Persefone.jl: Modelling Biodiversity in Dynamic Agricultural Landscapes

Daniel Vedder[1,2,3*], Marco C. Matthies[1,3], Gabriel Díaz Iturry[4], Guy Pe'er[1,3]

[1] Department of Biodiversity and People, Helmholtz Centre for Environmental Research - UFZ, Permoserstraße 15, 04318 Leipzig, Germany

[2] Institute of Biodiversity, Ecology and Evolution, Friedrich Schiller University Jena, Dornburger Straße 159, 07743 Jena, Germany

[3] German Centre for Integrative Biodiversity Research (iDiv) Halle-Jena-Leipzig, Puschstraße 4, 04103 Leipzig, Germany

[4] Departamento de Física, Universidad Mayor de San Simón, C. Sucre Esq. Parque La Torre, Cochabamba, Bolivia

* corresponding author: daniel.vedder@idiv.de

1. Agricultural landscapes are highly dynamic, constantly changing across space and time due to the effects of farm management, plant phenology, and environmental fluctuations. These dynamics play a critical role in shaping biodiversity patterns and contribute to negative population trends in numerous species, but are rarely considered in ecological models of farmland biodiversity.
2. Here, we present Persefone.jl, a new process-based model to study wildlife populations in changing agricultural landscapes. By combining land use maps and daily weather data with submodels for farm management and crop growth, Persefone.jl allows the simulation of spatiotemporal landscape dynamics at high resolution. This lays the foundation for a collection of individual-based models of wildlife animal species, which can be used to evaluate the biodiversity impacts of environmental or management scenarios.
3. To showcase the model's capabilities, we simulate the population dynamics of one bird and one butterfly species in six agricultural landscapes in Germany. We demonstrate how empirically observed population patterns emerge from the interaction of model processes, including a management-related ecological trap and weather-dependent population fluctuations.
4. Persefone.jl is open-source and designed to be easily transferable and extensible, so that it can be applied to other regions and scenarios, and can be expanded to include new animal species models. We discuss its potential as a flexible tool for both theoretical and applied agroecological research.

Keywords: agricultural landscapes, farm management, biodiversity, policy, individual-based model

1 Introduction

Agricultural landscapes constantly change. Aside from natural processes like plant phenology and weather, they are shaped particularly by anthropogenic factors, such as farm management. This has numerous effects on biodiversity (Mupepele et al., 2021; Tscharntke et al., 2005). Some of these effects are direct, such as disturbances created by tillage, harvest, or pesticide application. Others are indirect, such as the changing habitats formed by crop rotations, the creation or removal of semi-natural habitat, or the flow of nutrient, water, and pollutants. Together, these effects create spatiotemporal patterns of resource availability and disturbance that influence all species living in agricultural landscapes (Vasseur et al., 2013).

It is well known that many taxa, such as birds and butterflies, are experiencing strong population declines in regions with intensive agriculture, including large parts of Europe (e.g. Rigal et al., 2023; van Swaay et al., 2025). These losses are closely tied to the spatiotemporal dynamics of the surrounding landscapes (Marrec et al., 2022). In response, many conservation efforts are seeking to establish approaches such as wildlife-friendly farming and agroecology (Pywell et al., 2012; Runhaar, 2021). There are also responses at the policy level, using a combination of regulations and subsidies to enforce or encourage agroecological practices. Of particular importance, due to its scope and volume, is the EU's Common Agricultural Policy (CAP), although its benefits to biodiversity have been mixed (Pe'er et al., 2020; Pe'er et al., 2022; Röder et al., 2024). A general challenge is that predicting the ecological effectiveness of agri-environmental measures can be difficult since species differ in their requirements and behavioural responses (Vickery et al., 2004), and outcomes often depend on the landscape context (le Clech et al., 2024).

Simulation models can help assess the likely consequences of changes in agricultural practice, whether policy-induced or otherwise (Topping et al., 2019). However, while economic simulation models are already widely used for agricultural policy assessments, this is not yet the case for biodiversity models (Reidsma et al., 2018). A recent review identified several reasons that contribute to this: First, few biodiversity models simulate the impact of farm management and the associated spatiotemporal dynamics of landscapes. Second, many current biodiversity models are very abstract, often simulating virtual species and landscapes rather than attempting to represent real agroecosystems. Third, few models combine ecological and economic perspectives, for example through jointly considering farmer decision-making, crop production, and biodiversity outcomes (Vedder et al., 2025).

Here we present Persefone.jl, a model of animal populations in dynamic agricultural

landscapes that is intended to address these issues. The model simulates management practices and crop growth on real landscapes, and combines this with a suite of individual-based models of wildlife animal species. This allows it to simulate the spatiotemporal population dynamics of its target species in response to environment and management. Designed to be easy to apply to new regions, species, and scenarios, its aim is to further research into the interactions between agriculture and biodiversity, and to provide a platform for rapid policy assessment in the context of European agricultural landscapes. We demonstrate its current applications and discuss potential and planned developments and applications.

2 Methods

2.1 Model structure

In the following, we describe the model structure of Persefone.jl using an abbreviated form of the ODD protocol (Grimm et al., 2006, 2010), following the guidance by Grimm et al. (2020) for large models. The full model documentation is available in the appendices and on the model website (https://persefone-model.eu), while a graphical summary of the model structure is provided in Fig. 1.

Persefone.jl is designed to fulfil two *purposes*:

1. To represent the spatiotemporal **landscape dynamics** created by arable farming (currently focussing on Europe), including crop rotations, plant growth, and yield formation.
2. To reproduce the **population dynamics** of selected wildlife animal species in response to environmental conditions and agricultural management, considering key mechanisms such as movement, reproduction, and mortality.

To this end, the model simulates three main *entities*, or agents: farmers, agricultural fields, and wildlife animals. These are each represented by separate *submodels*, which are described in more detail in subsequent sections. All agents are located on and interact with the model landscape, which is created by reading in environmental *input data* for the simulated region. These include vector–based land cover and soil type maps, as well as daily weather data. Wherever possible, the model uses standardised data variables to ease transferability. For Germany, automated scripts are available to generate all model inputs from public data sources (Table 1).

