LUTO II (neoLUTO)

Fjalar J de Haan[*]

Planet A, Centre for Integrative Ecology, Deakin University, Australia

Abstract

This document provides an overview of the LUTO II modelling software package.

Contents

1 Introduction **2**

2 Architecture of the model **3**

2.1 Loading data (`luto.data`) 4

2.2 Economic calculations (`luto.economics`) 7

2.2.1 Production costs (`luto.economics.cost`) 8

2.2.2 Transition costs (`luto.economics.transitions`) . . . 8

2.2.3 Yields (`luto.economics.quantity`) 9

2.3 Solving and running (`luto.solvers`) 10

2.3.1 Spatial coarse graining and spatial sub-setting 10

3 Optimisation mathematics **12**

4 Data requirements and preparation **12**

5 Running the model **12**

[*]Corresponding author. Email: f.dehaan@deakin.edu.au, fjalar@fjalar.org

1 Introduction

This document provides an overview of the software package of LUTO II. It describes its architecture, the mathematical model behind the cost-minimising dynamics, the data required and how to build them from the raw datasets. It also gives instructions on how to set up and run the model.

The LUTO II model simulates land-use change. The model is spatially explicit and the modelled territory is discretised into 1×1 km^2 (approximately) grid cells. These grid cells are represented in the model as 1D arrays, with the array entry representing the land use. A *land use* is, as the name suggests, the way land is used, which typically means what kind of crop is grown or what type of livestock grazes on it. Associated with the land use are production costs and yields, both annual, which allow the model to decide which land use is optimal for a cell. There is a list of land uses considered and these are represented by integer values (e.g. 'Apples' = 1, 'Dairy' = 7). Thus, at the heart of it, land-use change is modelled by a changing integer array. This integer array is called the land-use map or *lumap* for short.

In addition to the land use, the model also takes into account *land management*. Apples, for example, can be cultivated with (or without) irrigation, using organic methods (or not) and so on as well as combinations of such land managements. Depending on the land management, there will typically be different production costs and different yields associated with the land use. Irrigation is more expensive but may be expected to increase yields. There is a list of land managements and these are, like land uses, represented by integer values. Conventional, dry-land agriculture is the default and represented by zero, while irrigation is represented by one. The model currently only considers conventional dry land and irrigation as land managements. Thus there is also a land-management map, or *lmmap* for short.

The *lumap* and *lmmap* are the dependent variables of the model. Everything else is calculated on the basis of these two 1D arrays. The mechanisms driving the dynamics are economic. In the original CSIRO LUTO, the economic rationale was profit maximisation, i.e. farmers were supposed to cultivate whatever would profit them most (subject to some risk-based thresholds). In LUTO II the economic rationale is more systemic. The idea is that the agricultural system tries to meet demands (inputs to the model) the best it can, subject to certain constraints, while trying to minimise the total cost of production. Thus, using a yearly time step, the model is fed new demands and the model's solver produces a new land-use map.

Minimising cost while meeting demand means the model is trading off the expenditure of production against the yield of the crop or livestock. In addition to this, there are *transition costs* associated with switching from one land use to another. This means that if a grid cell changes from growing apples to raising cattle, there is not only the new production cost to be paid (which may be lower) but also a transition cost. These transition costs subsume various costs of switching (including infrastructural investments and changed irrigation costs). Transition costs introduce 'memory' into the model, avoiding that the land-use map changes unrealistically much at each time step as it attempts to meet demand at lowest cost.

LUTO II, like its progenitor, is an optimisation model. The economic rationale is formalised as a linear programme, which is then solved using external, commercial, black-box, closed-source solver software (GUROBI for LUTO II and CPLEX for LUTO I). The mathematical model of LUTO II is an actual linear programme but an alternative solver prototype using binary decision variables is also part of the package.

