Persefone.jl User Manual

Daniel Vedder, Marco C. Matthies, Guy Pe'er

Contents

Part I

Introduction

Figure 0.1: Persefone.jl splash screen

Persefone.jl models agricultural practice and how it impacts animal species at a landscape scale. It includes a farm submodel, a crop growth submodel, and individual-based models of multiple indicator species. Its aim is to investigate how changes in farm operations (e.g. through policy changes in the CAP) influence biodiversity.

The model is open-source software available on Gitlab.

This documentation was last updated on 2025-01-15 for **Persefone.jl v0.6.0** *(commit f324167).*

Part II

User guide

Chapter 1

The Persefone.jl Package

This page describes how to run Persefone.jl as a command line application or Julia package, which is the default mode. To use the model with a graphical user interface, see here.

1.1 Installation

For more detailed installation instructions, see here.

Install the latest version of the Julia programming language (1.10+). The recommended editors are VSCode or Emacs. To install the package dependencies, open a Julia REPL in this folder and run:

```julia
using Pkg
Pkg.activate(".")
Pkg.instantiate()
```

1.2 Running from the command line

This is the normal mode of operation. Simply execute `run.jl` in a terminal, typically like so (in Linux):

```
> julia run.jl -c <config>
```

where `<config>` specifies the configuration file to use. The recommended workflow is to copy `scr/parameters.toml` to a location of your choice and edit the copy to suit your requirements. The adapted config file can then be passed to `run.jl`. (If no configuration file is specified, Persefone will run with its default settings.)

The full list of commandline arguments is:

```
usage: run.jl [-c CONFIGFILE] [-s SEED] [-o OUTDIR] [-l LOGLEVEL]
              [--version] [-h]

optional arguments:
  -c, --configfile CONFIGFILE
                        name of the configuration file
  -s, --seed SEED       inital random seed (type: Int64)
  -o, --outdir OUTDIR   location of the output directory
  -l, --loglevel LOGLEVEL
                        verbosity: "debug", "info", or "quiet"
```

```
  --version              show version information and exit
  -h, --help             show this help message and exit
```

To run the test suite, switch to the `test` directory and execute `runtests.jl`.

If you are on Linux or MacOS, you can also use make:

```
> make run       # run a simulation with default values
> make test      # run the test suite
> make profile   # run and profile a default simulation
> make docs      # build the documentation
> make release   # create a release
```

1.3   Running from within Julia

To use the model from within Julia (either inside an interactive REPL or if you want to import it from your own software), do the following:

```
using Pkg
Pkg.activate(".") # assuming you're in the Persefone root folder
using Persefone
```

You can then access all Persefone functions, such as simulate, initialise, stepsimulation!, simulate!, or visualiseoutput. (See `src/Persefone.jl` for a list of exported functions.)

Chapter 2

Graphical User Interface

Due to the computational demands of simulating many individuals at high temporal and spatial resolution, Persefone.jl is primarily designed to be run non-interactively on an HPC. However, to allow interactive exploratory simulations to be conducted while learning or developing the model, a graphical user interface is available as an additional package: Persefone Desktop.

2.1 Quick start

Follow these instructions if you simply want to try out the software as a user. If you want to play around with the source code, see the next section.

1. Download the Julia programming language and install it on

your computer.

1. Start Julia. This should launch a commandline interface/REPL.

2. Execute the following commands (copy-and-paste should work):

```julia
using Pkg
Pkg.add(url="https://git.idiv.de/persefone/persefone-model.git")
Pkg.add(url="https://git.idiv.de/persefone/persefone-desktop.git")
using PersefoneDesktop
ENV["QSG_RENDER_LOOP"] = "basic" # only needed on Windows
launch()
```

2.2 Running from the repo

Follow these instructions if you want to get to grips with the source code. For more detailed installation instructions, see here.

To install: Install Julia and download/clone the repository. Open a Julia REPL in the downloaded folder and execute the following to install all dependencies:
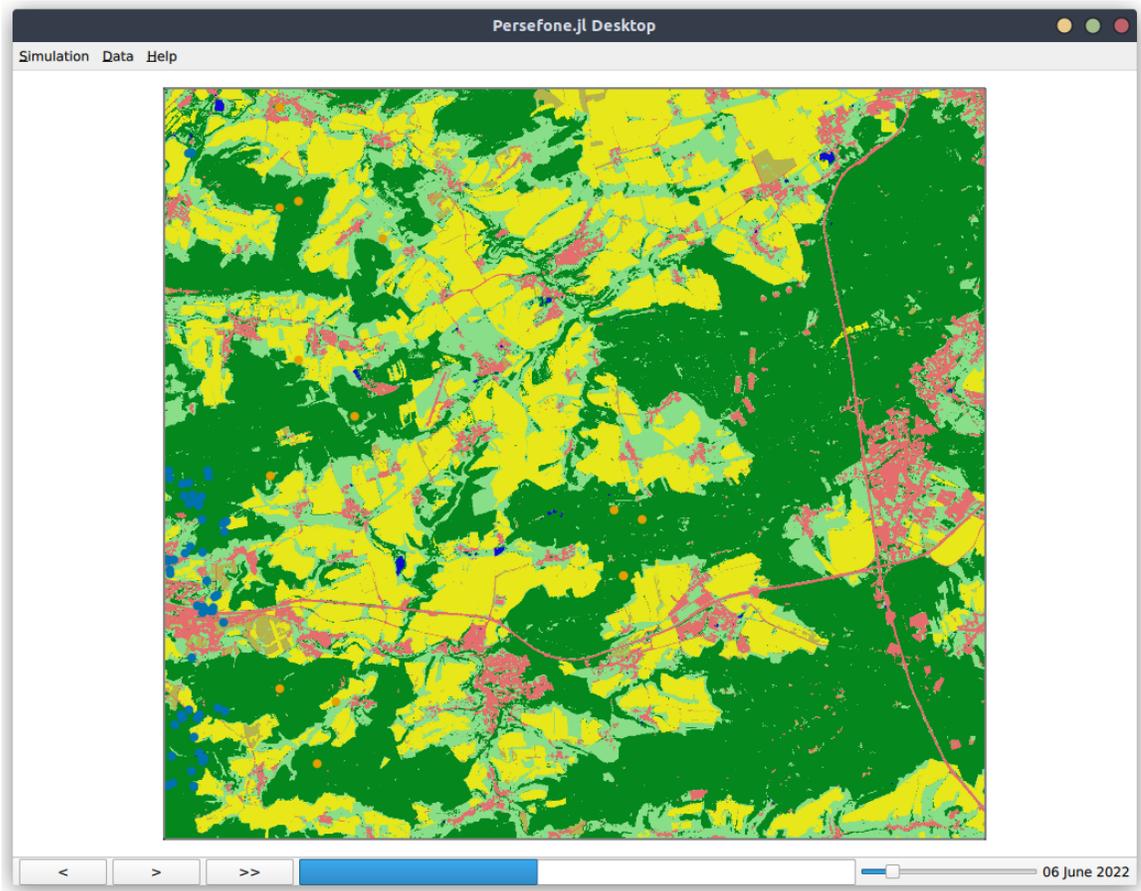
Figure 2.1: Persefone.jl Desktop screenshot

```
using Pkg
Pkg.activate(".")
Pkg.instantiate()
```

To run: Run desktop.jl. Alternatively, open a Julia REPL in this folder and run:

```
using Pkg
Pkg.activate(".")
using PersefoneDesktop
launch()
```

Note: Due to the necessary pre-compilation done by Julia, installing and launching the application can take quite a long time. (Start-up time with desktop.jl is currently about 2 minutes.) We will reduce this as much as possible in future releases.

2.3   User interface

The main window component is the **map view**. This displays a land cover map of the simulated region: dark green are forests, light green grassland, yellow fields, red built-up areas and blue water. On it, little circles show the position of individual animals, with different species denoted by different colours.

Control bar

- **Back button:** Rewind the simulation by one day.

- **Step button:** Advance the simulation by one day.

- **Run button:** Run the simulation until the button is pressed again or the end date is reached.

- **Progress bar:** Shows the percentage of time elapsed between the start and end dates of the simulation.

- **Speed slider:** Set the time delay between each simulation step when running.

- **Date:** Shows the simulation date currently displayed on the map.

Menu bar

Simulation:

- **New simulation:** Reset the model and start over.

- **Configure simulation:** Change the model settings (*not yet implemented*).

- **Load saved state:** Load a model object file saved by a previous simulation run.

- **Save current state:** Save a model object file for later use.

- **Quit:** Close the application.

Data:

- **Show population graph:** Show a window with a graph of population sizes over time in the current model run.

- **Save simulation output:** Save the model output data to file (saves both raw CSV data and generated graphics).

Help:

- **Documentation:** Open the Persefone.jl online documentation in a browser.

- **Website:** Open the main Persefone.jl website in a browser.

- **About:** Show a window with core information about the application.

Chapter 3

Configuration

Persefone requires three input files: a configuration file and two map files. How to generate the map files is documented elsewhere. The configuration file defines parameter values and looks like this (see `src/parameters.toml` for the default):

```
### Persefone.jl - a model of agricultural landscapes and ecosystems in Europe.
###
### This is the default configuration file for Persefone, containing all model parameters.
### The syntax is described here: https://toml.io/en/

[core]
configfile = "src/parameters.toml" # location of the configuration file
outdir = "results" # location and name of the output folder
overwrite = "ask" # overwrite the output directory? (true/false/"ask")
logoutput = "both" # log output to screen/file/both
csvoutput = true # save collected data in CSV files
visualise = true # generate result graphs
storedata = true # keep collected data in memory
loglevel = "debug" # verbosity level: "debug", "info", "warn"
processors = 2 # number of processors to use on parallel runs
seed = 2 # seed value for the RNG (0 -> random value)
startdate = 2022-01-01 # first day of the simulation
enddate = 2022-12-31 # last day of the simulation

[world]
landcovermap = "data/regions/jena/landcover.tif" # location of the landcover map
farmfieldsmap = "data/regions/jena/fields.tif" # location of the field geometry map
weatherfile = "data/regions/jena/weather.csv" # location of the weather data file

[farm]
farmmodel = "FieldManager" # which version of the farm model to use (not yet implemented)

[nature]
targetspecies = ["Wolpertinger", "Wyvern"] # list of target species to simulate
popoutfreq = "daily" # output frequency population-level data, daily/monthly/yearly/end/never
indoutfreq = "end" # output frequency individual-level data, daily/monthly/yearly/end/never
insectmodel = ["season", "habitat", "pesticides", "weather"] # factors affecting insect growth

[crop]
cropmodel = "almass" # crop growth model to use, "almass" or "aquacrop"
cropfile = "data/crops/almass/crop_data_general.csv" # file with general crop parameters
```

```
growthfile = "data/crops/almass/almass_crop_growth_curves.csv" # file with crop growth
↪   parameters
```

> **Parameter scanning**
>
> You can set any parameter to a list of different values, e.g. `seed = [1,2,3]`. Persefone will then set up and run multiple simulations, one for every possible combination of parameters that you entered (i.e. do a full-factorial simulation experiment).