In terms of *processes*, the farm submodel manages the fields in the landscape, choosing which crops to grow where and when to carry out management actions. The crop sub-

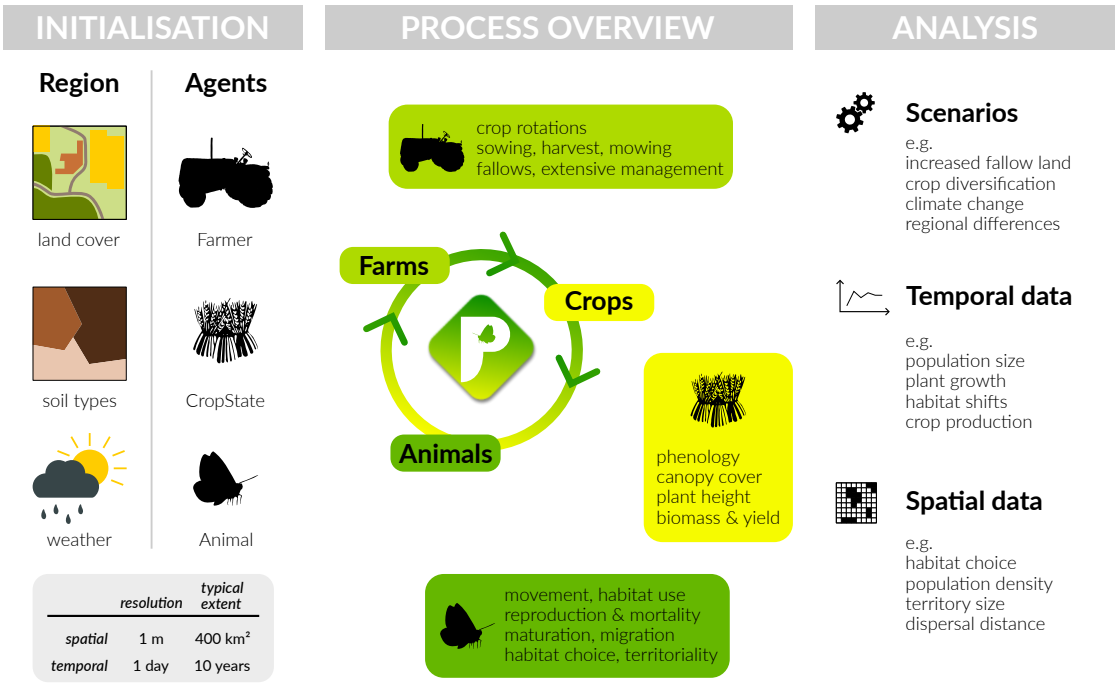

Figure 1: Graphical overview of Persefone.jl. The model contains three major submodels, simulating farm management, crop growth, and animal life cycles.

model simulates the growth of the crop plants on these fields over time, based on the environmental input and farm management. Finally, the animal submodel models the behaviour and life cycle of different indicator species, the individuals of which perceive and interact with the changing landscape created by the other submodels. The model runs at a landscape *scale*, with a spatial resolution of 1 m and daily updates, and a typical extent (depending on the configuration) of around 400 km² and 10 years.

The Persefone.jl software is open–source and can be downloaded from a public git repository. It is implemented in Julia, a programming language designed for performant scientific computing (Bezanson et al., 2017). (The ".jl" suffix in the model name denotes that it is available as a Julia package.) Due to its significant computational demands, it is primarily intended to be run on a high–performance computing cluster (HPC). However, individual simulations can be run on a personal computer for testing and development purposes. For more details, see the user manual in Appendix A.

Table 1: Data sets used as input, for calibration, or for validation. All data are publicly available for our study regions. For links to the sources, see the user manual in Appendix A.

Data	**Description**	**Purpose**	**Source**
Land cover	Vector map of 32 different land cover classes.	input	derived from ATKIS-DLM
Soil types	Vector map of soil types (i.e. different mixtures of clay, silt, and sand).	input	Bundesanstalt für Geowissenschaften und Rohstoffe
Weather	Daily observations of standard meteorological variables from the closest weather station.	input	Deutscher Wetterdienst
Crop phenology	Annual observations of the onset of growth stages (e.g. emergence, flowering, harvest) in different plant species.	calibration / validation	Deutscher Wetterdienst
Crop yield	Annual district-level average yields per hectare.	calibration / validation	Statistische Ämter des Bundes und der Länder
Plant growth	Measurements of crop parameters (e.g. height, biomass) during the course of the growing season.	calibration	Reichenau et al. (2020)
Butterfly monitoring	Population trends of butterflies in Germany.	validation	Kühn et al. (2024)
Common bird monitoring	Population trends of common breeding birds in Germany.	validation	Busch et al. (2020)

2.1.1 Farm management

The farm submodel defines a `FARMER` agent who manages a collection of agricultural fields. In the current model version, a single agent is responsible for all fields in the region, and manages them using a set of configurable practices related to crop rotations, fallows, and grassland management (cf. Table 2).

Crop rotations are defined as a set of crops that are grown sequentially on a given field. Each crop is harvested when it is ripe (as determined by the crop submodel) and the next one sown according to its planting schedule (taken from the agronomic literature). Each year, a number of fields can be left fallow. Grassland is managed either intensively (with 4-5 cuts per year) or extensively (with 2 cuts per year). The proportion of meadows managed extensively is configurable, as is the proportion of arable land left fallow. Currently, management practices that are explicitly simulated are sowing, harvest, and mowing.

2.1.2 Crop growth

The purpose of the crop component is twofold: First, it simulates how agricultural landscapes change phenologically over the course of a year, as different stages of crop growth provide different degrees of habitat quality to wildlife species—thereby enhancing the ecological realism of the landscape. Second, it estimates yields, enabling Persefone.jl to provide economically–relevant output alongside the ecological simulation results.

The crop submodel provides the `CROPSTATE` entity, which is initialised for every arable and grassland field in the simulated region. The farmer (see above) decides when the field is to be sown with a given crop, and when it is to be harvested or mown. Between sowing and harvest (and year-round for grassland) the crop component models how the plants on the field grow. Specifically, it simulates four main output variables: plant height, canopy cover, crop maturity, and yield. These values are available to both the farm and the animal components, and can be used for instance to decide when to harvest (for the farmer) or to calculate habitat suitability (for the animal species).

To simulate these variables, Persefone.jl can use two different crop models. The primary crop model is AquaCrop, originally developed by the FAO and recently translated into Julia (Díaz Iturry et al., 2025). AquaCrop is an intermediate–complexity process–based crop model, which simulates plant growth and maturation based on water availability, meteorological parameters, and soil quality (Raes et al., 2009; Steduto et al., 2009). It has been applied to numerous crops worldwide and compares favourably with other

Table 2: A selection of important model configuration parameters. The full list of parameters can be found in the user manual in Appendix A, species–specific parameters are given in Appendix C.