The model is solved under various constraints. One constraint ensures that all of every cell is in use. That is, there is always *some* land use on a cell and all of it is used. In principle, a cell can multiple 'fractional' land uses, though in practice, if the territory consists of many cells this seems not to occur. Another constraint is that the deviation of production from the demanded quantities should be minimal. This is a so-called soft constraint. These two constraints are an essential part of the model formulation. The remaining constraints are concerned with environmental targets and they can be switched on or off. Of these, water use is the first considered and the only one currently implemented. The water constraint demands that current water use relative to the water yield (how much runs of) in a river region should not exceed the water use in 2010 relative to pre-European yields (which are estimated using 1985 data and assuming deeply-rooted vegetation). Water is implemented as a hard constraint.

2 Architecture of the model

The LUTO II software is contained in a Python 3 package. The package structure (see Figure 1) is meant to reflect the different stages of the modelling process. The various sub packages live under a main package called `luto`. Thus there is a sub package concerned with the loading of data (`luto.data`), one with cost calculations (`luto.economics`) and so on. Each sub package typically contains several modules, each with several functions.

A small number of design principles has generally been adhered to, though not into the extreme. The most important were:

- Avoiding global variables and avoiding statefullness. The only exception to this being the `luto.solvers.simulation` module which has the express purpose of keeping the state of a simulation.
- Avoiding object orientation where possible. The notable exception is again `luto.solvers.simulation` which uses a `class Data`.
- Functions should be *pure*. This means that a function should return something, it should always return the same thing if given the same arguments and it should do nothing else ('no side effects'). Deviations from this principle are typically explicit (e.g. setters and functions to write to file).
- Dependencies should be few. Generally, the external dependencies are limited to Numpy and Pandas (almost everything uses these) and things like HDF5 libraries. Dependencies on other LUTO modules are used where unavoidable, e.g. in a module where many things come together. Again, in `luto.solvers.simulation` there are of course plenty of internal imports.

The user interface of the model is the `luto.solvers.simulation` module. Importing this module will implicitly load the `data` module. It will use the parameters set in the `luto.settings` module. At this point, those parameters are limited to directory locations and some settings relating to the water constraints. If all the requisite data is available where it should be (`input/` by default), this is sufficient to run the model. One can run 'interactively', that is, from the prompt of a Python interpreter or call the functions of the `luto.solvers.simulation` module from a run script. Preparing the input data and setting up the model to run is discussed in Sections 4 and 5, respectively.

2.1 Loading data (`luto.data`)

The main idea behind the `luto.data` module is to have a namespace for all input data and parameters. For example, importing the module, by issuing

```
import luto.data as data
```

will make the list of land uses available as `data.LANDUSES`. Data is made available in three ways, (1) by reading the data directly from a file, typically

```
/
├── luto/ ................................ main sub-package, i.e. ‘src’
│   ├── data/ ...................................... loading of inputs
│   │   ├── __init__.py .......................... actual data module
│   │   ├── economic.py .................... data preparation functions
│   │   ├── spatial.py .................................. legacy code
│   │   └── temporal.py ................................. legacy code
│   ├── economics/ ................ costs, yields and water calculations
│   │   ├── cost.py ........................ production cost functions
│   │   ├── quantity.py .............................. yield functions
│   │   ├── transitions.py .................. transition cost functions
│   │   └── water.py .................... water use and yield functions
│   ├── reporting/ ..................... derivative output calculations
│   ├── solvers/ ................ functions and interface to run model
│   │   ├── resfactor.py .......... functions for spatial coarse graining
│   │   ├── simulation.py ..... interface module for running simulations
│   │   ├── solvers.py ................ solver function calling GUROBI
│   │   └── stacksolver.py ........ alternative solver (toy model stage)
│   ├── tools/ ...................................... utility functions
│   │   ├── __init__.py ...................... various utility functions
│   │   ├── compmap.py ...................... cross-tabulation functions
│   │   ├── highposgtiff.py ................ to write lumaps to geotiffs
│   │   ├── plotmap.py ................. lumap viewer using matplotlib
│   │   └── write.py ........... write simulation output to files in a dir
│   └── settings.py ......... settings like file locations and parameters
├── input/ ......................... *.npy, *.csv and *.hdf5 data files
├── output/ .................. model results, lumaps and lmmaps etc.
└── docs/ ........................ documentation, including this file
```

Figure 1: Directory structure of LUTO II Python package.

a *.npy, *.csv or *.hdf5 file, by (2) post-processing data read from file to some extent, or (3) by defining variables in the code directly (i.e. 'magic numbers'). This third option is obviously to be avoided and the only magic number is the starting year, 2010, which is added to the year index (zero-based) whenever an actual calendar year is required.