Part III

Scientific documentation

Chapter 4

Farm management

TODO

4.1 Crop rotations and management

4.2 Environmental regulations

Chapter 5

Crop models

TODO

5.1 ALMaSS

5.2 AquaCrop

Chapter 6

Skylark

Alauda arvensis is a common and charismatic species of agricultural landscapes. This animal model is one component of the nature submodel of Persefone.jl.

The model description follows the ODD (Overview, Design concepts, Details) protocol (Grimm et al., 2006; 2010; 2020):

6.1   1. Purpose

The purpose of this animal model is to simulate the abundance and distribution of a population of *Alauda arvensis* in response to farm management in Central European agricultural landscapes.

6.2   2. Entities, state variables, and scales

2.1 Landscape

The simulated landscape consists of a grid of pixels with a resolution of 10m and can have an extent of 20km²-200km² (approximately; depending on the chosen input map). Each pixel is assigned a land cover class. It may also be associated with a farm plot, in which case it will contain information about the type and growth stage of the crop planted here. Farm management determines which crops are grown when, and when disturbance (e.g. mowing, harvesting, tillage) takes place.

2.2 Animals

The simulated individuals (a.k.a. agents) are mature skylarks. Each skylark is characterised by the following variables:

- `ID` A unique identifier for this individual, which can be used to link it to its parents and its offspring.

- `sex` Male or female.

- `phase` The individual's current stage in the annual/life cycle. May be one of: `migration`, `nonbreeding`, `territorysearch`, `occupation`, `matesearch`, `nesting`, `breeding`.

- `position` The individual's position in the simulated landscape.

- `mate` The ID of the individual with which this individual has mated this year, if any.

- `territory` A list of coordinates of the positions in the landscape that this individual claims as its nesting and feeding territory.

- `nest` A coordinate giving the location of the currently active nest.

- `clutch` The number of juvenile (i.e. not yet independent) skylarks that this individual is currently raising.

6.3   3. Process overview and scheduling

The simulation proceeds in time steps of one day. Every day, each individual executes the function associated with their current life phase:

- `migration`: The individual is held in a separate data structure (apart from the model landscape) and does nothing until its return date is reached. Then, it is re-introduced to the landscape and assigned the phase `territorysearch` (for males) or `matesearch` (for females).

- `territorysearch`: Males return first from migration. If they already have a territory from a previous year, they return to this. Otherwise, they move randomly through the landscape until they find a contiguous territory that satisfies their habitat requirements. Once a male has a territory, it changes its phase to `occupation`.

- `matesearch`: Females return later than males from their winter migration. If they already had a partner the previous year, they have a given probability of remaining with this partner. Otherwise, they move randomly through the landscape, looking for a male with a territory and without a partner. Once the female has a partner, it changes its phase to `nesting`.

If an individual fails to find a territory or a mate, it changes its phase to `nonbreeding` once the breeding season is over.

- `occupation`: The male moves at random about its territory until the breeding season is over. Then it changes its phase to `nonbreeding`. (*Note:* Skylark males actively help with feeding their chicks. However, feeding is only modelled indirectly here, through the process of habitat selection when the male forms its territory - see section 4.1.)

- `nesting`: The female selects a suitable location within the male's territory for the nest. Building the nest and laying eggs takes a number of days, during which she does nothing else. Then, she changes her phase to `breeding`.

- `breeding`: The female checks for mortality. The probability of brood loss varies with the age of the clutch and the nesting habitat. If and when the chicks reach independence (30 days after hatching), they are instantiated as new individuals in the `nonbreeding` phase.

If a nest fails due to predation or disturbance, or a brood leaves the nest successfully, the female resets her phase to `nesting` and begins again if the breeding season is not yet over. If it is, she changes her phase to `nonbreeding`.

- `nonbreeding`: Non-breeding mature birds move randomly around the landscape, keeping close to other individuals (flocking behaviour). Once their individual migration date is reached, they are removed from the landscape until the following year (see above). Mature birds have a mortality probability for their first summer, and others thereafter for each winter.

6.4 4. Design concepts

4.1 Basic principles

This model assumes that the two most important drivers of skylark distribution and abundance are **habitat availability** and **juvenile mortality** (see literature below). The factors and processes affecting these are therefore given the most attention in the model, while other factors and processes are only included superficially, indirectly, or not at all. Specifically, this means that the phases `territorysearch`, `nesting`, and `breeding` are the most relevant and detailed parts of the model, as these determine the selection of habitat and the survival of offspring.

Furthermore, the model concentrates on predation and anthropogenic disturbance (through management actions such as mowing) as the main causes of juvenile mortality. Other causes, such as hunger or bad weather, are currently ignored as they are usually not significant.

The focus on habitat availability and juvenile mortality opens up two avenues by which agricultural management influences skylark populations. First, the farmers' choice of crops and date of sowing determines the quality of the habitat when skylarks select a territory. (For example, unlike summer grain, winter grain is already so high and dense in spring that it is generally avoided for nesting.) Secondly, the frequency and timing of management actions (especially mowing) is a major cause of brood loss. This means that there are direct causal links between agriculture and population trends.

Concentrating on these two drivers allows the rest of the model to be kept simple, reducing both the scientific complexity and computational costs. Thus, foraging movement (both during and after the breeding season) can be ignored or represented as random movement, as it does not directly impact either of the drivers. Likewise, chick growth and winter migration are represented very simply.

4.2 Emergence

Multiple patterns emerge from the basic principles outlined above. The most important are listed here:

- **Territory size and population density:** The model assumes that skylarks occupy only as much area as they need to satisfy their nesting and foraging requirements, and that population size is limited by the amount of available habitat. This means that territories in high-quality habitat are smaller than in low-quality habitat. Scaling up, this leads to a pattern whereby population densities are highest in open landscapes with a diversity of crops, grassland, semi-natural habitat, and lower in landscapes with low habitat diversity or many woody features.

- **Ecological traps:** Jenny (1990) describes a strong ecological trap effect whereby skylarks avoid winter grain crops, preferentially nesting in more open grassland sites. However, the mowing frequency associated with modern agriculture means that nest loss in grassland is almost assured, since there is insufficient time between two mowing dates to raise a brood. This means that landscape composition leads skylarks to breed in habitats that have a high mortality, resulting in population declines.

4.3 Adaptation

In the model, skylarks primarily adapt to their surroundings by choosing suitable territories. These are chosen by evaluating the quality of surrounding habitats for breeding and foraging, and occupying as much area as needed to satisfy requirements (see section 7.1).

4.4 Objectives

Skylarks' main objective in the model is to have sufficient habitat available to raise a brood. Habitat quality is calculated as a function of habitat type, vegetation height, vegetation cover, and distance to vertical structures (see section 7.1).

4.5 Learning

The model includes no learning by individuals.

4.6 Prediction

The model includes no predictions by individuals.

4.7 Sensing

Skylarks can perceive the landscape structure in a given radius around them (habitat type, vegetation height and cover). They can also see nearby conspecifics and are aware of the territories claimed by other individuals. When mating, they recognise whether another individual already has a mate, and mated individuals share information about their territory and brood status.

4.8 Interaction

The model includes two direct forms of interaction. First, during mating, females move around the landscape looking for males who have a territory but no mate yet. Once they have found one, the two individuals set each other as their mate. Secondly, after the breeding seasons, individuals move around the landscape, keeping close to other individuals in their vicinity (flocking behaviour).

There are also indirect interactions, in that there is a competition for habitat (territory that has been claimed by one male cannot be occupied by another) and males (males that have mated with one female will not mate with another in the same season).

4.9 Stochasticity

Stochasticity is used when modelling mortality and movement. Predation mortality is modelled as an age- and habitat-dependent probability, while migration mortality is a simple probability. Dispersal movement (when searching for a territory or a mate) is modelled as a random walk, as it is assumed that skylarks are not significantly impeded in their long-range movement by habitats that are unsuitable for foraging or nesting. Foraging movement by the male and by non-breeding individuals is also random, as it is desirable to show movement (to help model analysis) but unimportant to model this exactly.

4.10 Collectives

After the breeding season, skylarks move around in loose agglomerations (flocking behaviour). However, this has no relevant ecological effect.

4.11 Observation

TODO

6.5   5. Initialisation

At the beginning of a model run, pairs of skylarks are created on grassland and agricultural land, keeping a distance of 60m to vertical structures and allowing each pair approximately 3ha of suitable habitat (an average territory size in agricultural landscapes).

For details, see the source code and the associated documentation.

6.6 6. Input data

The general input to Persefone (i.e. land use maps and weather data) is described here.

The following extract from the source code lists the species parameters and values used by the Skylark model, based on the literature cited below:

```julia
@species Skylark begin
    const movementrange::Length = 500m #XXX arbitrary
    const visionrange::Length = 200m #XXX arbitrary

    const eggtime::Int64 = 11 # days from laying to hatching
    const nestlingtime::Int64 = 9 # days from hatching to leaving nest
    const fledglingtime::Int64 = 21 # days from leaving the nest to independence

    #XXX predation mortality should be habitat-dependent
    const eggpredationmortality::Float64 = 0.03 # per-day egg mortality from predation
    const nestlingpredationmortality::Float64 = 0.03 # per-day nestling mortality from predation
    const fledglingpredationmortality::Float64 = 0.01 # per-day fledgling mortality from predation
    const firstyearmortality::Float64 = 0.38 # total mortality in the first year after independence
    const migrationmortality::Float64 = 0.33 # chance of dying during the winter

    const minimumterritory = 5000m² # size of territory under ideal conditions
    const mindistancetoedge = 60m # minimum distance of habitat to vertical structures
    const maxforageheight = 50cm # maximum preferred vegetation height for foraging
    const maxforagecover = 0.7 # maximum preferred vegetation cover for foraging
    const nestingheight = (15cm, 25cm) # min and max preferred vegetation height for nesting
    const nestingcover = (0.2, 0.5) # min and max preferred vegetation cover for nesting

    const matefaithfulness = 0.5 # chance of a female retaining her previous partner
    const nestingbegin::Tuple{Int64,Int64} = (April, 10) # begin nesting in the middle of April
    const nestbuildingtime::UnitRange{Int64} = 4:5 # 4-5 days needed to build a nest (doubled for
    ↪   first nest)
    const eggsperclutch::UnitRange{Int64} = 2:5 # eggs laid per clutch
    const nestingend::Int64 = July # last month of nesting
end
```

6.7 7. Submodels

7.1 Territory formation

TODO

7.2 Juvenile mortality

TODO

6.8 8. References

- Bauer, H.-G., Bezzel, E., & Fiedler, W. (Eds.). (2012). Das Kompendium der Vögel Mitteleuropas: Ein umfassendes Handbuch zu Biologie, Gefährdung und Schutz (Einbändige Sonderausg. der 2., vollständig überarb. und erw. Aufl. 2005). AULA-Verlag

- Delius, J. D. (1965). A Population Study of Skylarks Alauda Arvensis. Ibis, 107(4), 466–492.