Parameter	Possible values	Description
`seed`	Integer	The numeric value that is used to seed the random number generator. Simulation runs with identical configuration will have identical outcomes.
`startdate`	Date (1991–2024)	Date on which to initialise the simulation.
`enddate`	Date (1991–2024)	Date on which to terminate the simulation.
`region`	"jena", "eichsfeld", "thueringer_becken", "hohenlohe", "bodensee", "oberrhein"	Name of the region whose input files will be loaded to create the model landscape.
`farmmodel`	"BasicFarmer"	Which implementation of the farm submodel to use (currently, only one is available).
`croprotation`	List of crop names (see Appendix B)	The name and order of crops to use as the crop rotation on arable fields.
`setaside`	Float (0.0–1.0)	Proportion of arable land set aside as annual fallow.
`extensivegrassland`	Float (0.0–1.0)	Proportion of grassland managed extensively.
`scenarios`	" ", "thuringian_fallows"	Names of scenarios to apply (optional). Scenarios are functions that can change configuration settings during the course of a run, or otherwise modify the behaviour of the farm submodel.
`fieldoutfreq`	"daily", "monthly", "yearly", "end", "never"	Frequency with which to output data related to field use and crop growth.
`cropmodel`	"almass","aquacrop"	Crop model(s) to use.
`targetspecies`	"MarbledWhite", "Skylark"	List of animal species to simulate.
`popoutfreq`	"daily", "monthly", "yearly", "end", "never"	Frequency with which to output population-level data from the animal submodel.
`indoutfreq`	"daily", "monthly", "yearly", "end", "never"	Frequency with which to output individual-level data from the animal submodel.

crop models in terms of accuracy and cross–site reliability (Kostková et al., 2021; Mialyk et al., 2024). The challenge is that to achieve maximum accuracy, the model needs to be calibrated with regionally–specific crop growth data, which are not always publicly available.

To overcome this challenge, Persefone.jl complements AquaCrop with a second crop model, namely the vegetation component of the ALMaSS ecosystem model (Topping et al., 2003; Topping & Duan, 2024). This is a correlative model, predicting plant growth based on growing–degree days (i.e. temperature) and time of the year. While it considers fewer environmental conditions than AquaCrop, it has been parameterised for a wide range of crop types in Central Europe, and can also simulate grassland and some non–crop vegetation types. We therefore use it as a fallback for crop types that cannot be parameterised for AquaCrop.

A fuller description of the two crop models, together with details about their calibration and validation, may be found in Appendix B.

2.1.3 Animal species

The animal submodel produces the main ecological output of Persefone.jl, namely the abundance and distribution of the wildlife species over time and space. To do so, it models the behaviour, reproduction, and mortality of individual animals in the changing landscapes generated by the other components.

Each target species is represented by a separate individual-based model with its own set of rules and parameters. Within each species model, the life cycle of the species is decomposed into a series of "life phases". These provide a conceptual framework to structure the differing behaviour and physiology of individuals across their lives, such as a larva, a dispersing juvenile, an adult on winter migration, or a breeding adult (cf. Uchmański & Grimm, 1996). On a technical level, they are implemented as software functions that determine an individual's daily behaviour during each part of its life history, and decide when and under which conditions it switches to a different phase.

All species are impacted by and can respond to environmental conditions. Individuals can at any time access the full state of the simulation, such as the current weather, local land cover, or the state of crop plants in a given field. Individuals can be affected by management actions at their current location, and can interact with other individuals, including those of different species.

2.2 Model case study

Following the above description of the fundamental model structure, we now describe how we set up Persefone.jl for our first study. This study is intended to provide a proof–of–concept for the core claim of our model: that population–level ecological patterns of farmland wildlife species can be reproduced by simulating the spatiotemporal dynamics of agricultural landscapes along with individual animal behaviour and life history. To this end, we simulated a bird and a butterfly species in six regions in Germany.

These study regions were the focus of the research project CAP4GI, which worked together with farmers to develop recommendations for novel agri–environment measures in the CAP (Velten et al., 2023). Three of the regions are in the federal state of Thuringia and are characterised by large farming businesses that emerged from the former East German agricultural cooperatives. The other three are in the federal state of Baden–Württemberg and typically have much smaller, often family–run, farms. Within each of the two states, the regions were chosen to form a gradient of increasing land use intensity, as measured by the proportion of arable area. The regions range in size between 270 km^2 and 1560 km^2, with a median area of 440 km^2 (Fig. 2).

To simulate crop rotation, we selected a cropping sequence consisting of oilseed rape, winter wheat, silage maize, and winter barley, which was carried out on all arable fields with a randomised starting point. These crops were selected because they are the most common in Germany by area, covering two–thirds of the arable land (Duden et al., 2024), and are often used in rotation in our study regions (economic survey data by Theilen–Loges et al., *in review*). We calibrated AquaCrop for these four crops for each region, while using ALMaSS to model plant growth on intensive and extensive grassland as well as on fallows.

Next, we implemented and tested two animal species: the skylark *Alauda arvensis* and the marbled white *Melanargia galathea*. For both species, model design began with a literature search to identify the most important factors shaping their respective population trends, and the mechanisms by which environment and management impact individuals' behaviour and life history. This allowed us to select the mechanisms most likely to determine the observed population trends, thus keeping the species models as simple as possible for their intended purpose (Sun et al., 2016). The literature search also provided the empirical patterns that were later used to validate the model using the POM approach (see section 2.3).

The following paragraphs briefly introduce each of the two species models, while the full ODD documentation is provided in Appendix C.