The files required by the `luto.data` module are:

- ...

The data in the files is used to *infer* various parameters of the modelling problem. In other words, in principle, the code knows nothing about the particulars of the modelling problem under consideration. Things like the spatial extent (number of grid cells), the lists of land uses and land managements and so on, the model obtains from inspecting the data files. This has the obvious advantage that the model is not tied to a certain number of cells or a particular set of land uses.

There are complications to this picture. The most notable being the following. The list of *land uses* (whatever lives on the land) is not the same as, and not in a one-one relation with, the list of *products* those land uses yield. For example, 'Sheep - natural land' (a land use) yields 'SHEEP - NATURAL LAND MEAT' as well as 'SHEEP - NATURAL LAND WOOL'. To complicate matters further, the list of *commodities* that are demanded (the time series of which are a key input to the model) are not the same nor in a one-one relationshep with either the list of land uses or products. For example, the demand for wool is indifferent to whether the sheep grazed on natural or modified land. So, in the list of products there is just 'sheep wool'.

To avoid confusion and force useful errors, the three lists (`data.LANDUSES`, `data.PRODUCTS`, `data.COMMODITIES`) use different cases (sentence, all upper and all lower case, respectively). Nevertheless, the lists are produced by inferring the list of land-uses from the agricultural data and a host of string manipulations. This is clearly error prone in light of future extensions. Moreover, the *conversions* from e.g. land-use to product representation, which are necessary to set up the linear programme in the solver, employ conversion matrices. These matrices are defined in the `luto.data` module following the above logic.

The situation with the land-uses, products and commodities makes the `luto.data` module cluttered, hard to read and a potential source of future bugs. The schema can likely be simplified to just two lists and some processing could be factored out while other things can be turned into file reads. At the moment however, the module works and once loaded the complications are at any rate not so visible.

Since the model avoids global dependencies, the `luto.data` loaded module needs to be explicitly passed as an argument to all functions that need it. While this may appear as complicating things unnecessarily, there are two key advantages to this approach:

1. Many functions would take very long to import if they had the `luto.data` module as a hard-coded dependency. Those functions are now fairly light weight.

2. One can also pass a *different* data object to those functions. For example, some small dummy object for testing purposes. The `luto.simulation` module makes extensive use of this possibility by creating derivative data objects with many grid cells masked out to save computational space and time (see Section 2.3).

The `luto.data` sub-package has several modules but only the `/__init__.py` is used to load it. The modules `luto.data.spatial` and `luto.data.temporal` are legacy code. The `luto.data.economic` module, however, is used in the data preparation (see Section 4).

2.2 Economic calculations (`luto.economics`)

The sub-package `luto.economics` is concerned with calculating the production costs, transition costs and yields related to the land uses. This sub-package also contains a module to calculate water uses and yields. All functions in these modules are pure and most require a data object as an argument.

The calculations in this sub-package are typically returned in a matrix format, including 'matrices' with more than two indices. This is partially because it is the natural format from the perspective of the mathematical formulation and GUROBI solver API. It also enables the use of some linear algebra tools making for compacter code. This at the expense of a lot of matrices of various shapes in the model. To keep overview and avoid errors, the following convention were used: A matrix data type is denoted by a letter followed by an underscore and a list of indices, e.g. `c_mrj`. The letter relates to the quantity in question (here 'cost') and the indices run over various ranges, depending on what the axis represents. See Table 1 for the constantly recurring indices and the sets they enumerate.

Index	Set	Range	Set in LUTO II	Range in LUTO II
r	Grid cells	7×10^7		`data.NCELLS`
j	Land uses	28	`data.LANDUSES`	`data.NLUS`
m	Land man's	2	`data.LANDMANS`	`data.NLMS`
p	Products	32	`data.PRODUCTS`	`data.NPRS`
c	Commodities	26	`data.COMMODITIES`	`data.NCMS`

Table 1: Common indices for matrix-like quantities.