- Donald et al. (2002). Survival rates, causes of failure and productivity of Skylark Alauda arvensis nests on lowland farmland. Ibis, 144(4), 652–664.

- Glutz von Blotzheim, Urs N. (Ed.). (1985). Handbuch der Vögel Mitteleuropas. Bd. 10. Passeriformes (Teil 1) 1. Alaudidae - Hirundidae. AULA-Verlag, Wiesbaden. ISBN 3-89104-019-9

- Jenny, M. (1990). Territorialität und Brutbiologie der Feldlerche Alauda arvensis in einer intensiv genutzten Agrarlandschaft. Journal für Ornithologie, 131(3), 241–265.

- Jeromin, K. (2002). Zur Ernährungsökologie der Feldlerche (Alauda arvensis L. 1758) in der Reproduktionsphase [Doctoral thesis]. Christian-Albrechts-Universität zu Kiel.

- Püttmanns et al. (2022). Habitat use and foraging parameters of breeding Skylarks indicate no seasonal decrease in food availability in heterogeneous farmland. Ecology and Evolution, 12(1), e8461.

Part IV

Developer guide

Chapter 7

Developing Persefone

7.1 Setting up

If you haven't worked with Julia before, here are detailed instructions for how to set up your development environment. The main development is currently done on Linux (and as the primary execution platform will be an HPC, Linux compatibility is important), but developing on Windows works too.

Visual Studio Code on Windows

1. Download and install Julia, git and Visual Studio Code.

2. Install the Julia extension for VS Code: In VS Code, open the extensions pane (Ctrl+Shift+X). Search for and install Julia Language Support.

3. Clone the Gitlab repository: In VS Code, open the source control pane (Ctrl+Shift+G). Click on Clone and enter the repo URL. Then select a folder on your computer to download the files into, and let VS Code open the project once it has been cloned.

4. Start a Julia REPL: In VS Code, bring up the command palette (Ctrl+Shift+P). Execute the command Julia: Start REPL. Then install all dependencies of Persefone by running using Pkg; Pkg.activate("."); Pkg.instantiate(). (This will take some time.)

5. Open the file run.jl and click Execute (triangular button in the top right). The source code will compile (this can take a lot of time the first time you do it) and run a default simulation.

6. Further steps: You may want to familiarise yourself with how to use git with VS Code. You may also want to clone the Persefone Desktop repository (repeat steps 3 to 5).

Emacs on Linux

You can of course also use VS Code on Linux. In that case, follow the instructions above.

Make sure you have git and Julia installed. Git should be in your distro's repos (e.g. sudo apt install git). To install Julia, download the binary and unpack it. For greater ease of use, copy the unpacked files to /usr/local/lib/julia (or similar) and create a symlink to the executable: sudo ln -s /usr/local/lib/julia/bin/julia /usr/local/bin/julia. Then go the to folder that you want to use for development and run git clone https://git.idiv.de/persefone/persefone-model.git . in your terminal.

There are a couple of addons that make working with Julia much nicer in Emacs:

1. julia-mode gives syntax highlighting. Install with M-x package-install julia-mode.

2. `julia-snail` provides IDE-like features, especially a fully-functional REPL and the ability to evaluate code straight from inside a buffer. Note that the installation can be somewhat tricky. You first need to manually install all the dependencies of its dependency `vterm`, then install vterm itself with `M-x package-install vterm`, *before* you can do `M-x package-install julia-snail`. Then add it to your init.el with `(require 'julia-snail)` and `(add-hook 'julia-mode-hook #'julia-snail-mode)`.

3. `company-mode` integrates with Snail to give code completion. Install with `M-x package-install company`, then add `(add-hook 'julia-mode-hook #'company-mode)` and `(global-set-key (kbd "C-<tab>") 'company-complete)` to your init.el.

4. `magit` is a great git interface for Emacs. Install with `M-x package-install magit` and add `(global-set-key (kbd "C-x g") 'magit-status)` to your init.el.

7.2   Development workflow

1. Pull the current version from the master branch on Gitlab: https://git.idiv.de/persefone/persefone-model.

2. If you are working on a new feature, create a new branch to avoid breaking the `master` branch. (The `master` branch on Github should always be in a runnable and error-free state.)

3. Implement your changes.

4. Run an example simulation and the test suite to make sure everything works without crashing (`make run` and `make test` on Linux, or execute `run.jl` and `test/runtests.jl` manually.)

5. Commit your work frequently, and try to keep each commit small. Don't forget to add relevant tests to the test suite.

6. Once your satisfied with your work, do another pull/merge from the `master` branch in case somebody else changed the branch in the meantime. Then merge your work into `master` and push to the Gitlab server.

7. Repeat :-)

The Gitlab issue tracker can be used to create, discuss, and assign tasks, as well as to monitor progress towards milestones/releases. Once we have a first release, we will start using semantic versioning and a changelog.

7.3   Important libraries

Revise.jl

`Revise.jl` allows one to reload code without restarting the Julia interpreter. Get it with `Pkg.add("Revise")`, then add `using Revise` to `.julia/config/startup.jl` to have it automatically available.

Test

Persefone uses the inbuilt Julia testing framework. All new functions should have appropriate tests written for them in the appropriate file in the `test` directory. (See `test/runtests.jl` for details.) There are three ways to run the test suite: in the terminal, executing `make test` or `cd test; julia runtests.jl`; or in the Julia REPL, `Pkg.activate("."); Pkg.test()`.

Documenter.jl

The HTML documentation is generated using Documenter.jl. Therefore, all new functions should have doc-strings attached. New files need to be integrated into the relevant documentation source files in `docs/src`, and if necessary into `docs/builddocs.jl`. To build the documentation, run `make docs`, or `cd docs; julia builddocs.jl` (if using the latter, don't forget to update the date and commit in `docs/src/index.md`).

Graphics and user interface

Persefone uses Makie as a plotting library to generate its output graphics. Additionally, Persefone Desktop uses QML.jl to create its graphical user interface.

Unitful.jl

Throughout the source code, variables can be tagged with their appropriate units using the Unitful.jl library. This makes the code easier to understand, and also allows automatic unit conversion:

```julia
julia> 1ha == 10000m²
true

julia> 2km |> m
2000 m

julia> 2km / 10m
200.0
```

Within Persefone, the following units and dimensions have been imported for direct usage: `cm`, `m`, `km`, `m²`, `ha`, `km²`, `mg`, `g`, `kg`, `Length`, `Area`, `Mass`.

Dates

Persefone expands the default Dates library with the AnnualDate type, which can be used to store dates that recur every year (e.g. migration or harvest). AnnualDates can be compared and added/subtracted just as normal dates. Use `thisyear()` to convert an AnnualDate to a Date.

Chapter 8

Adapting Persefone

A key development goal of Persefone is to be FAIR: *findable, accessible, interoperable, and reusable*. We aim to build a model that is both easy to use and easy to adapt to new situations.

There are multiple ways to adapt Persefone for a new modelling study:

Changing the parameters

The simplest way to adapt Persefone is simply by changing the parameters. Copy `src/parameters.toml` to a new location, adjust it to your needs, and run the model using `julia run.jl -c <configfile>`.

Changing the region

To apply Persefone to a new region, you need to create new input maps of land cover and farmplots. How to do so is described here.

You may also need to change aspects of the `farm` submodel. This is not yet implemented.

Adding new animal species

To implement a new species to the `nature` submodel, add a new file to the `src/nature/species` directory and include it in `src/Persefone.jl`, as well as adding the name of the species to the `nature.targetspecies` parameter. In the new file, implement the species using the @species syntax as described here.

Adding new crop species

Crop growth is not yet implemented.

Adding new farmer behaviour

Farmer behaviour is not yet implemented.

Adding a new submodel

To add a new submodel in addition to the existing ones (`nature`, `crop`, and `farm`), you need to familiarise yourself with the software architecture. In particular, you need to understand how initialisation and scheduling works in `src/core/simulation.jl`, and what information is stored in the `model` object.

If you want to add a new agent type, create a subtype of ModelAgent, implement a stepagent! function for it and add it to Persefone.initmodel.

Linking to another model

Persefone can also be used as a software library and be called from another application. For this purpose, it is set up as a Julia package, with a module exporting various model functions, types, and macros (see `src/Persefone.jl`). Of particular interest are the functions `simulate` (set up and run a complete simulation based on a config file), `initialise` (create one or more model objects from a config file), `simulate!` (do a simulation run with an existing model object), and `stepsimulation!` (update a model object by one time step).

To interface with Julia from another language, see the Julia docs here and here.

Chapter 9

Source code architecture

Chapter 10

Model components

Persefone is divided into four components, three of which are semi-independent submodels:

1. `core` and `world`: These two directories provide the foundation of the model software, which sets up and executes simulation runs. It also reads all input files (the configuration file, landscape maps, and weather data), and provides data output functionality.

2. `nature`: This is an individual-based model of species in agricultural landscapes. It defines the Animal agent type, and a set of macros that can be used to rapidly create new species. It also includes ecological process functions that are useful for all species.

3. `farm`: This is an agent-based model of farmer decision making. It is not yet implemented, but will provide the Farmer agent type.