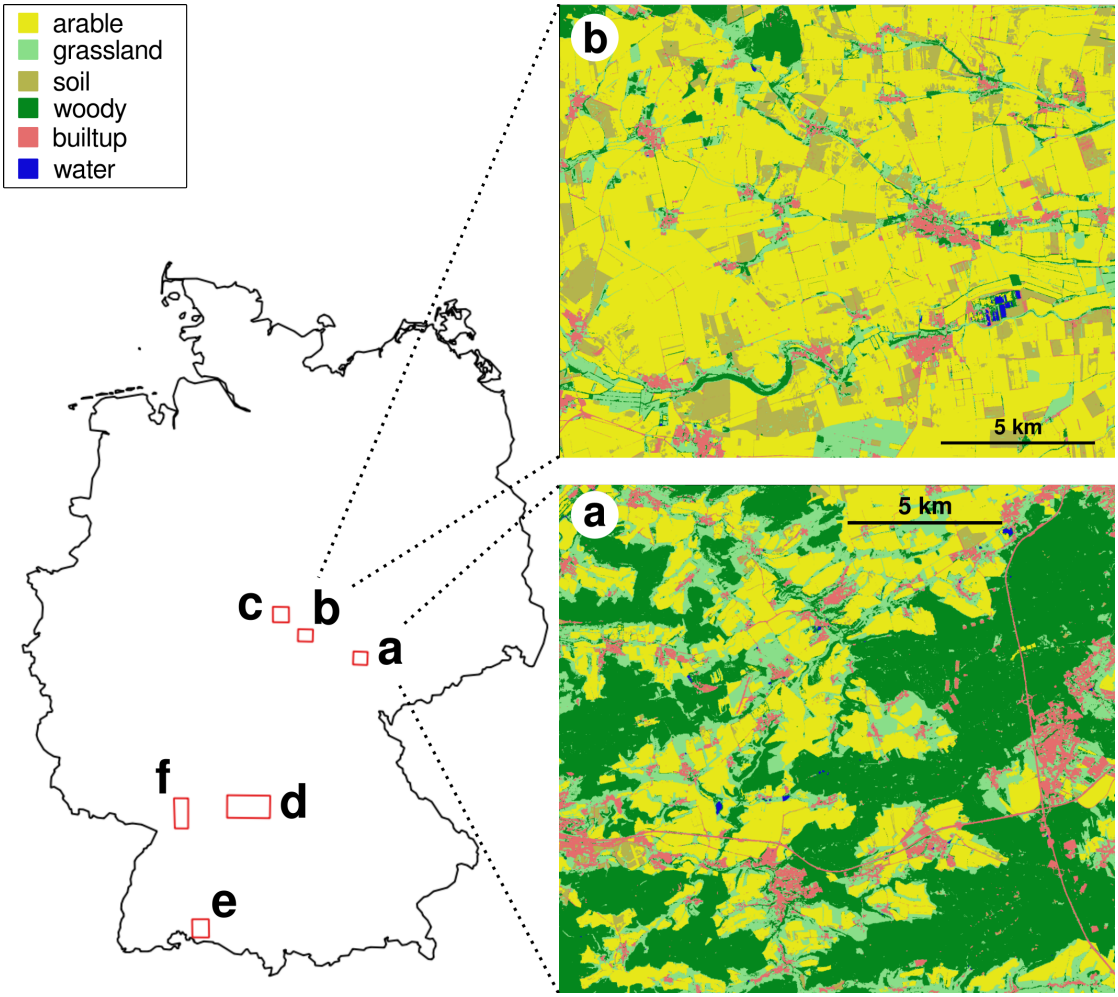

Figure 2: Regions simulated in this study: a) Jena, b) Thuringian Basin, c) Eichsfeld, d) Hohenlohe, e) Bodensee, f) Oberrhein. Insets show land cover of two of the regions, using data by mundialis GmbH & Co. KG (2021).

2.2.1 Skylark

The skylark is a common and charismatic species of agricultural landscapes, which nests on the ground in open areas. Though still common, it has lost over 50 % of its population in Germany over the past decades, due to various factors related to agricultural intensification (Busch et al., 2020). Of particular concern is the increased mortality due to a higher frequency of mowing in grassland, coupled with the increased proportion of less–favoured winter cereals, which pushes skylarks to breed preferentially in the (frequently mown) grassland. This ecological trap has been observed repeatedly and discussed extensively in the agroecological literature (e.g. Donald et al., 2002; Jenny, 1990; Poulsen et al., 1998; Püttmanns, Lehmann et al., 2022).

The phase cycle of the skylark model (Fig. 3a) begins in spring, when the birds return from their winter migration. Males return first and begin to look for a territory of suitable size and location. Females return a little later and proceed to look for an unmated male with a territory with whom they can partner. After mating, a female will build a nest in the male's territory and raise a brood. If the brood has either fledged or is lost due to predation or harvest, she begins a new nest as long as the breeding season is not yet over. After the breeding season, skylarks forage non–territorially in small groups, before leaving for migration in autumn. Interaction with farm management thus revolves around breeding: crop choice and growth affects territory and nesting site choice, and harvest/mowing is an important cause of mortality. We do not model the migration or the period away.

2.2.2 Marbled White

Marbled white butterflies are univoltine grassland specialists that fly in June–August. While highly abundant in some places, and showing a slight positive trend overall in Germany, they do not tolerate fertilised grassland that is frequently mown (Reinhardt et al., 2021). They are also subject to strong population fluctuations caused by weather affecting their reproductive rate (Roy et al., 2001).

In the model (Fig. 3b), adult marbled whites are presumed to move randomly across suitable habitat, with a certain chance of crossing into unsuitable habitat. Only females are simulated, which lay a number of eggs each day as they fly. The distance moved and the number of eggs laid each day is temperature-dependent. Mortality from mowing is low, as the larvae develop close to the ground; most mortality is therefore caused by predation (represented as a constant probability in the model). Thus, the butterflies'

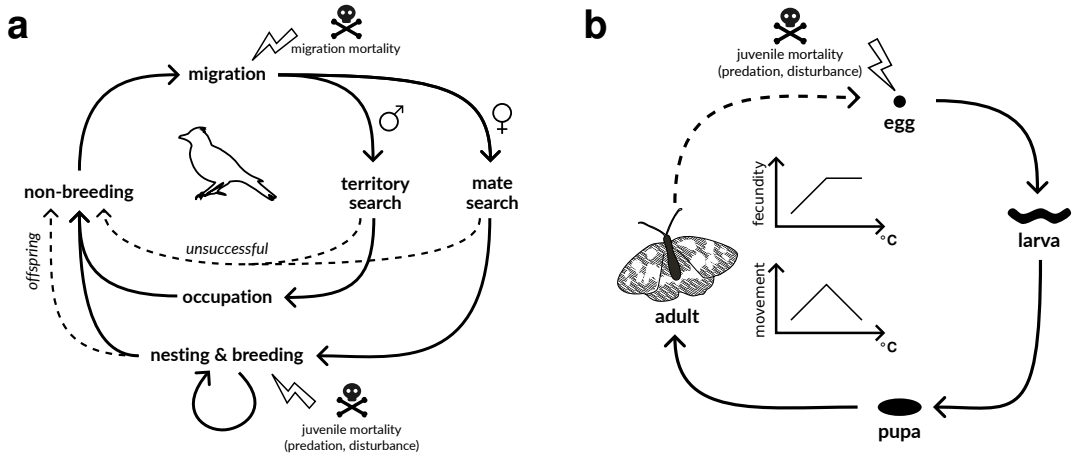

Figure 3: Animal model phase charts: a) Skylark *Alauda arvensis*, b) Marbled White *Melanargia galathea*.

main interaction with farm management is indirect, as they avoid grassland that has been fertilised or recently mown.

