      integ #= 1
      ubound   #= #var(LIMIT_LABOR_PERM)
      orow  #= -1*  #var(WAGE_PERMANENT)
      #this(instance)_labor_capacity   #= -1
   }
   #activity hhlabout {
      name  #= "Hiring out household labor of type: " ~
         #this(instance)
      unit     #= persons
      type  #= MASlabpout
      orow  #= 1*  #var(WAGE_HHOUSEHOLD)
      #this(instance)_hhlabor #= 1
    }
    #activity labor_transfer {
      name  #= "Transfer household labor of type:" ~#this(instance)
      unit     #=  persons
      #this(instance)_hhlabor #= 1
      #this(instance)_labor_capacity   #= -1
   }
}


#class MASCONSUMPTION {
'income top block

   #constraint xc_inc_transc  {
      name     #= "Income Transfer"
      type     #= MASxc_inc_trans
      unit     #=    #var(CURRENCY)
      eqtype      #= 3
   }
   #constraint xc_inc_funcc   {
      name     #= "Income Function"
      type     #= MASxc_inc_func
      unit     #=    #var(CURRENCY)
      eqtype      #= 3
```

```
   }
   #activity   xc_inc_transa  {
      name     #= "Income Transfer"
      type     #= MASxc_inc_trans
      unit     #=    #var(CURRENCY)
      #this(instance)_xc_inc_funcc  #=      -1
      #this(instance)_xc_inc_transc #= 1
   }
   #activity   xc_inc_funca   {
      name     #= "Income Function"
      type     #= MASxc_inc_func
      unit     #=    #var(CURRENCY)
      #this(instance)_xc_inc_transc #= -1
      #this(instance)_xc_sav_transc #= -1
      #this(instance)_xc_sav_balance   #= 1
   }

'savings block
   #constraint xc_sav_transc  {
      name     #= "Savings Transfer"
      type     #= MASxc_sav_trans
      unit     #=    #var(CURRENCY)
      eqtype      #= 3
   }
   #constraint xc_sav_funcc   {
      name     #= "Savings Function"
      type     #= MASxc_sav_func
      unit     #=    #var(CURRENCY)
      eqtype      #= 3
   }
   #constraint xc_sav_balance    {
      name     #= "Savings Balance INC = EXP + SAV"
      type     #= MASxc_sav_balance
      unit     #=    #var(CURRENCY)
      eqtype      #= 3
   }
   #constraintby   xc_sav_lbound  #steps(1, #count(XC_SAV_INCSEGS),
      1) {
      name     #= "Savings Segment lowerbound "~ #this(by,0)
      type     #= MASxc_sav_lbound
      unit     #=    #var(CURRENCY)

   }
   #constraintby   xc_sav_ubound  #steps(1, #count(XC_SAV_INCSEGS),
      1) {
      name     #= "Upperbound Savings Segment"~ #this(by,0)
      type     #= MASxc_sav_ubound
      unit     #=    #var(CURRENCY)
   }
   #activity   xc_sav_funca {
      name     #= "Savings Function"
      type     #= MASxc_sav_func
      unit     #=    #var(CURRENCY)
      #this(instance)_xc_sav_funcc  #= -1
      #this(instance)_xc_sav_balance   #= -1
```

```
   }
   #activity   xc_sav_seg_0   {
      name     #= "Savings Segment 0"
      type     #= MASxc_sav_seg0
      unit     #=    #var(CURRENCY)
      #this(instance)_xc_sav_transc    #= 1
      #this(instance)_xc_sav_funcc  #= 0
      #this(instance)_xc_sav_lbound_#steps(1, #count(XC_SAV_INCSEGS),
         1)   #= -1
' ubound set by MP-MAS

   }
   #activityby xc_sav_seg #steps(1, #count(XC_SAV_INCSEGS)-1, 1)     {
      name     #= "Savings Segment "~#this(by,0)
      type     #= MASxc_sav_seg2
      unit     #=    #var(CURRENCY)
      #this(instance)_xc_sav_transc    #= 1
      #this(instance)_xc_sav_lbound_#this(by,0) #=     -1
      #this(instance)_xc_sav_ubound_#this(by,0) #= 1
      ubound      #= #listelement(XC_SAV_INCSEGS, #this(by,0))
   }
   #activity   xc_sav_seg_last  {
      name     #= "Savings Segment "
      type     #= MASxc_sav_seg3
      unit     #=    #var(CURRENCY)
      #this(instance)_xc_sav_transc    #= 1
      #this(instance)_xc_sav_ubound_#count(XC_SAV_INCSEGS)  #= 1
      ubound      #= #listelement(XC_SAV_INCSEGS,
         #count(XC_SAV_INCSEGS))
   }
   #activity   xc_sav_bin_1  {
      name     #= "Savings Binary 1"
      type     #= MASxc_sav_bin1
      unit     #=    #var(CURRENCY)
      integ    #= 1
      ubound      #= 1
      #this(instance)_xc_sav_ubound_1  #= -1 *
         #listelement(XC_SAV_INCSEGS, 1)
   }
   #activityby xc_sav_bin #steps(2, #count(XC_SAV_INCSEGS), 1)    {
      name     #= "Savings Binary  "~#this(by,0)
      type     #= MASxc_sav_bin2
      unit     #=    #var(CURRENCY)
      integ    #= 1
      ubound      #= 1
      #this(instance)_xc_sav_lbound_#this(by,0)    #=
         #listelement(XC_SAV_INCSEGS, #this(by,0) -1)
      #this(instance)_xc_sav_ubound_#this(by,0) #= -1 *
         #listelement(XC_SAV_INCSEGS, #this(by,0))
   }
'food - non food block
   #constraint xc_expenditure_balance {
      name     #= "Expenditure balance"
      type     #= MASxc_expbal
      unit        #=    #var(CURRENCY)
      eqtype      #= 3
```