4. `crop`: This is a mathematical growth model for various crops. It is not yet implemented, but already provides the agent type FarmPlot, representing one field and its associated extent and crop type.
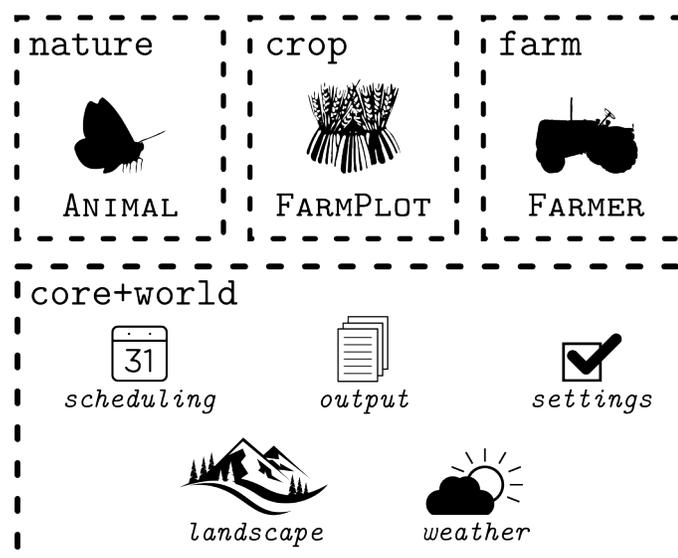
Figure 10.1: "model architecture"

Conceptually, `core` provides functionality that is needed by all of the submodels. Decisions made by `Farmers` affect the `FarmPlots` they own, and (directly or indirectly) the `Animals` in the model landscape.

Chapter 11

Important implementation details

The `model` object

A cursory reading of the source code will quickly show that most functions take an `SimulationModel` object as one of their arguments. The concrete type for this is `AgricultureModel`, a struct that holds all state that is in any way relevant to a simulation run. (Persefone has a strict "no global state" policy to avoid state-dependent bugs and allow parallelisation.) The model object gives access to all agent instances. It also stores the configuration (`model.settings`), the landscape (`model.landscape`, a matrix of `Pixel` objects that store the local land cover, amongst other things), and the current simulation date (`model.date`). (See `Persefone.initmodel` for details.)

Model configuration/the `@param` macro

The model is configured via a `TOML` file, the default version of which is at `src/parameters.toml`. An individual run can be configured using a user-defined configuration file, commandline arguments, or function calls (when Persefone is used as a package rather than an application). During a model run, the `@param` macro can be used
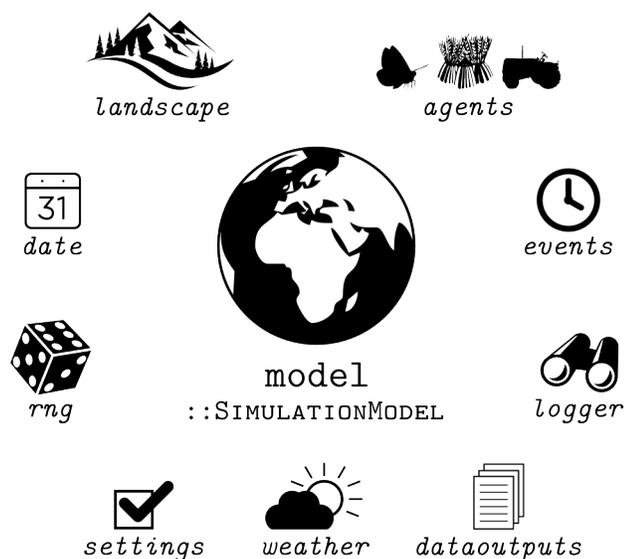

Figure 11.1: "the model object"

to access parameter values. Note that parameter names are prepended with the name of the component they are associated with. For example, the `outdir` parameter belongs to the `[core]` section of the TOML file, and must therefore be referenced as `@param(core.outdir)`. (See `src/core/input.jl` for details.)

> **@param and other macros**
>
> As `@param(parameter)` expands to `model.settings["parameter"]`, it can obviously only be used in a context where the `model` object is actually available. (This is the case for most functions in Persefone, but not for all.) Similarly, many of the `nature` macros depend on specific variables being available where they are called, and can therefore only be used in specific contexts (this is indicated in their documentation).

Output data

Persefone can output model data into text files with a specified frequency (daily, monthly, yearly, or at the simulation end). Submodels can use `Persefone.newdataoutput!` to plug into this system. For an example of how to use this, see `src/nature/ecologicaldata.jl`. (See `src/core/output.jl` for details.)

Farm events

The `FarmEvent` struct is used to communicate farming-related events between submodels. An event can be triggered with `createevent!` and affects all pixels within a `FarmPlot`. (See `src/core/landscape.jl` for details.)

Random numbers and logging

By default in Julia, the random number generator (RNG) and the system logger are two globally accessible variables. As Persefone needs to avoid all global data (since this would interfere with reproducibility in parallel runs), the `model` object stores a local logger and a local RNG. The local logger generally does not change the way the model uses log statements, it is only relevant for some functions in `src/core/simulation.jl`.

> **Using the model RNG**
>
> Whenever you need to use a random number, you must use the `model.rng`. The easiest way to do this is with the `@rand` and `@shuffle!` macros. (Note that these, too, require access to the `model` object.)

Chapter 12

Maps and weather data

Persefone.jl requires three map input files: one for land cover, one for for field geometries, and one for soil types. Additionally, a weather input file is needed. This documents describe how to obtain and process the data needed for each of these.

12.1   Land cover maps

Land cover maps for Germany at 10m resolution can be obtained from Mundialis. These are generated annually from Sentinel data and comprise the following land cover classes:

```
10: forest
20: low vegetation
30: water
40: built-up
50: bare soil
60: agriculture
```

To create a Persefone map input file, you need to crop the national Mundialis map to the extent that you want to simulate (suggestion: approx. 10x10km is a reasonable size).

To do so, download the Mundialis map and import it into QGIS. Then create a new vector layer and create a rectangle feature to delimit the extent of your region. Then go to `Raster -> Extraction -> Clip Raster by Extent`. Select the Mundialis map as the input layer, set the clipping extent by choosing your region vector layer under `Calculate from Layer` and specify the output file name before clicking Run. This will generate a TIF file that you can pass to Persefone as the `landcovermap` parameter.

12.2   Field ID maps

In addition to the land cover data explained above, Persefone also needs information about agricultural field boundaries in order to assign these to the farming agents. Unfortunately, getting this is rather more complicated.

In the EU, every country runs a Land Parcel Information System (LPIS) to administer CAP payments. In Germany, this is called InVeKoS and is run by the Länder. For example, you can view and download the InVeKoS data for Thüringen here. This gives you a vector layer which can be loaded into QGIS. However, it needs to be converted to a raster layer and cropped to your region extent before it can be used in Persefone.

The first thing to do is to make sure that the vector layer has a numeric (!) field with a unique identifier for each field block (check the attribute table). The Thüringen data has the FBI ("Feldblockident") field, but this

is a string value and therefore not usable by the rasteriser. So, we set the vector layer to edit mode, open the field calculator, enter the information for a new field (call it "FID" and set it to a 32-bit integer), and enter `@row_number` in the expression field. Then save the layer and close the calculator.

Secondly, you need to filter out all non-field/non-grassland plot types. (LPIS also has data on forests and various landscape elements that are not relevant to our use case.) Assuming you're working with the Thüringen InVeKoS data (other data sets may have a different structure), right-click on the layer name in QGIS' layer overview and click on "Filter...". Then, enter this expression in the query builder: `"BNK" = 'AL' OR "BNK" = 'GL'` and click "OK". This will select only field and grassland plots.

Next, open the rasteriser (`Raster -> Conversion -> Rasterize`). Select your FID field as the "Field to use for a burn-in value", and your land cover map (as created above - this ensures the two layers match) as the output extent. Make sure the "fixed value to burn" is "Not set". Then choose "Georeferenced units" as the "Out raster size units" and set horizontal and vertical resolution to 10.0. In the advanced parameters, set the output data type to UInt32. Finally, enter an output file name and run. The resulting TIF file can be passed to Persefone as the `farmfieldmap` parameter.

12.3 Soil data

Soil data for Germany is provided by the Bundesanstalt für Geowissenschaften und Rohstoffe in form of the Bodenatlas. This provides a (coarse, but for our purposes sufficient) map of the distribution of the basic soil types such as clay, silt, sand, and loam.

To create the Persefone input file, you need crop this national map to the same region as the other maps, then rasterise it (see instructions in the last paragraph of the previous section).

12.4 Weather data

Currently, Persefone uses historical weather data from the closes weather station as its weather input. (In future, this may be changed to a more detailed raster input, which could then also provide future weather predictions under climate change.) Weather data can be downloaded from the German weather service (DWD). The relevant data are in the folder `daily/kl/historical`.

The description of this data set and the list of weather stations can be found in the Persefone repository, in the docs folder (or downloaded from the link above). Using the list of weather stations, select the one closest to the area of study. Note that not all stations were continuously in operation; make sure that the selected station covers the years of interest.

- **Region Jena:** station number 02444 ("Jena (Sternwarte)")

- **Region Eichsfeld:**

- **Region Thüringer Becken:** station number 00896 ("Dachwig")

- **Region Hohenlohe:**

- **Region Bodensee:**

- **Region Nördlicher Oberrhein:**

From the link above, download the ZIP file associated with the station number. Check the included meta-data if there is any important missing data, or other relevant information (e.g. the station moved position).

Then use `data/extract_weather_data.R` to process the data into the format needed by Persefone:

```
library(tidyverse)

## replace this with the correct file name
weatherfile = "produkt_klima_tag_18210101_20221231_02444.txt"

data = read.table(weatherfile, sep=";", header=T)

weather = data %>%
    ## drop values before 2000 to save space
    filter(MESS_DATUM>=20000101) %>%
    ## select relevant variables and convert place-holder values to NA
    select(MESS_DATUM, FM, RSK, SDK, VPM, TMK, TXK, TNK) %>%
    mutate(date=MESS_DATUM, MESS_DATUM=NULL,
           mean_windspeed=na_if(FM, -999), FM=NULL,
           precipitation=na_if(RSK, -999), RSK=NULL,
           sunshine_hours=na_if(SDK, -999), SDK=NULL,
           mean_vapour_pressure=na_if(VPM, -999), VPM=NULL,
           mean_temperature=na_if(TMK, -999), TMK=NULL,
           max_temperature=na_if(TXK, -999), TXK=NULL,
           min_temperature=na_if(TNK, -999), TNK=NULL)

## replace with the desired file name
write.csv(weather, file="weather_jena.csv", row.names=FALSE)
```

Note: for calculating the reference evapotranspiration ET_0 (needed for the AquaCrop model), use the FAO Penman-Monteith equation.