2.3 Model verification, calibration, and validation

Throughout the modelling process, we employed multiple techniques to ensure that the individual submodels and the complete model are adequate for their intended purpose (Troost et al., 2023). We utilised the "evaludation" framework proposed by Augusiak et al. (2014) to assess the different aspects of model credibility (Table 3).

To allow a thorough inspection of model functioning, we embedded data logging facilities in each model component to record events, save output data to file, and generate analysis graphs. We also used unit testing and code reviews to verify the technical correctness of our software (Ropella et al., 2002; Vedder et al., 2021).

Another important strategy for establishing reliability is our consequent use of public data from the scientific literature and official data portals. Aside from being openly accessible and well–documented, these data have undergone peer review and/or other quality assurance processes and therefore have a high level of trustworthiness.

Related to this is our choice of using two established crop models, both of which have been used in research for many years. We ensured that our Julia implementations of both crop models corresponded exactly to the original implementations (Díaz Iturry et al., 2025). To carry out the necessary regional calibration of AquaCrop, we used public data sets of district–level yield and phenology data from our study regions, then used

Table 3: Verification and validation strategies for Persefone.jl, following the "evaludation" framework (Augusiak et al., 2014).

Element	Execution in Persefone.jl	Further details
Data evaluation	Use of quantitative and qualitative data published in the scientific literature or on official data portals.	Table 1
Conceptual model evaluation	Use of existing crop models, literature review of empirical work on target animal species, critical feedback by agricultural and taxon experts.	Sections 2.3 and 4.2, Appendix C
Implementation verification	Unit testing, code reviews, event and data logging, for crop models: comparison with reference implementation.	Source code (available online)
Model output verification	For crop models: comparison of output to empirical data, for animal models: comparison with known individual– and population–level patterns.	Section 3, Appendix B
Model analysis	Running model in multiple regions over multiple years with different combinations of animal species parameters.	Section 3, Appendix C
Model output corroboration with new data (*optional*)	In marbled white model: prediction of high abundances in the Oberrhein region.	Section 3

cross–validation to test the robustness of the calculated parameters. (For more details on the calibration and validation of the crop component, see Appendix B.)

For the animal species model, we were most concerned with structural validation, as our aim was to demonstrate the emergence of population dynamics from individual–level mechanisms. For this, we used pattern–oriented modelling (Gallagher et al., 2021; Grimm & Railsback, 2011), using the ecological literature on our target species to identify one population–level and several individual–level patterns for each. We then tested whether the model was capable of replicating the same patterns, without them having been explicitly programmed in.

For the skylarks, we selected four individual–level patterns: a) The size distribution of territories, which were generated procedurally in the model; b) The landscape–dependence of this size distribution; c) The choice of nesting habitat, which depends on the crops available; d) The change of nesting habitat over the course of the breeding season. For the population–level pattern, we chose the ecological trap described above, where the agricul-

tural switch from spring to winter cereals pushes skylark nest–building onto frequently–mown grassland, resulting in population declines. To study these patterns, we set up a simulation experiment with four different scenarios, varying the grassland usage intensity (20 % or 80 % intensive grassland) and the use of winter–sown crops (spring wheat and spring barley or winter wheat and winter barley in the crop rotation). Each scenario was run with ten replicates for each region from 2011–2020.

For the marbled white, we also considered four individual–level patterns: a) the number of eggs laid in a female's lifetime, which depends on temperature and habitat availability; b) the proportion of time spent moving through different habitats; c) the local population density; d) the lifetime displacement distance. On the population level, we selected the abundance trend recorded by the German butterfly monitoring scheme (Kühn et al., 2024). This shows a strongly fluctuating, but overall slightly decreasing trend from 2006 to 2015, followed by an increasing trend from 2016 to 2023. The fluctuations in the first period correlate with the previous year's mean summer temperature. These patterns we tested by running ten replicate simulations in each region from 2006–2022.

Alongside this pattern–oriented modelling, we used exploratory simulations to test the response of the model to different parameter values and combinations, and to identify particularly sensitive parameters (see Appendix C for an overview of parameters and values tested).

3 Results

Fig. 4 shows the comparison of AquaCrop's output after calibration to empirical data from our study regions for each of the four main crops. The graph aligns with previous observations by Kostková et al. (2021), who found that current process–based crop models are not as good at predicting yield as phenology, and that rape yield in particular is not captured well. For two regions (Hohenlohe and Bodensee) gaps in the DWD phenology data meant that we were unable to generate region–specific crop parameters. Here, we used the combined mean values of the other four regions' parameters, and tested these against the regional yield data.

The skylark model generally conforms well to the patterns against which we tested it. This shows that the species model's simple rules, in which territory and nest site selection are primarily based on vegetation height, interacts with the crop submodel to recreate empirically observed patterns of nesting habitat. Glutz von Blotzheim and Bauer (1985) review numerous studies of skylark territories, which found sizes ranging from 0.17–46 ha,