```
}
#constraint xc_fnf_funcc   {
   name     #= "Food Expenditure Function"
   type     #= MASxc_fnf_func
   unit     #=    #var(CURRENCY)
   eqtype      #= 3
}
#constraint xc_fnf_constc  {
   name     #= "Food Expenditure Constant"
   type     #= MASxc_fnf_const
   unit     #=    #var(CURRENCY)
   eqtype      #= 3
}
#constraint xc_fnf_hhsizec {
   name     #= "Food Expenditure HHSize"
   type     #= MASxc_fnf_hhsize
   unit     #=    #var(CURRENCY)
   eqtype      #= 3
}
#constraint xc_fnf_texc {
   name     #= "Food Expenditure TEX"
   type     #= MASxc_fnf_tex
   unit     #=    #var(CURRENCY)
   eqtype      #= 3
}
#constraintby xc_fnf_lbound #steps(1, #count(XC_FNF_EXPSEGS) -1 ,
   1) {
   name     #= "Food Expenditure TEX Segment Lower
      Bound"~#this(by,0)
   type     #= MASxc_fnf_lbound
   unit     #=    #var(CURRENCY)
   eqtype      #= 1
}
#constraintby xc_fnf_ubound #steps(2, #count(XC_FNF_EXPSEGS) ,1) {
   name     #= "Food Expenditure TEX Segment Upper
      Bound"~#this(by,0)
   type     #= MASxc_fnf_ubound
   unit     #=    #var(CURRENCY)
   eqtype      #= 1
}

#activity xc_fnf_funca  {
   name     #= "Total Expenditure Function"
   type     #= MASxc_fnf_func
   unit     #=    #var(CURRENCY)
   #this(instance)_xc_sav_balance   #= -1
   #this(instance)_xc_fnf_constc #= 1
   #this(instance)_xc_fnf_hhsizec   #= 1
   #this(instance)_xc_fnf_texc   #= 1
   #this(instance)_xc_expenditure_balance #= -1
}

#activity xc_fnf_consta {
   name     #= "Food Expenditure Constant"
   type     #= MASxc_fnf_const
   unit     #=    #var(CURRENCY)
```

```
      #this(instance)_xc_fnf_constc #= -1
      #this(instance)_xc_fnf_funcc  #= #var(XC_FNF_COEF_CONST)
   }
   #activity xc_fnf_hhsizea   {
      name     #= "Food Expenditure HHSize"
      type     #= MASxc_fnf_hhsize
      unit     #=    #var(CURRENCY)
      #this(instance)_xc_fnf_hhsizec   #= -1
   }
   #activity xc_fnf_seg_1  {
      name     #= "Food Expenditure TEX Seg 1"
      type     #= MASxc_fnf_seg1
      unit     #=    #var(CURRENCY)
      #this(instance)_xc_fnf_texc   #= -1
      #this(instance)_xc_fnf_lbound_1 #= -1
      ubound      #= #listelement(XC_FNF_EXPSEGS,1)
   }
   #activityby xc_fnf_seg #steps(2, #count(XC_FNF_EXPSEGS)-1  , 1)   {
      name     #= "Food Expenditure TEX Seg "~#this(by,0)
      type     #= MASxc_fnf_seg2
      unit     #=    #var(CURRENCY)
      #this(instance)_xc_fnf_texc   #= -1
      #this(instance)_xc_fnf_lbound_#this(by,0) #= -1
      #this(instance)_xc_fnf_ubound_#this(by,0) #= 1
      ubound      #= #listelement(XC_FNF_EXPSEGS,#this(by,0))
   }
   #activity xc_fnf_seg_last    {
      name     #= "Food Expenditure TEX Seg "
      type     #= MASxc_fnf_seg3
      unit     #=    #var(CURRENCY)
      #this(instance)_xc_fnf_texc   #= -1
      #this(instance)_xc_fnf_ubound_#count(XC_FNF_EXPSEGS)  #= 1
      ubound      #= #listelement(XC_FNF_EXPSEGS,
         #count(XC_FNF_EXPSEGS))

   }

   #activityby xc_fnf_bin #steps(1, #count(XC_FNF_EXPSEGS) -1 , 1)   {
      name     #= "Food Expenditure TEX Binary "~ #this(by,0)
      type     #= MASxc_fnf_bin
      unit     #=    #var(CURRENCY)
      integ    #= 1
      ubound      #= 1
      #this(instance)_xc_fnf_lbound_#this(by,0) #=
         #listelement(XC_FNF_EXPSEGS,#this(by,0))
      #this(instance) "_xc_fnf_ubound_"  (#this(by,0) + 1) #= -1 *
         #listelement(XC_FNF_EXPSEGS,#this(by,0) + 1)
   }
   #activity xc_nonfood_expenditure {
      name     #= "Non-food expenditure"
      type     #= MASxc_nf_exp
      unit     #=    #var(CURRENCY)
      #this(instance)_xc_expenditure_balance #= 1
      #foreach(MASSTANDARD)_liqendofyear  #= 1
   }
'LA/AIDS Food category block
```