Chapter 13

Defining new species

In order to make implementing new species as easy as possible, Persefone includes a domain-specific language (DSL) built from a collection of macros and functions.

Here is an example of what this looks like, using a hypothetical mermaid species:

```julia
@species Mermaid begin
    ageofmaturity = 2
    pesticidemortality = 1.0
end

@create Mermaid begin
    @debug "Created $(animalid(self))."
end

@phase Mermaid life begin
    @debug "$(animalid(self)) is swimming happily in its pond."
    @respond pesticide @kill(self.pesticidemortality, "poisoning")
    @respond harvesting @setphase(drought)
    if self.sex == female && length(@neighbours()) < 3 &&
        self.age >= self.ageofmaturity && @landcover() == water
        @reproduce()
    end
end

@phase Mermaid drought begin
    n = sum(1 for a in @neighbours())
    @debug "$(animalid(self)) is experiencing drought with $n neighbour(s)."
    @respond sowing @setphase(life)
end

@populate Mermaid begin
    birthphase = life
    initphase = life
    habitat = @habitat(@landcover() == water)
    pairs=true
end
```

A complete species definition consists of one call each to @species, @create, @populate, and one or more calls to @phase. Another important macro is @habitat. Further macros are available to provide convenience wrappers for common functions. (See `src/nature/nature.jl` for details.)

The first macro to call is `@species`. This takes two arguments: a species name and a definition block (enclosed in `begin` and `end` tags). Within the block, species-specific parameters and variables can be defined (and optionally given values) that should be available throughout a species' lifetime.

Next, each species must define one or more `@phase` blocks. The concept behind this is that species show different behaviours at different phases of their lifecycle. Each @phase block defines the behaviour in one of these phases. (Technically, it defines a function that will be called daily, so long as the species' phase variable is set to this phase.) Code in this section has access to the `model` object as well as a `self` object, which is the currently active `Animal` agent. Within a phase block, `@respond` can be used to define the species' response to a `FarmEvent` that affects the species' current location, while a variety of other macros provide wrappers to life history and movement functions from `src/nature/populations.jl`.

The third macro to call is `@create`. Like @phase, this defines a function with access to the `world` and `self` objects. This function is called whenever a new individual of this species is created (either at birth, or when the model is initialised).

The last macro that must be called is [@populate]. Whereas `@create` regulates the creation of individual animals, @populate determines how the population of a species is initialised at the start of a simulation. It does this by defining values for the parameters used by `initpopulation!`. The full list of parameters that can be used is documented under `PopInitParams`.

The final important macro is `@habitat`. This defines a "habitat descriptor", i.e. a predicate function that tests whether or not a given landscape pixel is suitable for a specified purpose. Such habitat descriptors are used as arguments to various functions, for example for population initialisation or movement. The argument to @habitat consists of a logical expression, which has access to the animal's current position (the `pos` tuple variable) and the `model`. Various macros are available to easily reference information about the current location, such as `@landcover` or `@distancetoedge`.

All of these macros are defined in `src/nature/macros.jl`.

Part V

Software API

Chapter 14

Simulation

The `core` and `world` directories hold source files that are important for all submodels, including scheduling, landscape, weather, and input/output functions.

14.1 Persefone.jl

This file defines the module, including all exported symbols and two high-level types.

`Persefone.ModelAgent` – Type.

```
ModelAgent
```

The supertype of all agents in the model (animal species, farmer types, farmplots).

source

`Persefone.SimulationModel` – Type.

```
SimulationModel
```

The supertype of AgricultureModel. This is needed to avoid circular dependencies (most types and functions depend on `SimulationModel`, but the definition of the model struct depends on these types).

source

`Persefone.AnnualDate` – Type.

```
AnnualDate
```

A type to handle recurring dates (e.g. migration, harvest). Stores a month and a day, and can be compared against normal dates. To save typing, a Tuple{Int64,Int64} is automatically converted to an AnnualDate, allowing this syntax: `nestingend::AnnualDate = (August, 15)`.

source

`Base.randn` – Function.

```
randn(vector)
```

Return a random element from the given vector, following a (mostly) normal distribution based on index values (i.e. elements in the middle of the vector will be returned most frequently).

source

Persefone.bounds – Method.

```
bounds(x; max=Inf, min=0)
```

A utility function to make sure that a number is within a given set of bounds. Returns max/min if x is greater/less than this.

source

Persefone.cycle! – Function.

```
cycle!(vector, n=1)
```

Move the first element of the vector to the end, repeat n times.

source

Persefone.thisyear – Method.

```
thisyear(annualdate, model)
nextyear(annualdate, model)
lastyear(annualdate, model)
```

Convert an AnnualDate to a Date, using the current/next/previous year of the simulation run.

source

Persefone.@areaof – Macro.

```
@areaof(npixels)
```

Calculate the area of a given number of landscape pixels, knowing the resolution of the world map (requires the model object to be available).

source

Persefone.@chance – Macro.

```
@chance(odds)
```

Return true if a random number is less than the odds (0.0 <= odds <= 1.0), using the model RNG. This is a utility wrapper that can only be used a context where the `model` object is available.

source

Persefone.@rand – Macro.

```
@rand(args...)
```

Return a random number or element from the sample, using the model RNG. This is a utility wrapper that can only be used a context where the `model` object is available.

source

Persefone.@randn – Macro.

```
@randn(vector)
```

Return a normally-distributed random number or element from the sample, using the model RNG. This is a utility wrapper that can only be used a context where the `model` object is available.

source

Persefone.@shuffle! – Macro.

```
@shuffle!(collection)
```

Shuffle the given collection in place, using the model RNG. This is a utility wrapper that can only be used a context where the `model` object is available.

source

14.2   simulation.jl

This file includes the basal functions for initialising and running simulations.

Persefone.AgricultureModel – Type.

```
AgricultureModel
```

This is the heart of the model - a struct that holds all data and state for one simulation run. It is created by initialise and passed as input to most model functions.

source

Persefone.finalise! – Method.

```
finalise!(model)
```

Wrap up the simulation. Finalises and visualises output, then terminates.

source

`Persefone.initialise` – Method.

```
initialise(configfile=PARAMFILE, params=Dict())
```

Initialise the model: read in parameters, create the output data directory, and instantiate the Simulation-Model object(s). Optionally allows specifying the configuration file and overriding specific parameters. This returns a single model object, unless the config file contains multiple values for one or more parameters, in which case it creates a full-factorial simulation experiment and returns a vector of model objects.

source

`Persefone.initmodel` – Method.

```
initmodel(settings)
```

Initialise a model object using a ready-made settings dict. This is a helper function for `initialise()`.

source

`Persefone.nagents` – Method.

```
nagents(model)
```

Return the total number of agents in a model object.

source

`Persefone.paramscan` – Method.

```
paramscan(settings)
```

Create a list of settings dicts, covering all possible parameter combinations given by the input settings (i.e. a full-factorial experiment). This is a helper function for `initialise()`.

source

`Persefone.simulate!` – Method.

```
simulate!(model)
```

Carry out a complete simulation run using a pre-initialised model object.

source

`Persefone.simulate` – Method.

```
simulate(configfile=PARAMFILE, params=Dict())
```

Initialise one or more model objects and carry out a full simulation experiment, optionally specifying a configuration file and/or specific parameters.

This is the default way to run a Persefone simulation.

source

Persefone.stepagent! – Method.

```
stepagent!(agent, model)
```

All agent types must define a stepagent!() method that will be called daily.

source

Persefone.stepsimulation! – Method.

```
stepsimulation!(model)
```

Execute one update of the model.

source

14.3 landscape.jl

This file manages the landscape maps that underlie the model.

Persefone.FarmEvent – Type.

```
FarmEvent
```

A data structure to define a landscape event, giving its type, spatial extent, and duration.

source

Persefone.LandCover – Type.

The land cover classes encoded in the Mundialis Sentinel data.

source

Persefone.Management – Type.

The types of management event that can be simulated

source

Persefone.Pixel – Type.

```
Pixel
```

A pixel is a simple data structure to combine land use and ownership information in a single object. The model landscape consists of a matrix of pixels. (Note: further landscape information may be added here in future.)

source

Persefone.createevent! – Function.

```
createevent!(model, pixels, name, duration=1)
```

Add a farm event to the specified pixels (a vector of position tuples) for a given duration.

source

Persefone.directionto – Method.

```
directionto(pos, model, habitatdescriptor)
```

Calculate the direction from the given location to the closest location matching the habitat descriptor function. Returns a coordinate tuple (target - position), or nothing if no matching habitat is found. Caution: can be computationally expensive!

source

Persefone.directionto – Method.

```
directionto(pos, model, habitattype)
```

Calculate the direction from the given location to the closest habitat of the specified type. Returns a coordinate tuple (target - position), or nothing if no matching habitat is found. Caution: can be computationally expensive!

source

Persefone.distanceto – Method.

```
distanceto(pos, model, habitatdescriptor)
```

Calculate the distance from the given location to the closest location matching the habitat descriptor function. Caution: can be computationally expensive!

source

Persefone.distanceto – Method.

```
distanceto(pos, model, habitattype)
```

Calculate the distance from the given location to the closest habitat of the specified type. Caution: can be computationally expensive!

source

Persefone.`distancetoedge` – Method.

```
distancetoedge(pos, model)
```

Calculate the distance from the given location to the closest neighbouring habitat. Caution: can be computationally expensive!

source

Persefone.`farmplot` – Method.

```
farmplot(position, model)
```

Return the farm plot at this position, or nothing if there is none (utility wrapper).

source

Persefone.`inbounds` – Method.

```
inbounds(pos, model)
```

Is the given position within the bounds of the model landscape?

source

Persefone.`initlandscape` – Method.

```
initlandscape(directory, landcovermap, farmfieldsmap)
```

Initialise the model landscape based on the map files specified in the configuration. Returns a matrix of pixels.

source

Persefone.`landcover` – Method.

```
landcover(position, model)
```

Return the land cover class at this position (utility wrapper).

source

Persefone.randomdirection – Method.

```
randomdirection(model, distance)
```

Get a random direction coordinate tuple within the specified distance.

source

Persefone.randompixel – Function.

```
randompixel(position, model, range, habitatdescriptor)
```

Find a random pixel within a given range of the position that matches the habitatdescriptor (create this using @habitat).

source

Persefone.safebounds – Method.

```
safebounds(pos, model)
```

Make sure that a given position is within the bounds of the model landscape.

source

Persefone.updateevents! – Method.

```
updateevents!(model)
```

Cycle through the list of events, removing those that have expired.

source

14.4 weather.jl

This file reads in weather data and makes it available to the model.