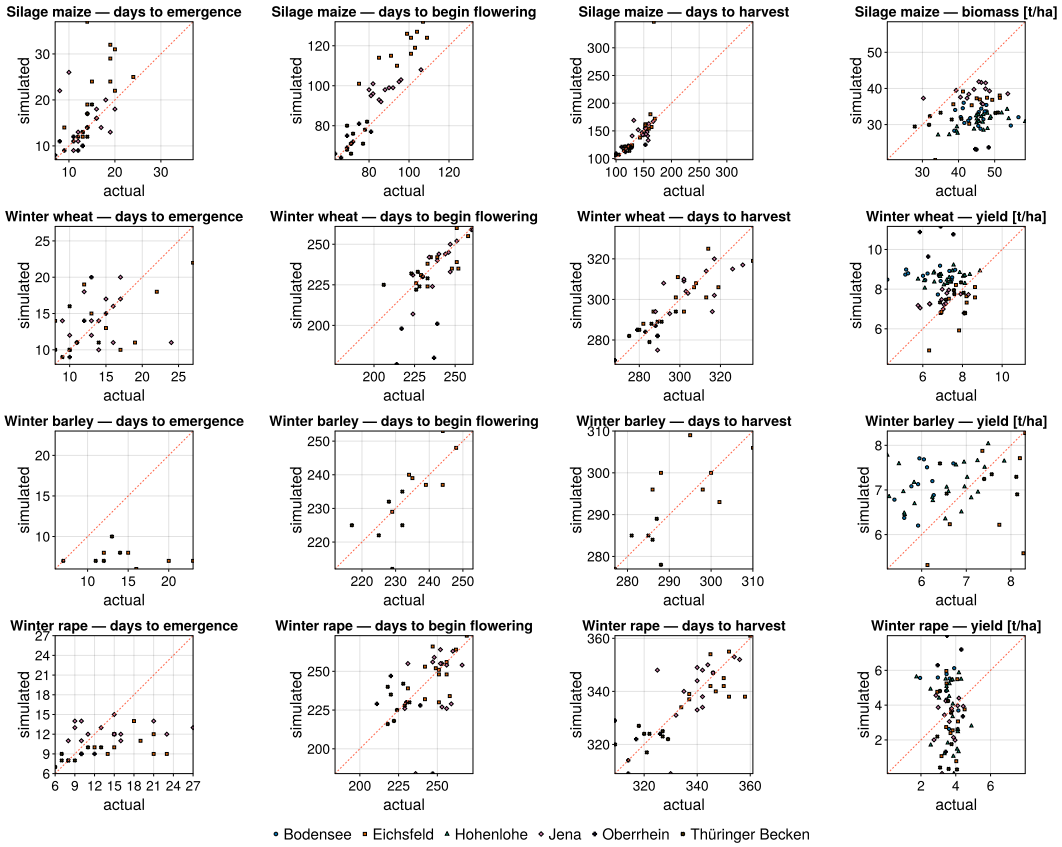

Figure 4: Output of the AquaCrop model compared to empirical data from the study regions, shown here for the four main crop types. The red line is the $x = y$ line, i.e. points above the line are overestimated, points below the line underestimated by the model.

with the most common values around 0.5–2 ha. In our study, Persefone.jl generates territories with a median size of 0.57 ha and an interquartile range of 0.54–0.68 ha, albeit with a very long tail of the distribution (maximum size 137 ha). We also observe that territory sizes in extensively used farmland are smaller on average, although this effect is slight and consists mostly of a reduction of the maximum size (to 45 ha in the spring cereals/extensive grassland scenario). Likewise, the choice of nesting habitat over the breeding season (Fig. 5) closely follows the description provided by empirical studies (Eraud & Boutin, 2002; Jenny, 1990; Miguet et al., 2013). In these observations, as in our model, grassland is always a favoured habitat; spring cereal is preferred over winter cereal, but both disappear later in the season; while maize is more used in the later season. Overall, there is a decrease of nesting attempts towards the later breeding season, associated with a loss of suitable habitat.

The ecological trap of agricultural intensification is also very visible, as is the landscape–dependence of skylark population trends (Fig. 6). Across regions, skylark populations grow over time in the scenario with mostly extensive grassland usage and spring–sown crops, while they decline over time in the scenario with intensive grassland usage and winter–sown crops. Scenarios with either intensive grassland usage or winter crops show intermediate but landscape–dependent trends: In the Thuringian Basin, which is almost entirely arable, population trends show a stark difference between spring and winter cereals, but little to grassland usage intensity. Meanwhile, populations in Jena, which has a high percentage of grassland, show the opposite response.

For the marbled white, the collected lifetime variables also correspond well to known literature values (Fig. 7). Fecundity peaks at around 120 eggs per female, which is in the range of 90–134 eggs given by Reinhardt et al. (2007). Lifetime displacement is usually below 1 km, but can reach up to about 5 km, which agrees with the results of capture–mark–recapture studies. (For instance, Baguette et al., 2000 found most dispersal was <500 m but could be up to 2500 m, whereas Vandewoestijne et al., 2004 report maximum values around 7 km). In terms of movement, extensively managed grassland is the primary habitat used, although some dispersal movement through other habitat types also takes place (cf. Baguette et al., 2000; Lenda & Skórka, 2010).

In terms of the population development, the marbled white model shows a high variability of trends across regions, corresponding to each region's mean summer temperature. As with the monitoring data, the model data too show an initial period of population decline, followed by stabilisation and (in most regions) increase. The effect of weather can also be seen, with pronounced population peaks happening especially in 2007 and 2021, when a hot summer was followed by a cold one. Despite the regional heterogeneity, taking the