```
#constraintby xc_foo_foodcat #table(foodcats,0,[]) {
   name     #= "Food Category"~#table(foodcats,2,[#this(by,0)])
   type     #= MASxc_foo_foodcat
   unit     #= #var(CURRENCY)
   #consumption>category   #= #this(by,0)
   #consumption>xc_foo_coeff_expsi     #=
      #table(foodcats,5,[#this(by,0)])
   #consumption>xc_foo_coeff_hh     #=
      #table(foodcats,6,[#this(by,0)])
   #consumption>xc_foo_crossprice_#table(foodcats,0,[])  #=
      #table(crossprice, 3, [#this(by,0), #this(field,1)])
   #consumption>budget_share     #=
      #table(foodcats,7,[#this(by,0)])
}
#constraint xc_foo_constc  {
   name     #= "LA/AIDS Constant"
   type     #= MASxc_foo_const
   unit     #=    #var(CURRENCY)
   eqtype      #= 3
}
#constraint xc_foo_hhsizec {
   name     #= "LA/AIDS HHSize"
   type     #= MASxc_foo_hhsize
   unit     #=    #var(CURRENCY)
   eqtype      #= 3
}
#constraint xc_foo_pricec  {
   name     #= "LA/AIDS Price"
   type     #= MASxc_foo_price
   unit     #=    #var(CURRENCY)
   eqtype      #= 3
}
#constraint xc_foo_fexc {
   name     #= "LA/AIDS Food expenditure"
   type     #= MASxc_foo_fexc
   unit     #=    #var(CURRENCY)
   eqtype      #= 3
}
#constraintby xc_foo_lbound #steps(1, #count(XC_FOO_FEXSEGS) -1 ,
   1) {
   name     #= "LA/AIDS FEX Segment Lower Bound"~ #this(by,0)
   type     #= MASxc_foo_lbound
   unit     #=    #var(CURRENCY)
   eqtype      #= 1
}
#constraintby xc_foo_ubound #steps(2, #count(XC_FOO_FEXSEGS)  , 1)
   {
   name     #= "LA/AIDS FEX Segment Upper Bound"~ #this(by,0)
   type     #= MASxc_foo_ubound
   unit     #=    #var(CURRENCY)
   eqtype      #= 1
}
#activity xc_foo_funca  {
   name     #= "Food Expenditure Function"
   type     #= MASxc_foo_func
   unit     #=    #var(CURRENCY)
```

```
   #this(instance)_xc_expenditure_balance #= 1
   #this(instance)_xc_fnf_funcc      #= -1
   #this(instance)_xc_foo_constc     #= 1
   #this(instance)_xc_foo_hhsizec            #=  1
   #this(instance)_xc_foo_pricec     #= 1
   #this(instance)_xc_foo_fexc       #= 1
}
#activity xc_foo_consta {
   name      #= "LA/AIDS Constant"
   type      #= MASxc_foo_const
   unit      #=    #var(CURRENCY)
   #this(instance)_xc_foo_constc     #= -1
   #this(instance)_xc_foo_foodcat_#table(foodcats,0,[])   #=
      #table(foodcats,4,[#this(field,1)])
}
#activity xc_foo_hhsizea   {
   name      #= "LA/AIDS HHSize"
   type      #= MASxc_foo_hhsize
   unit      #=    #var(CURRENCY)
   #this(instance)_xc_foo_hhsizec   #= -1
             #this(instance)_xc_foo_foodcat_#table(foodcats,0,[])
                #=   #table(foodcats,6,[#this(field,1)])
}
#activity xc_foo_pricea {
   name      #= "LA/AIDS Price"
   type      #= MASxc_foo_price
   unit      #=    #var(CURRENCY)
   #this(instance)_xc_foo_pricec     #= -1
             #this(instance)_xc_foo_foodcat_#table(foodcats,0,[])
                #=   #table(foodcats,5,[#this(field,1)])
}
#activity xc_foo_seg_1  {
   name      #= "LA/AIDS FEX Seg 1"
   type      #= MASxc_foo_seg1
   unit      #=    #var(CURRENCY)
   #this(instance)_xc_foo_fexc       #= -1
   #this(instance) "_xc_foo_lbound_1"  #= -1
   ubound       #= #listelement(XC_FOO_FEXSEGS,1)
}
#activityby xc_foo_seg #steps(2, #count(XC_FOO_FEXSEGS)-1  , 1)   {
   name      #= "LA/AIDS FEX Seg "~#this(by,0)
   type      #= MASxc_foo_seg2
   unit      #=    #var(CURRENCY)
   #this(instance)_xc_foo_fexc       #= -1
   #this(instance)_xc_foo_lbound_#this(by,0) #= -1
   #this(instance)_xc_foo_ubound_#this(by,0) #= 1
   ubound       #= #listelement(XC_FOO_FEXSEGS,#this(by,0))
}
#activity xc_foo_seg_last    {
   name      #= "LA/AIDS FEX  Seg "~#count(XC_FOO_FEXSEGS)
   type      #= MASxc_foo_seg3
   unit      #=    #var(CURRENCY)
   #this(instance)_xc_foo_fexc       #= -1
   #this(instance)_xc_foo_ubound_#count(XC_FOO_FEXSEGS) #= 1
   ubound       #=
      #listelement(XC_FOO_FEXSEGS,#count(XC_FOO_FEXSEGS))
```