Persefone.Weather – Type.

```
Weather
```

A single weather datum, combining the observations from one day.

source

Persefone.initweather – Method.

```
initweather(weatherfile, startdate, enddate)
```

Load a weather file, extract the values that are relevant to this model run (specified by start and end dates), and return a dictionary of Weather objects mapped to dates.

Note: This requires a weather file in the format produced by data/extract_weather_data.R.

source

Persefone.maxtemp – Method.

```
maxtemp(model)
```

Return today's maximum temperature in °C.

source

Persefone.meantemp – Method.

```
meantemp(model)
```

Return today's mean temperature in °C.

source

Persefone.mintemp – Method.

```
mintemp(model)
```

Return today's minimum temperature in °C.

source

Persefone.precipitation – Method.

```
precipitation(model)
```

Return today's total precipitation in mm.

source

Persefone.sunshine – Method.

```
sunshine(model)
```

Return today's sunshine duration in hours.

source

`Persefone.vapourpressure` – Method.

```
vapourpressure(model)
```

Return today's average vapour pressure in hPa.

source

`Persefone.windspeed` – Method.

```
windspeed(model)
```

Return today's average windspeed in m/s.

source

Chapter 15

Input and Output

These functions are responsible for reading in all model configurations (passed by config file or commandline), administrating them during a run, and printing or plotting any output.

15.1   input.jl

Persefone.AVAILABLE_CROPMODELS – Constant.

> The crop models that can be used in the simulation.
>
> source

Persefone.PARAMFILE – Constant.

> The file that stores all default parameters: src/parameters.toml
>
> source

Persefone.flattenTOML – Method.

> ```
> flattenTOML(dict)
> ```
>
> An internal utility function to convert the two-dimensional dict returned by TOML.parsefile() into a one-dimensional dict, so that instead of writing settings["domain"]["param"] one can use settings["domain.param"]. Can be reversed with prepareTOML.
>
> source

Persefone.getsettings – Function.

> ```
> getsettings(configfile, userparams=Dict())
> ```
>
> Combines all configuration options to produce a single settings dict. Precedence: function arguments - commandline parameters - user config file - default values
>
> source

Persefone.loadmodelobject – Method.

```
loadmodelobject(fullfilename)
```

Deserialise a model object that was previously saved with [savemodelobject](@ref).

source

Persefone.parsecommandline – Method.

```
parsecommandline()
```

Certain software parameters can be set via the commandline.

source

Persefone.preprocessparameters – Method.

```
preprocessparameters(settings)
```

Take the raw input parameters and process them where necessary (e.g. convert types or perform checks). This is a helper function for getsettings.

source

Persefone.@param – Macro.

```
@param(domainparam)
```

Return a configuration parameter from the global settings. The argument should be in the form <domain>.<parameter>, for example @param(core.outdir). Possible values for <domain> are core, nature, farm, or crop. For a full list of parameters, see src/parameters.toml.

Note: this macro only works in a context where the model object is available!

source

15.2 output.jl

Persefone.LOGFILE – Constant.

Log output is saved to simulation.log in the output directory

source

Persefone.RECORDDIR – Constant.

All input data are copied to the inputs folder within the output directory

source

Persefone.DataOutput – Type.

```
DataOutput
```

A struct for organising model output. This is used to collect model data in an in-memory dataframe or for CSV output. Submodels can register their own output functions using newdataoutput!.

Struct fields: - frequency: how often to call the output function (daily/monthly/yearly/end/never) - databuffer: a vector of vectors that temporarily saves data before it is stored permanently or written to file - datastore: a data frame that stores data until the end of the run - outputfunction: a function that takes a model object and returns data values to record (formatted as a vector of vectors) - plotfunction: a function that takes a model object and returns a Makie figure object (optional)

source

Persefone.createdatadir – Method.

```
createdatadir(outdir, overwrite)
```

Creates the output directory, dealing with possible conflicts.

source

Persefone.data – Method.

Retrieve the data stored in a DataOutput (assumes core.storedata is true).

source

Persefone.modellogger – Function.

```
modellogger(loglevel, outdir, output="both")
```

Create a logger object that writes output to screen and/or a logfile. This object is stored as model.logger and can then be used with with_logger(). Note: requires createdatadir to be run first.

source

Persefone.newdataoutput! – Function.

```
newdataoutput!(model, name, header, frequency, outputfunction, plotfunction)
```

Create and register a new data output. This function must be called by all submodels that want to have their output functions called regularly.

source

Persefone.outputdata – Function.

```
outputdata(model, force=false)
```

Cycle through all registered data outputs and activate them according to their configured frequency. If `force` is `true`, activate all outputs regardless of their configuration.

source

Persefone.`prepareTOML` – Method.

```
prepareTOML(dict)
```

An internal utility function to re-convert the one-dimensional dict created by `flattenTOML` into the two-dimensional dict needed by `TOML.print`, and convert any data types into TOML-compatible types where necessary.

source

Persefone.`record!` – Method.

```
record!(model, outputname, data)
```

Append an observation vector to the given output.

source

Persefone.`saveinputfiles` – Method.

```
saveinputfiles(model)
```

Copy all input files into the output directory, including the actual parameter settings used. This allows replicating a run in future.

source

Persefone.`savemodelobject` – Method.

```
savemodelobject(model, filename)
```

Serialise a model object and save it to file for later reference. Includes the current model and Julia versions for compatibility checking.

WARNING: produces large files (>100 MB) and takes a while to execute.

source

Persefone.`visualiseoutput` – Method.

```
visualiseoutput(model)
```

Cycle through all data outputs and call their respective plot functions, saving each figure to file.

source

Persefone.`withtestlogger` – Method.

```
withtestlogger(model)
```

Replace the model logger with the currently active logger. This is intended to be used in the testsuite to circumvent a Julia issue, where `@test_logs` doesn't work with local loggers.

source

Persefone.`@data` – Macro.

```
@data(outputname)
```

Return the data stored in the given output (assumes `core.storedata` is true). Only use in scopes where `model` is available.

source

Persefone.`@record` – Macro.

```
@record(outputname, data)
```

Record an observation / data point. Only use in scopes where `model` is available.

source

15.3 makieplots.jl

Persefone.`croptrends` – Method.

```
croptrends(model)
```

Plot a dual line graph of cropped area and average plant height per crop over time. Returns a Makie figure object.

source

Persefone.`datetickmarks` – Method.

```
datetickmarks(dates)
```

Given a vector of dates, construct a selection to use as tick mark locations. Helper function for `[populationtrends](@ref)`

source

Persefone.`populationtrends` – Method.

```
populationtrends(model)
```

Plot a line graph of population sizes of each species over time. Returns a Makie figure object.

source

`Persefone.skylarkpopulation` – Method.

```
skylarkpopulation(model)
```

Plot a line graph of total population size and individual demographics of skylarks over time. Returns a Makie figure object.

source

`Persefone.skylarkstats` – Method.

```
skylarkstats(model)
```

Plot various statistics from the skylark model: nesting habitat, territory size, mortality.

source

`Persefone.visualisemap` – Function.

```
visualisemap(model, date, landcover)
```

Draw the model's land cover map and plot all individuals as points on it at the specified date. If no date is passed, use the last date for which data are available. Optionally, you can pass a landcover map image (this is needed to reduce the frequency of disk I/O for Persefone Desktop). Returns a Makie figure object.

source

Chapter 16

Nature submodel

16.1   nature.jl

This file is responsible for managing the animal modules.

Persefone.Animal – Type.

```
Animal
```

This is the generic agent type for all animals. Individual species are created using the @species macro. In addition to user-defined, species-specific fields, all species contain the following fields:

- id An integer unique identifier for this individual.
- sex male, female, or hermaphrodite.
- parents The IDs of the individual's parents.
- pos An (x, y) coordinate tuple.
- age The age of the individual in days.
- phase The update function to be called during the individual's current life phase.
- energy A DEBparameters struct for calculating energy budgets.
- offspring A vector containing the IDs of an individual's children.
- territory A vector of coordinates that comprise the individual's territory.

source

Persefone.animalid – Method.

```
animalid(animal)
```

A small utility function to return a string with the species name and ID of an animal.

source

Persefone.create! – Method.

```
create!(animal, model)
```

The `create!` function is called for every individual at birth or at model initialisation. Species must use `@create` to define a species-specific method. This is the fall- back method, in case none is implemented for a species.

source

Persefone.initnature! – Method.

```
initnature!(model)
```

Initialise the model with all simulated animal populations.

source

Persefone.killallanimals! – Method.

```
killallanimals!(model)
```

Remove all animal individuals from the simulation.

source

Persefone.speciesof – Method.

```
speciesof(animal)
```

Return the species name of this animal as a string.

source

Persefone.speciestype – Method.

```
speciestype(name)
```

Return the Type of this species.

source

Persefone.stepagent! – Method.

```
stepagent!(animal, model)
```

Update an animal by one day, executing it's currently active phase function.

source

Persefone.updatenature! – Method.

```
updatenature!(model)
```

Run processes that affect all animals.

source

16.2 macros.jl

This file contains all the macros that can be used in the species DSL.

Persefone.@animal – Macro.

```
@animal(id)
```

Return the animal object associated with this ID number. This can only be used in a context where the model object is available (e.g. nested within @phase).

source

Persefone.@countanimals – Macro.

```
@countanimals(radius=0, species="")
```

Count the number of animals at or near this location, optionally filtering by species. This can only be used nested within @phase or @habitat.

source

Persefone.@create – Macro.

```
@create(species, body)
```

Define a special phase function (create!()) that will be called when an individual of this species is created, at the initialisation of the simulation or at birth.