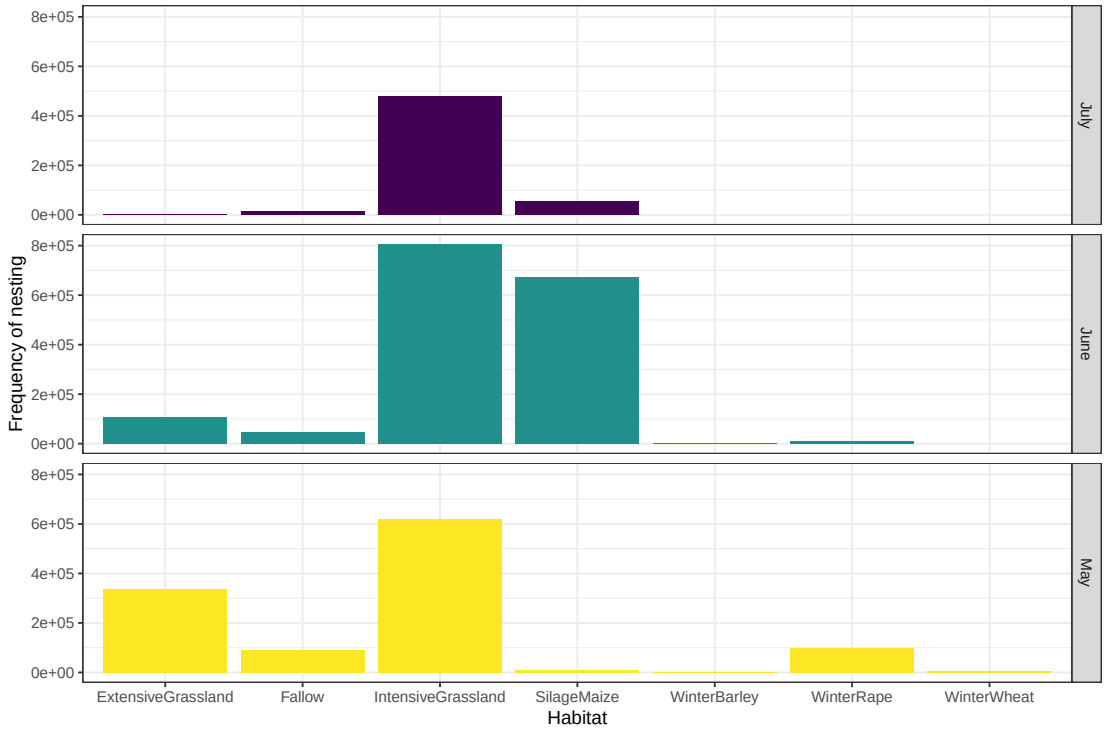

Figure 5: Habitat usage by nesting skylarks over the summer months. Data from the intensive grassland / winter cereal scenario (cf. Fig. 6).

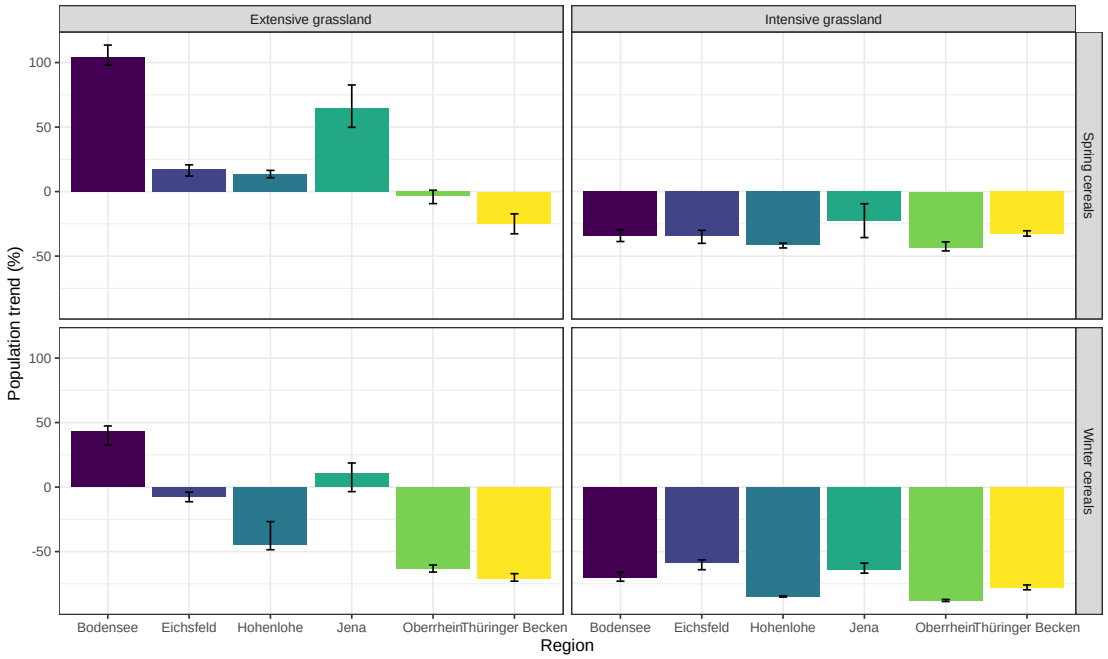

Figure 6: Skylark population trends in four different land-use scenarios in the six model regions. Columns show populations' mean percentage increase/decrease after 10 simulated years (2011-2020), error bars show minimum/maximum values.

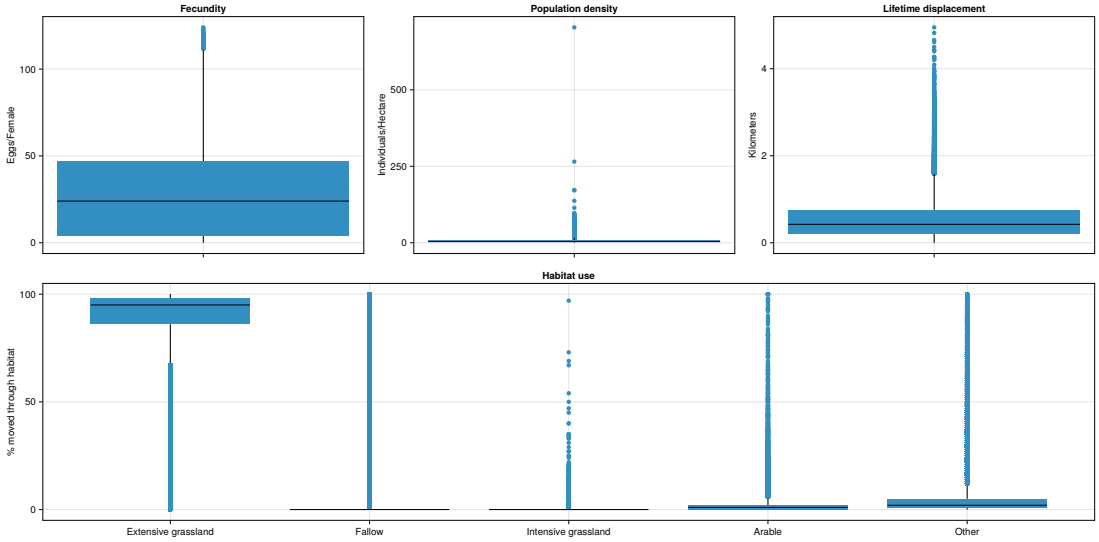

Figure 7: Pattern–testing for the marbled white model, showing several lifetime variables. Top left: Number of eggs laid by each female. Top center: Population density experienced by each individual (i.e. number of conspecifics in the surrounding hectare). Top right: Distance of the location at death from the location at birth for each individual. Bottom: Proportion of movement steps taken in different habitat types. Figure shows output of a sample run in the Jena region (2006–2022, $n = 127818$ individuals).

mean of five of the regions produces a trend that follows the national monitoring trend quite closely (Fig. 8).