```
   }

   #activityby xc_foo_bin #steps(1, #count(XC_FOO_FEXSEGS) -1 , 1)   {
      name      #= "LA/AIDS FEX Binary " ~#this(by,0)
      type      #= MASxc_foo_bin
      unit      #=    #var(CURRENCY)
      integ     #= 1
      ubound       #= 1
      #this(instance)_xc_foo_lbound_#this(by,0) #=
         #listelement(XC_FOO_FEXSEGS,#this(by,0))
      #this(instance)_xc_foo_ubound_(#this(by,0) + 1)#= -1 *
         #listelement(XC_FOO_FEXSEGS,#this(by,0) + 1)
   }
'Food category transfers
   #constraintby xc_foo_cattransc #table(foodcats,0,[]) {
      name      #= "Food Category  Transfer:
         "~#table(foodcats,2,[#this(by,0)])
      type      #= MASxc_foo_cattrans
      unit      #= #var(CURRENCY)
      eqtype       #= 3
      #consumption>category   #= #this(by,0)
   }
   #activityby xc_foo_cattransa #table(foodcats,0,[]) {
      name      #= "Food Category Transfer:
         "~#table(foodcats,2,[#this(by,0)])
      type      #= MASxc_foo_cattrans
      unit      #= #var(CURRENCY)
      #this(instance)_xc_foo_cattransc_#this(by,0) #=  1
      #this(instance)_xc_foo_foodcat_#this(by,0)      #= -1
   }
   #activityby buy_foodcat #table(foodcats,0,[]) {
      name      #= "Buy_"~#table(foodcats,2,[#this(by,0)])
      type      #= MASfoodbuy
      orow      #= -1 * #table(foodcatprice ,3, [#this(by,0),
         #var(PRICEYEAR)])
      orow_#foreach(YEARS) #= -1 * #table(foodcatprice ,3,
         [#this(by,0), #var(PRICEYEAR)])
      #this(instance)_hhnutrient_supplyc_#foreach(HHNUTRIENTS) #=
         #table(food_category_nutrients , 3 , [#this(by,1),
         #this(field,1)])
      #consumption>category   #= #this(by,0)
                #this(instance)_xc_foo_cattransc_#this(by,0)   #=  -1
                   * #table(food_category_prices ,3, [#this(by,1),
                   #var(PRICEYEAR)])
   }
'Balancing Food Requirements

   #constraintby hhnutrient_demandc HHNUTRIENTS {
      name      #= #this(by,0)~" demand"
      type      #= MASnutdemand
      eqtype       #= 3
      #consumption>nutrient #= #this(by,0)
   }
   #constraintby hhnutrient_supplyc HHNUTRIENTS {
      name      #= #this(by,0) ~" supply"
      type      #= MASnutsupply
```

```
      eqtype       #= 3
   }
   #constraintby hhnutrient_deficitc HHNUTRIENTS {
      name     #= #this(by,0)~" deficit"
      type     #= MASnutdeficit
      eqtype       #= 3
   }
   #constraintby hhnutrient_balance HHNUTRIENTS {
      name     #= #this(by,0) ~" supply"
      type     #= MASnutbalance
      eqtype       #= 1
   }
   #activityby hhnutrient_demanda HHNUTRIENTS {
      name     #= #this(by,0)~" demand"
      type     #= MASnutdemand
      #this(instance)_hhnutrient_balance_#this(by,0)  #= 1
      #this(instance)_hhnutrient_demandc_#this(by,0)  #= 1
   }
   #activityby hhnutrient_supplya HHNUTRIENTS {
      name     #= #this(by,0) ~" supply"
      type     #= MASnutsupply
      #this(instance)_hhnutrient_supplyc_#this(by,0)  #= -1
      #this(instance)_hhnutrient_balance_#this(by,0)  #= -1
   }
   #activityby hhnutrient_deficita HHNUTRIENTS {
      name     #= #this(by,0)~" deficit "
      type     #= MASnutdeficit
      #this(instance)_hhnutrient_balance_#this(by,0)  #= -1
      #this(instance)_hhnutrient_deficitc_#this(by,0) #= 1
   }
   #activityby utility_penalty_hhnutrient HHNUTRIENTS {
      name     #= #this(by,0)~" deficit "
      type     #= MASnutpenalty
      orow     #= -1 * #table(utility_penalty, 2, [#this(by,1)])
      #this(instance)_hhnutrient_deficitc_#this(by,0) #= -1
   }

}
```

Appendix C

mpmasql function language

`mpmasql` and `mpmasdist` use a special function language that can be used at specific points in their input files to express variable values. A piece of function code that defines a value is generally called an expression. During the preparation of input files `mpmasql` may evaluate an expression many times under many different circumstances (e.g. for different instances and scenarios) and arrive at very different values depending on the values that certain variables have under these circumstances.

Expressions can evaluate to single values (like a number or a string), empty results or a list of values (array).

C.1 Numbers and Strings

The most simple expressions only consists of a number or a single string, e.g.

```
 1
-1
0.0753
crop
"Hire labor "
```

Such expressions evaluate to itself, except for double quoted strings, where the resulting value is only the string inside the double quotes. Generally, strings without quotes are allowed as long as they consist only of letters of the (English) alphabet, digits and underscores. If they include spaces or punctuation they have to be included in double quotes. In general numbers can also be treated as strings, but strings not necessarily as numbers.

C.2 Operators

Numerical operators evaluate to what you would expect: `1 +1` result in 2 and `2^4` in 16. Logical and comparison operators result in 1 whenever for true, or 0 for false. Logical operators accept any value unequal to zero or an empty value as true. Logical operators are short-circuit, i.e. if the condition left of an `||` evaluates to true, the right hand condition is not evaluated any more, and the result is true, likewise if the condition on the left of an && is false, the right hand condition is not evaluated and the result is false. The $\sim$ operator forms one string out of two.

Table C.1: Operators

Operator	Description
	Strings
~	concatenation (optional)
	Simple arithmetics
+	addition
-	subtraction
*	multiplication
/	division
^	exponentiation
	Numerical comparison
==	equal
!=	not equal
>=	greater or equal
<=	smaller or equal
>	greater than
<	smaller than
	Logical
&&	AND
||	OR
!	NOT

C.3 Arrays

Arrays can be formed using square brackets and commas, e.g. `[1,2,3, "wheat"]` is a four element array with the first three natural numbers and the string "wheat" as its elements. Arrays can also result from certain functions (e.g. `#table`, `#foreach`, `#sql`). Arrays remain arrays when an arithmetic or string operator is applied in combination with a scalar, and form Cartesian products (combinations of every element with every other element) whenever an operator acts on two arrays. When a numerical comparison operator is applied to an array, the size of the array is used as an argument, while logic operators take an array to be true whenever it is not empty.

e.g.