As for @phase, the body of this macro has access to the variables self (the individual being created) and model (the simulation world), and can thus use all macros available in @phase.

source

Persefone.@cropcover – Macro.

```
@cropcover
```

Return the percentage ground cover of the crop at this position, or nothing if there is no crop here. This is a utility wrapper that can only be used nested within @phase or @habitat.

source

Persefone.@`cropheight` – Macro.

```
@cropheight
```

Return the height of the crop at this position, or nothing if there is no crop here. This is a utility wrapper that can only be used nested within @phase or @habitat.

source

Persefone.@`cropname` – Macro.

```
@cropname
```

Return the name of the local croptype, or an empty string if there is no crop here. This is a utility wrapper that can only be used nested within @phase or @habitat.

source

Persefone.@`destroynest` – Macro.

```
@destroynest(reason)
```

Utility wrapper for `destroynest!()` in the Skylark model.

source

Persefone.@`directionto` – Macro.

```
@directionto
```

Calculate the direction to an animal or the closest habitat of the specified type or descriptor. This is a utility wrapper that can only be used nested within @phase or @habitat.

source

Persefone.@`distanceto` – Macro.

```
@distanceto
```

Calculate the distance to an animal or the closest habitat of the specified type or descriptor. This is a utility wrapper that can only be used nested within @phase or @habitat.

source

Persefone.@`distancetoedge` – Macro.

```
@distancetoedge
```

Calculate the distance to the closest neighbouring habitat. This is a utility wrapper that can only be used nested within @phase or @habitat.

source

Persefone.@follow – Macro.

```
@follow(leader, distance)
```

Move to a location within the given distance of the leading animal. This is a utility wrapper that can only be used nested within @phase.

source

Persefone.@habitat – Macro.

```
@habitat
```

Specify habitat suitability for spatial ecological processes.

This macro works by creating an anonymous function that takes in a model object and a position, and returns true or false depending on the conditions specified in the macro body.

Several utility macros can be used within the body of @habitat as a short-hand for common expressions: @landcover, @cropname, @cropheight, @distanceto, @distancetoedge, @countanimals. The variables model and pos can be used for checks that don't have a macro available.

Two example uses of @habitat might look like this:

```
movementhabitat = @habitat(@landcover() in (grass agriculture soil))

nestinghabitat = @habitat((@landcover() == grass ||
                          (@landcover() == agriculture && @cropname() != "maize" &&
                           @cropheight() < 10)) &&
                         @distanceto(forest) > 20)
```

For more complex habitat suitability checks, the use of this macro can be circumvented by directly creating an equivalent function.

source

Persefone.@here – Macro.

```
@here()
```

Return the landscape pixel of this animal's current location. This can only be used nested within @phase.

source

Persefone.@isalive – Macro.

```
@isalive(id)
```

Test whether the animal with the given ID is still alive. This can only be used in a context where the model object is available (e.g. nested within @phase).

source

Persefone.@isoccupied – Macro.

```
@isoccupied(position)
```

Test whether this position is already occupied by an animal of this species. This can only be used nested within @phase.

source

Persefone.@kill – Macro.

```
@kill
```

Kill this animal (and immediately abort its current update if it dies). This is a thin wrapper around kill!, and passes on any arguments. This can only be used nested within @phase.

source

Persefone.@killother – Macro.

```
@killother
```

Kill another animal. This is a thin wrapper around kill!, and passes on any arguments. This can only be used nested within @phase.

source

Persefone.@landcover – Macro.

```
@landcover
```

Returns the local landcover. This is a utility wrapper that can only be used nested within @phase or @habitat.

source

Persefone.@lastyear – Macro.

```
@lastyear(annualdate)
```

Construct a date object referring to the last year in the model from an AnnualDate. Only use in scopes where `model` is available.

source

Persefone.@migrate – Macro.

```
@migrate(arrival)
```

Remove this animal from the map and add it to the migrant species pool. It will be returned to its current location at the specified `arrival` date. This can only be used nested within @phase.

source

Persefone.@move – Macro.

```
@move(position)
```

Move the current individual to a new position. This is a utility wrapper that can only be used nested within @phase.

source

Persefone.@nearby_animals – Macro.

```
@nearby_animals(radius=0, species="")
```

Return an iterator over all animals in the given radius around the current position. This can only be used nested within @phase or @habitat.

source

Persefone.@neighbours – Macro.

```
@neighbours(radius=0, conspecifics=true)
```

Return an iterator over all (by default conspecific) animals in the given radius around this animal, excluding itself. This can only be used nested within @phase.

source

Persefone.@nextyear – Macro.

```
@nextyear(annualdate)
```

Construct a date object referring to the next year in the model from an AnnualDate. Only use in scopes where `model` is available.

source

Persefone.@occupy – Macro.

```
@occupy(position)
```

Add the given position to this animal's territory. Use @vacate to remove positions from the territory again. This can only be used nested within @phase.

source

Persefone.@phase – Macro.

```
@phase(name, body)
```

Use this macro to describe a species' behaviour during a given phase of its life. The idea behind this is that species show very different behaviour at different times of their lives. Therefore, @phase can be used define the behaviour for one such phase, and the conditions under which the animal transitions to another phase.

@phase works by creating a function that will be called by the model if the animal is in the relevant phase. When it is called, it has access to the following variables:

- `self` a reference to the animal itself. This provides access to all the variables defined in the @species definition, as well as all standard Animal variables (e.g. `self.age`, `self.sex`, `self.offspring`).
- `pos` gives the animal's current position as a coordinate tuple.
- `model` a reference to the model world (an object of type SimulationModel). This allows access, amongst others, to `model.date` (the current simulation date) and `model.landscape` (a two-dimensional array of pixels containing geographic information).

Many macros are available to make the code within the body of @phase more succinct. Some of the most important of these are: @setphase, @respond, @kill, @reproduce, @neighbours, @migrate, @move, @occupy, @rand.

source

Persefone.@populate – Macro.

```
@populate(species, params)
```

Set the parameters that are used to initialise this species' population. For parameter options, see PopInitParams.

```
@populate <species> begin
    <parameter> = <value>
    ...
end
```

source

Persefone.@randomdirection – Macro.

```
@randomdirection(range=1)
```

Return a random direction tuple that can be passed to @walk. This is a utility wrapper that can only be used nested within @phase.

source

Persefone.@randompixel – Macro.

```
@randompixel(range, habitatdescriptor)
```

Find a random pixel within a given range of the animal's location that matches the habitatdescriptor (create this using @habitat). This is a utility wrapper that can only be used nested within @phase.

source

Persefone.@reproduce – Macro.

```
@reproduce
```

Let this animal reproduce. This is a thin wrapper around reproduce!, and passes on any arguments. This can only be used nested within @phase.

source

Persefone.@respond – Macro.

```
@respond(eventname, body)
```

Define how an animal responds to a landscape event that affects its current position. This can only be used nested within @phase.

source

Persefone.@setphase – Macro.

```
@setphase(newphase)
```

Switch this animal over to a different phase. This can only be used nested within @phase.

source

Persefone.@species – Macro.

```
@species(name, body)
```

A macro used to add new species types to the nature model. Use this to define species-specific variables and parameters.

The macro works by creating a keyword-defined mutable struct that contains the standard fields described for the Animal type, as well as any new fields that the user adds:

```
@species <name> begin
    <var1> = <value>
    <var2> = <value>
    ...
end
```

To complete the species definition, the @phase, @create, and @populate macros also need to be used.

source

Persefone.@thisyear – Macro.

```
@thisyear(annualdate)
```

Construct a date object referring to the current model year from an AnnualDate. Only use in scopes where model is available.

source

Persefone.@vacate – Macro.

```
@vacate(position)
```

Remove the given position from this animal's territory. This can only be used nested within @phase.

source

Persefone.@vacate – Macro.

```
@vacate()
```

Remove this animal's complete territory. This can only be used nested within @phase.

source

Persefone.@walk – Macro.

```
@walk(direction, speed)
```

Walk the animal in a given direction, which is specified by a tuple of coordinates relative to the animal's current position (i.e. (2, -3) increments the X coordinate by 2 and decrements the Y coordinate by 3.) This is a utility wrapper that can only be used nested within @phase.

source

16.3 individuals.jl

This file contains life-history and other ecological functions that apply to all animal individuals, such reproduction, death, and movement.

Persefone.`followanimal!` – Function.

```
followanimal!(follower, leader, model, distance=0)
```

Move the follower animal to a location near the leading animal.

source

Persefone.`kill!` – Function.

```
kill!(animal, model, probability=1.0, cause="")
```

Kill this animal, optionally with a given percentage probability. Returns true if the animal dies, false if not.

source

Persefone.`migrate!` – Method.

```
migrate!(animal, model, arrival)
```

Remove this animal from the map and add it to the migrant species pool. It will be returned to its current location at the specified `arrival` date.

source

Persefone.`move!` – Method.

```
move!(animal, model, position)
```

Move the animal to the given position, making sure that this is in-bounds. If the position is out of bounds, the animal stops at the map edge.

source

Persefone.`occupy!` – Method.

```
occupy!(animal, model, position)
```

Add the given location to the animal's territory. Returns `true` if successful (i.e. if the location was not already occupied by a conspecific), `false` if not.

source

Persefone.`reproduce!` – Function.

```
reproduce!(animal, model, mate, n=1)
```

Produce one or more offspring for the given animal at its current location. The `mate` argument gives the ID of the reproductive partner.

source

Persefone.vacate! – Method.

```
vacate!(animal, model, position)
```

Remove this position from the animal's territory.

source

Persefone.vacate! – Method.

```
vacate!(animal, model)
```

Remove the animal's complete territory.

source

Persefone.walk! – Function.

```
walk!(animal, model, direction, distance=-1)
```

Let the animal move in the given direction, where the direction is defined by an (x, y) tuple to specify the shift in coordinates. If maxdist >= 0, move no further than the specified distance.

source

Persefone.walk! – Function.

```
walk!(animal, model, direction, distance=1pixel)
```

Let the animal move a given number of steps in the given direction ("north", "northeast", "east", "southeast", "south", "southwest", "west", "northwest", "random").

source

16.4   populations.jl

This file contains functions that apply to all animal populations, such as for initialisation, or querying for neighbours.

Persefone.PopInitParams – Type.

```
PopInitParams
```

A set of parameters used by initpopulation! to initialise the population of a species at the start of a simulation. Define these parameters for each species using @populate.