2.2.1 Production costs (`luto.economics.cost`)

A major part of the optimisation logic are the annual production costs, i.e. what it costs to have a certain crop or livestock on a particular grid cell. These costs depend not only on the land use, but also on spatial location as well as the land-management type (whether it is irrigated or not) and the year. The `cost` module provides several functions to calculate these annual production costs. Of these, `get_cost_matrices()` provides the costs in the matrix format used by the solver. The actual cost calculations are done in separate functions (`get_cost_crop()` and `get_cost_lvstk()`). If one wants just the production costs of a certain land use and land management combination in a certain year, there is the function `get_cost(data, lu, lm, year)`, which returns a 1D spatial array with the cost per grid cell.

Production costs of crops are computed as *fixed* plus *area* plus *quantity* costs. The fixed costs in turn are the sum of fixed *labour*, *operational* and *depreciation* costs. These component costs come from data and only few operations are involved in the computation of the total production cost. Additionally, if it concerns an irrigated crop, there are water costs which are the product of the volume of water required and the delivery costs.

A similar breakdown holds for livestock production costs, with some amendments. The quantity costs in the data are now given per head so multiplication by a 'yield potential' (number of heads per hectare) is necessary. In the water costs, now drinking water also needs to be included (regardless of whether the pasture is irrigated or not).

2.2.2 Transition costs (`luto.economics.transitions`)

The transition costs are the costs of changing the land-use or land-management on a certain cell. The key function in this module is `get_transition_matrices()` which returns the transition matrices in a t_{mrj} format. Thus the entry $t_{1,3,7}$ is the cost of switching cell number 3 (which is the fourth cell, arrays are zero based) to land use 7 ('Grapes') under land management 1 ('irr', i.e. irrigation). The knowledge about what the land-use and land-management it is switching *from* is subsumed in the t_{mrj} matrix, so naturally the current lumap and lmmap are arguments to the `get_transition_matrices()` function.

The actual transition costs are computed from a number of parts:

1. 'Raw' transition costs from one land use to another
2. The difference in water licence costs after and before
3. Irrigation infrastructure costs, if applicable
4. Foregone income at 3 times the annual production costs

Finally, the costs are amortised at 5% with a horizon of 30 years.

2.2.3 Yields (`luto.economics.quantity`)

Yields refer to the quantities of *product* that can be obtained from a land use on a certain grid cell. While a land use has only one associated production cost, it can have several associated yields — for example, as mentioned before, a land use like 'Sheep - natural land' yields 'SHEEP - NATURAL LAND MEAT' as well as 'SHEEP - NATURAL LAND WOOL'.

The functions in the `luto.economics.quantity` module are parallel to their counterparts in the `costs` module — with the crucial distinction that costs are for land uses, while yields pertain to products. Thus the module features the following functions. `get_quantity_matrices()` provides the yields in the matrix format used by the solver. The actual yield calculations are done in separate functions (`get_quantity_crop()` and `get_quantity_lvstk()`). If one wants just the yields of a certain product and land management combination in a certain year, there is the function `get_quantity(data, pr, lm, year)`, which returns a 1D spatial array with the cost per grid cell.

The yields for crops come directly from the input data and are only converted from per hectare to per cell units (in `get_quantity_crop()`). Also, for crops the correspondence between land uses and products is one-one. The calculations for livestock (in `get_quantity_lvstk()`) yields are a bit more involved and will be discussed below. The function `get_quantity()`, which branches to the before mentioned functions, additionally applies two multipliers, (1) the climate change impacts (via `get_ccimpact()`, interpolating input data) and (2) yield increases, i.e. productivity increases (directly from data).

The function `get_quantity_lvstk()` first infers which livestock and vegetation type are involved by calling the function `lvs_veg_types()` which determines this based on the product string. So, 'SHEEP - NATURAL LAND WOOL' corresponds to livestock type 'SHEEP' and vegetation type 'NATL' (i.e. natural). This information is then used to calculate a *yield potential* using the function `get_yield_pot()` which effectively is how many heads per hectare can be yielded.