```
    [1,2,3] * 3  = [3,6,9]

    [1,2,3] * [1,2,3] = [1,2,3,2,4,6,3,6,9]

    [1,2,3] > [1,2] = 1
    [1,2,3] > 5    = 0
    [] && [1,2,3]  = 0

    ["wheat", "maize", "barley"]  " on soil " [0,1] =
 ["wheat on soil 0", "wheat on soil 1", "maize on soil 0",  ....]
```

The behavior of functions, when applied to an array depends on the function. In general just a few functions can deal with arrays at all.

C.4 Functions

Functions take a number of arguments, i.e. numbers, strings or arrays or any expression resulting in either of them. Of course they can be nested.

#abs returns the absolute value of the expression

```
#abs(<expression>)
```

#annuity calculates an annuity

```
#annuity(<interest rate>, <present value>, <lifetime>)
```

#count counts the elements of an array

```
#count(<array>)
```

#floor returns the integer part of a number, i.e. removes any decimals.

```
#floor(<number>)
```

#foreach inserts a user defined array or variable as an array

```
#foreach(<user-defined array>)
```

#fvalue financial function used to calculate the coefficient of average fixed equity on an investment

$$\frac{(i+1)^t}{(i+1)^{(t-1)}} - \frac{1}{t*i}$$

```
#fvalue(<interest>, <lifetime>)
```

#if returns a value depending on a condition

```
#if(<condition>, <value if true>, <value if false>)
```

#ifalone distinguishes field values between conversion modes. The setup of some matrix elements differs depending on whether you are converting standalone matrix files or full MP-MAS input files. For example, investment activities should have a 0 in the constraint they add an endowment to in full MP-MAS mode (as this will be entered by MP-MAS according to the network file), while they need an entry there in Standalone Matrix Mode.

```
#ifalone(<value when in standalone mode>, <else>)
```

#ifthenelse like #if, but the condition is evaluated as a perl expression (allows the use of string comparison, regular expressions etc)

```
#ifthenelse(<condition>, <value if true>, <value if false>)
```

#list converts an array into a string containing the comma-separated list of the unquoted array values.

```
#list(<array or list>)
```

#list_concat concatenates several arrays.

```
#list_concat(<array1>, <array2>, ...)
```

#list_element inserts the ith element of an array .

```
#list_element(<array or list>,<index>)
```

#mapcell retrieves a value from a cell in a raster map (currently only in `mpmasdist`).

```
#mapcell(<mapname>,<x coordinate>,<y coordinate>)
```

#max returns the maximum of a list of values

```
#max(#var(VARIABLE1) , #var(VARIABLE2) , ...)
```

#min returns the minimum of a list of values

```
#min(#var(VARIABLE1) , #var(VARIABLE2) , ...)
```

#protected_divide performs a usual division, but returns zero instead of an error in case of a division by zero or an empty array

```
#protected_divide(<dividend>, <divisor>)
```

#qlist converts an array into a string containing the comma-separated list of the double-quoted array values. (Useful to construct IN subexpressions in SQL statements

```
#qlist(<array>)
```

#random_number draws a random number from a uniform distribution on the interval between 0 (inclusive) and the argument (exclusive).

```
#random_number(<upper bound>)
```

#regexp compares a string to a regular expression (see here `http://perldoc.perl.org/perlre.html` for information on the Perl regular expression syntax)

```
#regexp(<strings to check>, <regular expression>)
```

#replace_null replaces an empty result by the specified argument.

```
#replace_null(<function result to be replaced if null>,
   <replacement value>)
```

#report is a debugging function. It prints arguments to screen during expression evaluation. Inserts all but the first two of its arguments .

```
#report(id, condition, arguments to report, ...)
```

id is an id you can freely choose, it is meant to avoid confusion if you use several reports

condition report is only written to screen if condition is true so you can e.g. restrict it to certain buggy instances

The report function returns all except the first two arguments to the expression, so the actual calculation is not affected, even if you put it inside an expression.

Examples:

```
orow   #=    #report(#this(instance),1, -1 *
   #table(prices, 2,[#this(instance)]))
```

→ prints the result for the field orow for every instance of the class on the screen and each report is identified by the instance id

```
orow   #=    #report(test,#this(instance) == 24, -1 *
   #table(prices, 2,[#this(instance)]))
```

→ prints the result for the field orow for instance 24 onto the screen and the report is identified as test

```
orow   #=    -1 * #report(test,#this(instance) == 24,
   #table(prices, 2,[#this(instance)]))
```

→ prints the value read from the table for instance 24 onto the screen and the report is identified as test

#round_to rounds a number to specified number of decimals

```
#round_to(<number>, <decimals>)
```

#sql retrieves data from an sql database.

```
#sql(<SQL select statement>)
```

#steps evaluates to an array of values of equal distance

```
#steps(<start value>, <end value>, <stepsize>)
```

#table inserts a value or array read from a user-defined table.