- initphase determines which life phase individuals will be assigned to at model initialisation (required).
- birthphase determines which life phase individuals will be assigned to at birth (required).
- habitat is a function that determines whether a given location is suitable or not (create this using @habitat). By default, every cell will be occupied.
- popsize determines the number of individuals that will be created, dispersed over the suitable locations in the landscape. If this is zero or negative, one individual will be created in every suitable location. If it is greater than the number of suitable locations, multiple individuals will be created per location. Alternately, use indarea.
- indarea: if this is greater than zero, it determines the habitat area allocated to each individual or pair. To be precise, the chance of creating an individual (or pair of individuals) at a suitable location is 1/indarea. Use this as an alternative to popsize.
- If pairs is true, a male and a female individual will be created in each selected location, otherwise, only one individual will be created at a time. (default: false)
- If asexual is true, all created individuals are assigned the sex hermaphrodite, otherwise, they are randomly assigned male or female. If pairs is true, asexual is ignored. (default: false)

source

Persefone.countanimals – Method.

```
countanimals(pos, model; radius=0, species="")
```

Return the number of animals in the given radius around this position, optionally filtering by species.

source

Persefone.directionto – Method.

```
directionto(pos, model, animal)
```

Calculate the direction from the given position to the animal.

source

Persefone.distanceto – Method.

```
distanceto(pos, model, animal)
```

Calculate the distance from the given position to the animal.

source

Persefone.initindividuals! – Method.

```
initindividuals!(species, pos, popinitparams, model)
```

Initialise one or two individuals (depending on the `pairs` parameter) in the given location. Returns the number of created individuals. (Internal helper function for `initpopulation!()`.)

source

Persefone.initpopulation! – Method.

```
initpopulation!(speciesname, model)
```

Initialise the population of the given species, based on the parameters stored in `PopInitParams`. Define these using `@populate`.

source

Persefone.initpopulation! – Method.

```
initpopulation!(speciestype, popinitparams, model)
```

Initialise the population of the given species, based on the given initialisation parameters. This is an internal function called by `initpopulation!()`, and was split off from it to allow better testing.

source

Persefone.isalive – Method.

isalive(id, model)

Test whether the animal with the given ID is still alive.

source

Persefone.isoccupied – Method.

```
isoccupied(model, position, species)
```

Test whether this location is part of the territory of an animal of the given species.

source

Persefone.nearby_animals – Method.

```
nearby_animals(pos, model; radius= 0, species="")
```

Return a list of animals in the given radius around this position, optionally filtering by species.

source

Persefone.nearby_ids – Method.

```
nearby_ids(pos, model, radius)
```

Return a list of IDs of the animals within a given radius of the position.

source

Persefone.neighbours – Function.

```
neighbours(animal, model, radius=0, conspecifics=true)
```

Return a list of animals in the given radius around this animal, excluding itself. By default, only return conspecific animals.

source

Persefone.populationparameters – Method.

```
populationparameters(type)
```

A function that returns a PopInitParams object for the given species type. Parametric methods for each species are defined with @populate. This is the catch-all method, which throws an error if no species-specific function is defined.

source

Persefone.territorysize – Function.

```
territorysize(animal, model, stripunits=false)
```

Calculate the size of this animal's territory in the given unit. If stripunits is true, return the size as a plain number.

source

16.5 ecologicaldata.jl

This file contains a set of life-history related utility functions needed by species.

Persefone.initecologicaldata – Method.

```
initecologicaldata()
```

Create output files for each data group collected by the nature model.

source

Persefone.saveindividualdata – Method.

```
saveindividualdata(model)
```

Return a data table (to be printed to `individuals.csv`), listing all properties of all animal individuals in the model. May be called never, daily, monthly, yearly, or at the end of a simulation, depending on the parameter `nature.indoutfreq`. WARNING: Produces very big files!

source

`Persefone.savepopulationdata` – Method.

```
savepopulationdata(model)
```

Return a data table (to be printed to `populations.csv`), giving the current date and population size for each animal species. May be called never, daily, monthly, yearly, or at the end of a simulation, depending on the parameter `nature.popoutfreq`.

source

`Persefone.skylarkabundance` – Method.

```
skylarkabundance(model)
```

Save skylark abundance data, including total abundance and demographic data (abundances of breeding/non-breeding/juvenile/migrated individuals).

source

`Persefone.skylarkterritories` – Method.

```
skylarkterritories(model)
```

Return a list of all coordinates occupied by a skylark territory, and the ID of the individual holding the territory. WARNING: produces very big files.

source

Chapter 17

Species models

The ecological submodel in Persefone simulates a range of species in agricultural landscapes.

17.1 Skylark

Persefone.Skylark – Type.

```
Skylark
```

Alauda arvensis is a common and charismatic species of agricultural landscapes.

Sources: - Bauer, H.-G., Bezzel, E., & Fiedler, W. (Eds.). (2012). Das Kompendium der Vögel Mitteleuropas: Ein umfassendes Handbuch zu Biologie, Gefährdung und Schutz (Einbändige Sonderausg. der 2., vollständig überarb. und erw. Aufl. 2005). AULA-Verlag - Delius, J. D. (1965). A Population Study of Skylarks Alauda Arvensis. Ibis, 107(4), 466–492. https://doi.org/10.1111/j.1474-919X.1965.tb07332.x - Donald et al. (2002). Survival rates, causes of failure and productivity of Skylark Alauda arvensis nests on lowland farmland. Ibis, 144(4), 652–664. https://doi.org/10.1046/j.1474-919X.2002.00101.x - Glutz von Blotzheim, Urs N. (Ed.). (1985). Handbuch der Vögel Mitteleuropas. Bd. 10. Passeriformes (Teil 1) 1. Alaudidae - Hirundidae. AULA-Verlag, Wiesbaden. ISBN 3-89104-019-9 - Jenny, M. (1990). Territorialität und Brutbiologie der Feldlerche Alauda arvensis in einer intensiv genutzten Agrarlandschaft. Journal für Ornithologie, 131(3), 241–265. https://doi.org/10.1007/BF01640998 - Püttmanns et al. (2022). Habitat use and foraging parameters of breeding Skylarks indicate no seasonal decrease in food availability in heterogeneous farmland. Ecology and Evolution, 12(1), e8461. https://doi.org/10.1002/ece3.8461

source

Persefone.#1171#fun – Function.

Initialise the skylark population. Creates pairs of skylarks on grassland and agricultural land, keeping a distance of 60m to vertical structures and giving each pair an area of 3ha.

source

Persefone.allowsnesting – Method.

```
allowsnesting(skylark, model, pos)
```

Check whether the given position is suitable for nesting.

source

Persefone.breeding – Method.

Females that have laid eggs take care of their chicks, restarting the nesting process once the chicks are independent or in case of brood loss.

source

Persefone.create! – Method.

Initialise a skylark individual. Selects migration dates and checks if the bird should currently be on migration. Also sets other individual-specific variables.

source

Persefone.destroynest! – Method.

```
destroynest!(skylark, model, reason)
```

Remove the skylark's nest and offspring due to disturbance or predation.

source

Persefone.findterritory – Method.

```
findterritory(skylark, model)
```

Check whether the habitat surrounding the skylark is suitable for establishing a territory. If it is, return the list of coordinates that make up the new territory, else return an empty list.

source

Persefone.foragequality – Method.

```
foragequality(skylark, model, pos)
```

Calculate the relative quality of the habitat at this position for foraging. This assumes that open habitat is best (quality = 1.0), and steadily decreases as vegetation height and/or cover increase. (Linear regressions based on Püttmanns et al., 2021; Jeromin, 2002; Jenny, 1990b.)

source

Persefone.matesearch – Method.

Females returning from migration move around to look for a suitable partner with a territory.

source

Persefone.nesting – Method.

Females that have found a partner build a nest and lay eggs in a suitable location.

source

`Persefone.nonbreeding` – Method.

Non-breeding adults move around with other individuals and check for migration.

source

`Persefone.occupation` – Method.

Once a male has found a territory, he remains in it until the breeding season is over, adjusting it to new conditions when and as necessary.

source

`Persefone.territorysearch` – Method.

Males returning from migration move around to look for suitable habitats to establish a territory.

source

Chapter 18

Crop submodel

Eventually, the plan is to have Persefone include a reimplementation of the AquaCrop model, a well-established crop growth model developed by the FAO. Until then, we are using the crop growth submodel used in ALMaSS.

18.1   farmplot.jl

This file is responsible for the farm plots, i.e. the indidivual fields that farmers manage.

Persefone.`FarmPlot` – Type.

```
FarmPlot
```

A struct representing a single field, on which a crop can be grown.

source

Persefone.`averagefieldsize` – Method.

```
averagefieldsize(model)
```

Calculate the average field size in hectares for the model landscape.

source

Persefone.`cropcover` – Method.

```
cropcover(model, position)
```

Return the crop cover of the crop at this position, or nothing if there is no crop here (utility wrapper).

source

Persefone.`cropheight` – Method.

```
cropheight(model, position)
```

Return the height of the crop at this position, or nothing if there is no crop here (utility wrapper).

source

Persefone.`cropname` – Method.

```
cropname(model, position)
```

Return the name of the crop at this position, or an empty string if there is no crop here (utility wrapper).

source

Persefone.`croptype` – Method.

```
croptype(model, position)
```

Return the crop at this position, or nothing if there is no crop here (utility wrapper).

source

Persefone.`harvest!` – Method.

```
harvest!(farmplot, model)
```

Harvest the crop of this farmplot.

source

Persefone.`isgrassland` – Method.

```
isgrassland(farmplot, model)
```

Classify a farmplot as grassland or not (i.e., is the landcover of >80% of its pixels grass?)

source

Persefone.`sow!` – Method.

```
sow!(farmplot, model, cropname)
```

Sow the specified crop on the farmplot.

source

Persefone.`stepagent!` – Method.

```
stepagent!(farmplot, model)
```

Update a farm plot by one day.

source

Persefone.@harvest – Macro.

```
@harvest()
```

Harvest the current field. Requires the variables `field` and `model`.

source

Persefone.@sow – Macro.

```
@sow(cropname)
```

Sow the named crop on the current field. Requires the variables `field` and `model`.

source

18.2   crops.jl

This includes the types and functions needed for all crop growth model, which are also referenced by the other submodels.

Chapter 19

Farm submodel

Eventually, the aim is to create a full socio-economic farm decision model for Persefone. However, for the time being, we will restrict ourselves to a simple model that executes typical farm operations and crop rotations.

19.1   farm.jl

This file is responsible for managing the farm module(s).

Persefone.BasicFarmer – Type.

```
BasicFarmer
```

The BasicFarmer type simply applies a set crop rotation to his fields and keeps track of income.

source

Persefone.Farmer – Type.

This is the agent type for the farm ABM.

source

Persefone.findsetasides – Method.

```
findsetasides(farmer, model)
```

Return a vector of field IDs that this farmer should keep fallow to satisfy the configured set-aside rules.

source

Persefone.initbasicfarms! – Method.

```
initbasicfarms!(model)
```

Initialise the basic farm model. All fields are controlled by a single farmer actor and are assigned as grassland, set-aside, or arable land with a crop rotation.

source

Persefone.`initfarms!` – Method.

```
initfarms!(model)
```

Initialise the model with a set of farm agents, depending on the configured farm model.

source

Persefone.`stepagent!` – Method.

```
stepagent!(farmer, model)
```

Update a farmer by one day. Cycle through all fields and see what management is needed.

source