The livestock yield data holds both the total herd size and the fraction available for use (e.g. slaughter or export). The product of these to factors gives a per-head, per-hectare yield. Thus `get_quantity_lvstk()` multiplies this by the yield potential and converts from per-hectare to per-cell units.

2.3 Solving and running (`luto.solvers`)

The heart of the model, that is, where the next land use map is actually computed is the `solve()` function — 'the solver' — in the `luto.solvers.solver` module. This function takes all relevant input matrices, the demands, constraints and two conversion matrices as arguments and returns the new land-use and land-man' maps. Since this function solves the system for one time step at a time and a considerable list of arguments is required, there is an interface to solving the model provided in `luto.solvers.simulation`. The `simulation` module takes care of computing the input matrices, keeps track of the solved lumaps and lmmaps and provides a number of convenience functions. Section 5 describes in some detail how to use the `simulation` module.

2.3.1 Spatial coarse graining and spatial sub-setting

Resfactor (lossy)

To facilitate faster testing, one can run the model in a 'coarse-grained' fashion. The idea is simply to present the solver with a sampled subset of the spatial domain and correspondingly sliced matrices. This facility is called 'resfactor' after a similar functionality in LUTO I. The sampling can be either linear or quadratic, which means that if a sampling rate of n is given, either every n^{th} or every $n \times n^{\text{th}}$ cell is included. The linear mode produces sampling artefacts, so the quadratic mode should always be used. The sampling is implemented with a multiplicative mask. The following three functions are involved in recovering the maps from the coarse-grained solver output: `simulation.uncourse1D()`, `simulation.uncourse2D()` and `simulation.uncoursify()`.

Thought the resfactor can be set manually with the `simulation.set_resfactor()` function, for consistent settings it is recommended to instead pass the resfactor as an argument to `simulation.run()`.

The speed up is slightly better than proportional to the fraction of cells sampled. It should be superfluous to note that solving a coarse-grained map and then resizing the outputs does not provide the same results as an actual solve on the full map. Simulation results obtained with spatial coarse graining are to be used for quick inspections and sense checking in testing contexts only.

Subsetting (lossless)

Since the grid cells with land use 'Non-agricultural land' are not changed by the solver, there is no loss in solving the lumap with these cells excluded and reinserting them afterwards. This is precisely what is done when the solving is done using the `luto.solvers.simulation` interface. Like the spatial coarse graining, the sub-setting of the input map is achieved with a multiplicative map. In fact, when coarse graining is on, the two masks are indeed multiplied. The removed cells are re-inserted post-solve by the `simulation.reconstitute()` function. The speed up is proportional to the fraction of 'Non-agricultural land' cells in the lumap and there is no loss of accuracy as there is no approximation.

Not only the lumap and lmmap are spatially explicit and therefore amenable to this sort of sub-setting or coarse-graining. In fact, many of the input data are spatially explicit and calculations could be sped up considerably if e.g. costs and yields were computed on the same fraction of the cells as the lumap and lmmap. The functions in the `luto.economics` module are agnostic to the spatial extent and pure, so if they were passed a data object on a different, smaller, spatial domain, they would not mind.

The `class Data`

To make use of this, the `luto.solvers.simulation` module contains a class definition for a data object that can be spatially subsetted. This is a simple class with no methods. All it aims to do is to expose the same fields (at any rate the upper-case fields) as the `luto.data` module but defined on the masked spatial domain. To achieve this, it copies all the upper-case fields and applies the combined (multiplied) masks of resfactor and the lossless subsetting described earlier to the spatial fields.

// Local matrix getters.

The `simulation.step()` *function*

To call the solver, the `luto.simulation` module uses the `simulation.step()` function.

Since the functions for e.g. cost calculations, receive a `data` object as an argument Extensive use of this is being made by the `luto.simulation` module

3 Optimisation mathematics

4 Data requirements and preparation

5 Running the model