```
#table(<tablename>, <column index>, <keyarray>)
```

Internal tables in `mpmasql` and `mpmasdist` are indexed by a number of columns, which was defined when the table was loaded. As an example, let's consider the following example table named members and assume it has two key fields, i.e. the first two fields are supposed to uniquely identify each row in the table:

farm	member	gender	age
1	2	m	20
1	4	f	1
3	1	f	71
3	2	m	55

The code `#table(members, 4, [2, 1])` would return the value '71', i.e. the value which can be found in the fourth column of the row whose first column is 2 and whose second column is 1. If the key array of the table function contains less keys than the number of key columns defined for the table, the table function returns a list of all unique values in the first omitted key column (the column index is then ignored). E.g. `#table(members, 0, [])` would return [1,3] and `#table(members, 0, [1])` would return [2,4].

#this inserts the value of a local variable defined by `mpmasql` or `mpmasdist`

```
#this(<local variable>)
#this(<local array>, <arrayindex>)
#this(<local object>, <attribute name> )
```

#var inserts the value of a user-defined variable, global, MP-MAS parameter or configuration entry

```
#var(<variable name>)
```

Appendix D

Version history and changelog

D.1 Version History

D.2 Changes from version 1.03 to 2.00

MPMASQL version 2.00 comes with some convenient new features:

- much faster conversion for large models (the CSA model converts about 2 times faster than before)
- the possibility to have table definitions in .var split over several lines to increase readability
- the possibility to read in tab-separated ascii tables instead of sql queries as table information in fact you do not actually need a MySQL database anymore to work with mpmasql), still experimental
- the introduction of block comments /* */ to comment whole sets of lines
- syntax highlighting in the gedit editor
- improved structure of input files, more logical distribution of entries over files
- better naming for some entries
- special ordered integer sets in the matrix
- improved functionality in mqlmatrix
- the possibility to use ASSET_ENDOWMENT information for MPMAS to create right hand sides for single farm analysis
- `mpmasdist`, a tool to create agent populations

However, all of this comes with some major costs, you actually have to do some major changes to your input files and the old manual and tutorial are outdated. I will rework the documentation and files in the next days. This document already provides an overview of the changes you have to make in order to use mpmasql 2.0.

To update to mpmasql version 2.0, you have to do the following changes in your mpmasql input files:

Table D.1: Version History

mpmasql	Date	MPMAS executable	Default Input Dataset
2.52	13 Aug 2014	3.3.244	DEF241
2.48	13 June 2014	3.3.230	DEF241
2.47	08 May 2014	3.3.223	DEF240
2.40	21 Feb 2014	3.3.215	DEF239
2.39	12 Dec 2013	3.3.210	DEF239
2.38	11 Dec 2013	3.3.210	DEF239
2.36	17 Sep 2013	3.3.204	DEF238
2.34	8 Aug 2013	3.3.196	DEF237
2.33	5 Aug 2013	3.3.194	DEF236
2.32	30 Jul 2013	3.3.193	DEF235
2.28	6 May 2013	3.3.186	DEF233
2.27	26 March 2013	3.3.180	DEF229
2.26	24 March 2013	3.3.180	DEF229
2.23	30 January 2013	3.3.167	DEF224
2.22	16 January 2013	3.3.160	DEF222
2.19	15 November 2012	3.3.143	DEF217
2.14	4 September 2012	3.3.137	DEF209
2.12	16 July 2012	3.3.136	DEF209
2.10	04 July 2012	3.3.135	DEF207
2.08	10 May 2012	3.3.134	DEF206
2.06	26 April 2012	3.3.130	DEF206
2.05	20 April 2012	3.3.130	DEF205
2.04	03 April 2012	3.3.130	DEF204
2.02	26 March 2012	3.3.128	DEF202
2.00	03 March 2012	3.3	DEF200
1.05 beta	22 Feb 2012	3.3	DEF200
1.04 beta	12 Jan 2012	3.3	DEF192
1.03 beta	9 Dec 2011	3.1	DEF182
1.02 beta	16 Nov 2011	3.1	DEF176
0.97 beta	9 June 2011	3.1	DEF169
0.94 beta	23 May 2011	3.1	DEF164
0.93 beta	20 May 2011	3.1	DEF164
0.90 beta	10 May 2011	3.1	DEF161
0.88 alpha	15 Feb 2011	3.0	DEF155
0.87 alpha	21 Dec 2010	3.0	-
0.86 beta	03 Dec 2010	3.0	-
0.85 beta	30 Jun 2010	3.0	DEF152 (2010-06-18/Linux)
0.84 alpha	15 Feb 2010	3.0	DEF141 (2010-02-15/Linux)
0.82 alpha	24 Sep 2009	3.0	DEF128 (2009-05-24/Linux)

D.2.1 Conversion control file (.ini)

1. Rename the following entries:
   (a) CONFIG → CONFIGURATION
   (b) CLASSES → STRUCTURE
   (c) VARIABLES → DATA
2. Adapt the SCENLIST, SCENDEF, RHSLIST, RHSDEF entires to the new table format
   (a) For SCENDEF you now have two options (apart from leaving it empty):
      i.
      ```
      SCENDEF =(sql) { "SQL SELECT statement which can also
            span over several lines" }
      ```
      ii. (but still experimental):
      ```
      SCENDEF =(ascii) { "name of an ascii file with tab
          separated values" }
      ```
   (b) For RHSDEF you have the same options as for SCENDEF plus a third one:
      i.
      ```
      RHSDEF  = (transform_assets)
      ```
      If you use this option, the RHS is directly created from the ASSET_ENDOWMENT, HOUSEHOLD_COMPOSITION and AGENT_INFO tables (in the .var file, see below), and the soil RHS is either directly read from MP-MAS input maps ,if you specify the map directory in the control file:
      ```
      RHS_MAPDIR  = directory
      ```
      , or alternatively read from a table specified in the control file:
      ```
      RHS_SOILTAB = (sql) {"SQL SELECT statement which can also
              span over several lines"}
      ```
   (c) For SCENLIST and RHSLIST you have three options (apart from leaving them empty):
      i.
      ```
      SCENLIST  = (sql) { "SQL SELECT statement which can also
            span over several lines" }
      ```
      ii. but still experimental:
      ```
      SCENLIST = (ascii) { "name of an ascii file with one entry
          on each new line" }

      SCENLIST = (list) {[ list of comma-separated values ]}
      ```

D.2.2 Configuration file (.cfg)

1. Rename the following entries if you are using them:
   (a) MAXSOLVETIME → OSL_MAXSOLVETIME
   (b) MAXITERATE → OSL_MAXITERATE
   (c) MAXNODES → OSL_MAXNODES
   (d) (there is an additional entry OSL_SKIPMODE1, which skips mode 1)

2. A whole number of entries will be moved out of the configuration file and into the new [MPMAS PARAMETERS] and [MPMAS TABLES] section in the data file (.var). Everything which is a table has to go to [MPMAS TABLES]. Everything which is can be considered model configuration (like start year, submodel switches, osl control, number of soils, populations, type of consumption, type of diffusion) will remain in cfg. Everything which is rather a model parameter (like interest rates, liquidity, leverage, overlap) should go to [MPMAS PARAMETERS], although the later would still work if they remain in the configuration file. (Tables won't work in cfg anymore). More detail is provided in the section on .var below.

3. In case you used an external crop growth model, the XNTABLE and XNYIELDCHANGE have been replaced by the attribute group `#externalcropgrowth>`, to be implemented in .cll

D.2.3 Data file (.var)

1. The data file now has six sections:

   (a) [GLOBALS]
   (b) [MPMAS PARAMETERS]
   (c) [MPMAS TABLES]
   (d) [USER TABLES]
   (e) [USER VARIABLES]
   (f) [INSTANCES]

   Please rename TABLES → USER TABLES and VARIABLES → USER VARIABLES, and introduce the [MPMAS PARAMETERS] and [MPMAS TABLES] sections. The [ORDER] section is moved to the structure file (.cll) and reformated (see below)

2. Reformat your user tables. The new format is:

```
tablename   = (tabletype, number of keys) { MpmasQL Function
   Language that results in an
                    SQL statement and can span several lines
                       }
```

   e.g.:

```
 ofert   = (sql,1) {"SELECT goods_code, 1 FROM tbl_consumables
    WHERE goods_code IN (" ~ #qlist(ORGANICFERT) ~")" }
```

   or, same thing, but more readable:

```
ofert = (sql,1) {
       "SELECT goods_code, 1
       FROM tbl_consumables
       WHERE goods_code IN (" ~ #qlist(ORGANICFERT) ~")" }
```

3. Move all tables from the configuration file into the new [MPMAS TABLES] section and adapt them to the new format. Important!: as these tables have a predefined structure you must not specify the number of keys.

   The following tables have to be renamed:

   (a) DEMOTAB → HOUSEHOLD_DYNAMICS
   (b) SEXAGECAT → HOUSEHOLD_MEMBER_TYPES
   (c) ASSETENDOW → ASSET_ENDOWMENTS
   (d) HOUSEHOLDENDOW → HOUSEHOLD_COMPOSITION

(e) AGENTINFO → AGENT_INFO
(f) ASSETCDF → ASSET_CDF
(g) HOUSEHOLDCDF → HOUSEHOLD_CDF

So e.g. if this was before in your .cfg:

```
SEXAGECAT  = #sql("SELECT code, kind, sex, lowage, upage FROM
    tbl_MASsexagecat")
```

it should now look like this in your .var:

```
HOUSEHOLD_MEMBER_TYPES  = (sql) {"SELECT code, kind, sex, lowage,
   upage FROM tbl_MASsexagecat"}
```

or like this would also work:

```
HOUSEHOLD_MEMBER_TYPES  = (sql) {
      "SELECT code, kind, sex, lowage, upage
       FROM tbl_MASsexagecat"
        }
```

4. Move all non-table entries, which you deem model parameters, rather than model configuration from .cfg to .var [MPMAS PARAMETERS]. This is optional and rather cosmetic in nature, to mpmasql it does not matter whether entries are in cfg or .var [MPMAS PARAMETERS], but I consider this more logical if we have a configuration and a data file. So in my thinking, configuration is for thinks that activate submodels, debugging features or provides information on number of soils, populations etc, while parameters is everything that is directly used during simulation of processes, like interest rates, consumption shares etc.

D.2.4 Model structure file (.cll)

1. First a good message, I have programmed syntax highlighting for .cll files So after installing mpmasql you should now be able to choose View->Highlight Mode->mpmasql->mpmasql(cll) in gedit (maybe you have to close gedit, and reopen it after installing) An have a nicely colored and better readable cll syntax.

   Also, try Edit->Preferences->View->Highlight matching brackets, very helpful.

2. The model structure file is now also subdivided into sections, which are as follows:

   (a) [ACTIVITY TYPES]
   (b) [CONSTRAINT TYPES]
   (c) [DEFAULTS]
   (d) [MODEL]

3. Move the [ORDER] section from the .var file here and adapt it to the new format.

   If it looked like this in your .var file before:

```
[ORDER]
ACTTYPEORDER = grow, hire, default
CONTYPEORDER = balance, default
```

   It should now look like this in your .cll file:

```
[ACTIVITY TYPES]
grow, hire, default

[CONSTRAINT TYPES]
balance, default
```

4. The [DEFAULTS] section replaces the default class. The default class is now read from /etc/mpmasql/default.cll, and you only need to include entries into the defaults section in case you want to change default setting compared to the settings in /etc/mpmasql/default.cll. You only need to specify those settings, which you want to change and you must not put a `#class` DEFAULTS around it anymore. E.g. if you would like to make all activities integer activities by default and set the default priority to 100, you're DEFAULTS section would look like this:

```
[DEFAULTS]
#activity activity {
   integ #= 1
   priority #= 100
}
```

   If you do not want to specify any customary default settings, you can omit the whole section.

5. The [MODEL] section contains the rest of the .cll file as it was before, however there have been some major changes to the syntax, and you will need to adapt your code:

6. Networkobjects are now called assets so change all `#networkobject` to `#asset`

7. For performance reasons `#thisinst`, `#thisby` and `#thisfor[]` have been replaced by a generic `#this()` function. So please change:

   (a) `#thisinst` → `#this(instance)`

   (b) `#thisby` → `#this(by,0)`

   (c) `#thisfor[1]` → `#this(field,1)`

   (You can now also refer to parts of the by specification that resulted from `#tables` or `#foreach`, if you choose a number higher than 0, zero, gives you the whole construct. You can now also refer to the full field name by using `#this(field, 0)`.

8. The `#this()` function is a function and not a local variable anymore, so it cannot be included into strings. So, something like

