

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 (<code>luto.data</code>)	4
2.2	Economic calculations (<code>luto.economics</code>)	7
2.2.1	Production costs (<code>luto.economics.cost</code>)	8
2.2.2	Transition costs (<code>luto.economics.transitions</code>)	8
2.2.3	Yields (<code>luto.economics.quantity</code>)	9
2.2.4	Production costs	9
2.2.5	Transition costs	9
2.3	Solving and running	9
3	Optimisation mathematics	9
4	Data requirements and preparation	9
5	Running the model	9

*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 $1 \times 1 \text{ km}^2$ (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 *lmmmap* for short.

The *lumap* and *lmmmap* 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 ex-

penditure 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 off) 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 statefulness. 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

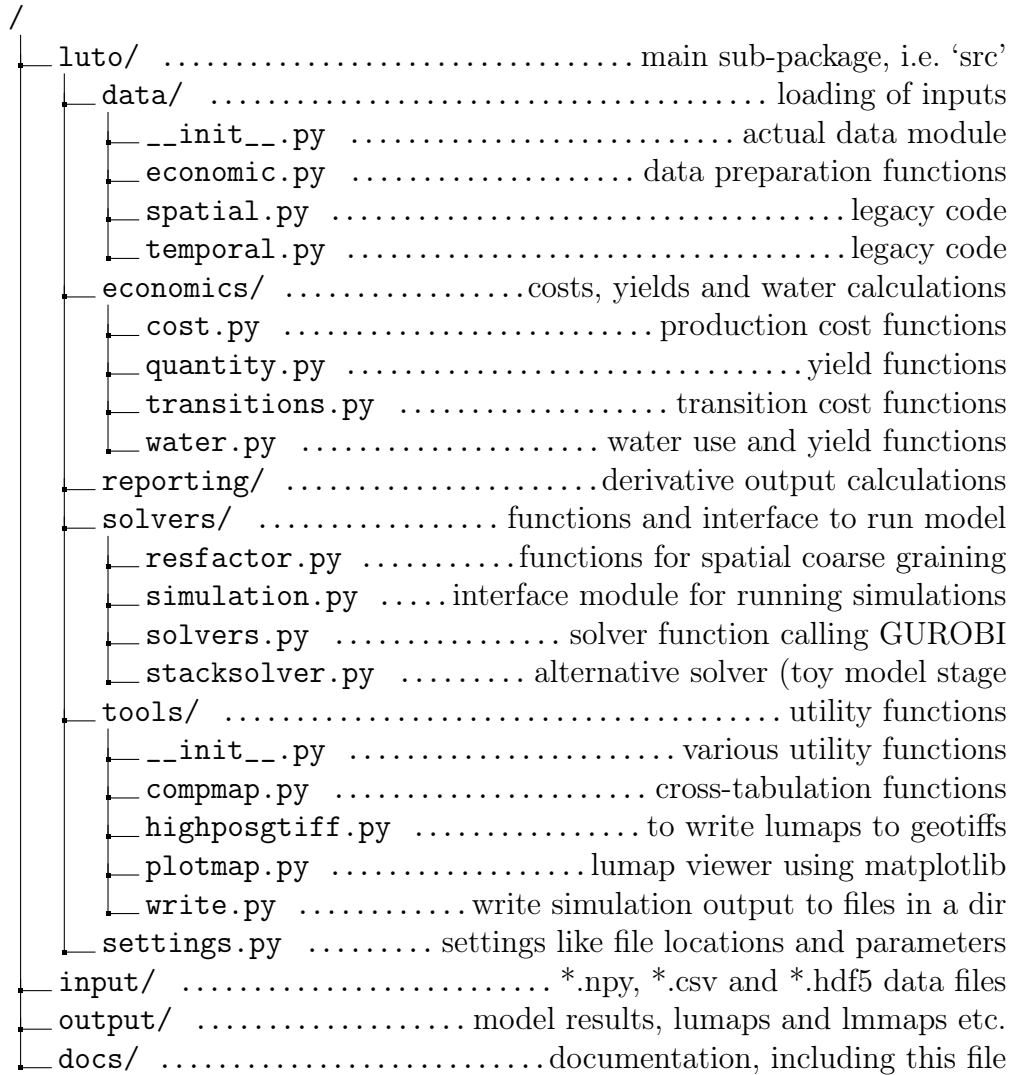


Figure 1: Directory structure of LUTO II Python package.

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 relationship 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		<code>data.NCELLS</code>
j	Land uses	28	<code>data.LANDUSES</code>	<code>data.NLUS</code>
m	Land man's	2	<code>data.LANDMANS</code>	<code>data.NLMS</code>
p	Products	32	<code>data.PRODUCTS</code>	<code>data.NPRS</code>
c	Commodities	26	<code>data.COMMODITIES</code>	<code>data.NCMS</code>

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 `lmap` and `lmap` 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`)

...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...

2.2.4 Production costs

2.2.5 Transition costs

2.3 Solving and running

3 Optimisation mathematics

4 Data requirements and preparation

5 Running the model