```
"Growing #thisinst on soil #thisby"
```

   will have to become

```
"Growing " ~ #this(instance) ~ " on soil " ~ #this(by,0)
```

9. A small change: the format of the result group feature has now been made consistent with other subelements, and uses a > ate the end

   So instead of

```
#resultgroup crop #=   #table(process,2,[#this(instance)])
```

   you now have to write:

```
#resultgroup> crop #=  #table(process,2,[#this(instance)])
```

10. In case you knew and used the sign to get debugging output, this has now become obsolete. It is replaced by a new `#report()` function.

    The syntax is

```
#report(id, condition, arguments to report, ...)
```

    id is an id you can freely choose, it is meant to avoid confusion if you use several reports

    condition report is only written to screen if condition is true so you can e.g. restrict it to certain buggy instances

The report function returns all except the first two arguments to the expression, so the actual calculation is not affected, even if you put it inside an expression.

Examples:

```
orow #=    #report(#this(instance),1, -1 * #table(prices,
    2,[#this(instance)]))
```

→ prints the result for the field orow for every instance of the class on the screen and each report is identified by the instance id

```
orow  #=    #report(test,#this(instance) == 24, -1 *
   #table(prices, 2,[#this(instance)]))
```

→ prints the result for the field orow for instance 24 onto the screen and the report is identified as test

```
orow #=    -1 * #report(test,#this(instance) == 24,
    #table(prices, 2,[#this(instance)]))
```

→ prints the value read from the table for instance 24 onto the screen and the report is identified as test

D.3 Changes in 2.03

- Corrections for perennials and livestock
- Introduction of OSL mode 4 (added necessary entries to model configuration)
- `mpmasql`now creates stata import scripts automatically (added necessary entries to control file)
- Implemented interface for `-P` flag to choose map locations

D.4 Changes in 2.04

- Special country model switch Germany (5) with new asset field `available_until_period`.
- bug fixes livestock and perennials
- implemented automatic creation of R import scripts
- renamed THIRDMILP to HARVEST_DECISION
- revised external crop growth interface

D.5 Changes in 2.08

- efficiency improvements, bug fixes and error checking mpmasdist, livestock, manual

D.6 Changes in 2.10

- change matrix coefficients over time (#exchange>)
- example models
- error correction in tutorial model in manual
- bug fixing

D.7 Changes in 2.12

- New naming convention MpmasExe -> mpmas

D.8 Changes in 2.14

- bug fixes
- `--runonly` option

D.9 Changes in 2.16

- bug fix: livestock model, replaced EE sign for total endowment by LE

D.10 Changes in 2.19

- bug fixes: mostly related to advanced consumption model and R scripts
- introduced artificial maps and cluster, etc. maps into mpmasdist

D.11 Changes in 2.20

- bug fixes: mostly related to advanced consumption model, livestock, and R scripts
- country model Germany: min/max age and inflation rate correction for assets at simulation start
- Result handling: partitioning of TGM into user-defined accounts
- Revision of advanced consumption model:
  - removed disinvestment and extra-consumption related activities and constraints from consumption class
  - add energy deficit constraints, deficit transfer and deficit penalty, including table utility_penalty
- updated R import to new p-File output
- advanced demography model

D.12 Changes in 2.21

- investment horizon component for advanced demography model
- income balance for advanced demography model

D.13 Changes in 2.22

- applied corrections for advanced consumption model
  - revised MASCONSUMPTION and MASSTANDARD classes
  - new user variables needed (see Section 4.4.2).

D.14 Changes in 2.23

- implemented change of coefficients between production and consumption LPs (“fine tuning parameters”)

D.15 Changes in 2.24

- implemented the several markets feature

D.16 Changes in 2.26

- bug fixes

D.17 Changes in 2.27

- bug fix in mpmasdist
- Advanced consumption model:
  - MASTANDARD activity stcredit_taken , set consfix to 1
  - MASTANDARD activity stcredit_defaulted , relaxing of stcredit_balance only in #harvest>
  - MASCONSUMPTION constraint xc_fnf_funcc set eqtype = 3

D.18 Changes in 2.28

- introduced the STANDARD_LAND_RENTS table.

D.19 Changes in 2.32

- introduced quadratic programming problems, plus bug fixes

D.20 Changes in 2.33

- compatibility for mpmas 3.3.194

D.21 Changes in 2.36

- compatibility for mpmas 3.3.204, mostly changes to Germany model and bug fixes quadratic programming
- extended mqlmatrix, improved quadratic programming interface

D.22 Changes in 2.39

- implemented BIOVERSION 4
- sorting of LandUseActivityIDs by LP Column
- Debugging output for Expert-N coupling

D.23 Changes in 2.40

- implemented producer organizations
- bug corrections (multiple markets)

D.24 Changes in 2.41

- implemented user variables in the scenario list and definition of the .ini file and –set flag for mpmasql

D.25 Changes in 2.47

- bug fixes
- activated exogenous yield expectations for country switch Peru
- added feature to add agent attributes to matrix (only country switch Germany)

D.26 Changes in 2.48

- bug fixes
- landscape cell attributes to matrix
- custom NRU grouping of soils
- disinvestment of assets (except perennials, only Germany)
- household member attribute

D.27 Changes in 2.52

- bug fixes
- land market parcel group version
- gross investments in p-File