The Oberrhein proved to be an outlier in this simulation experiment, as it produced a massively greater population growth than any of the other regions (with a 40–fold increase compared to the 2006 baseline). Investigating the reasons for this result, we found that the region lies in a very atypical bioclimatic zone for Germany (Beckmann et al., 2022) and is noticeably warmer than the rest of the country. We also found previous butterfly mapping efforts that have shown that this is indeed a region with much higher abundances of *Melanargia galathea* than the rest of Baden–Württemberg, thus confirming this prediction by the model (Ebert & Rennwald, 1991). However, because of the non–representative status of this region and the extreme outlier in the results, we excluded the Oberrhein results from Fig. 8 and the calculated mean. (A version of the figure including the Oberrhein data can be found in Appendix C.)

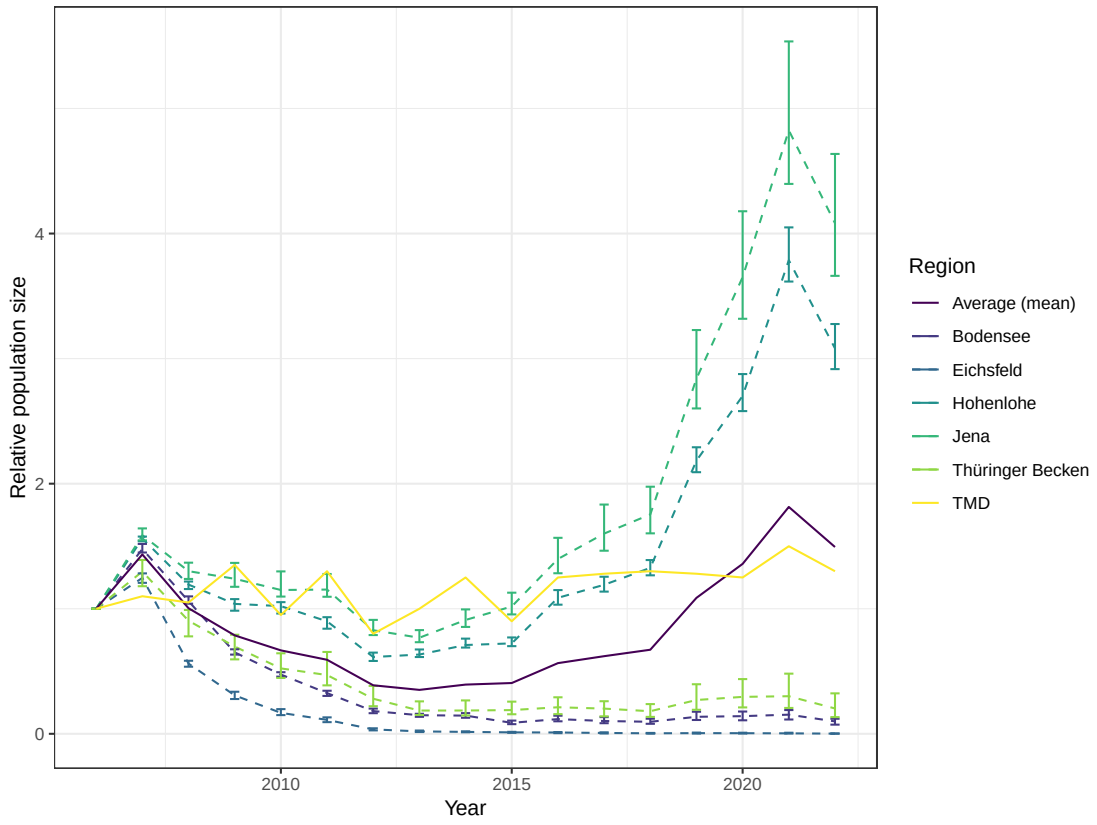

Figure 8: Marbled white population development in five model regions from 2006 to 2022, compared with the national trend collected by the German butterfly monitoring scheme (TMD; solid yellow line). Dotted lines show mean population size of ten replicate simulations in one region, relative to the population size in 2006; error bars show the respective maximum/minimum values. The solid purple line shows the mean trend across all regions and scenarios. TMD trend taken from Kühn et al. (2024). The results from the Oberrhein region were omitted due to over–abundance; a figure including these results can be found in Appendix C.

4 Discussion

4.1 Key features of Persefone.jl

Vedder et al. (2025) identified a need for new agroecological models that explicitly represent farm management and landscape dynamics, simulate real species and landscapes, and link ecological and economic perspectives. Persefone.jl is designed to fill this niche, in order to provide a model that can give insights into the interactions between agricultural management and wildlife species in European landscapes.

As the study results above show, Persefone.jl can successfully reproduce both landscape dynamics and population dynamics that represent realistic spatiotemporal patterns in agroecosystems. One feature of the model that is particularly important to both of these aims is the inclusion of the crop growth submodel, which is also crucial for linking the human and natural domains of agriculture. On the one hand, modelling explicitly crops allows the model to track intra–annual changes to habitats and resources in the landscape, as well as disturbance events caused by farm management (Marrec et al., 2022; Vasseur et al., 2013). The importance of this for biodiversity is seen clearly in the example of the skylark model above. On the other hand, crop growth is the foundation of arable agriculture, and therefore key to including agronomic and economic perspectives. Modelling crops allows the farm submodel to take management decisions (e.g. date of harvest) dynamically, depending on annual conditions and not just a preset schedule—a particularly important feature in light of climate change and necessary e.g. for modelling the impact of extreme events. It also allows the model to generate output data on yields, enabling a comparison of scenario outcomes from both economic and ecological perspectives. For these reasons, integrating crop growth in agroecological models is a necessary step towards a truly social–ecological modelling of agriculture.

Another notable feature of Persefone.jl is its use of continuous–space vector maps. While individual-based models typically use grid–based or patch–based representations of space, the use of continuous–space maps provides multiple scientific and technical benefits (cf. Chevy et al., 2025). One major benefit is the ability to represent small linear or point structures in the landscape (e.g. hedges or trees), which can be ecologically very important but often don't appear on raster maps due to their limited resolution. Another benefit is that with vector maps, discrete habitat areas are directly represented as objects in the model, allowing habitat–level operations such as density–dependence or management to be implemented much more easily than on a raster map (where the same habitat area may cover numerous grid cells). In addition, computational performance is improved

as there are less objects to keep in memory compared with a raster map of equivalent resolution, and the number of objects in a given area doesn't grow with the square of the radius.

We should note that a comparable model to Persefone.jl is already available. This is ALMaSS, which has a long track record of use in agroecological research (e.g. Topping et al., 2003; Topping et al., 2019). While the aim and concept of Persefone.jl and ALMaSS are quite similar (and we partly make use of their vegetation submodel), there are notable differences in modelling philosophy and design. For instance, while ALMaSS embraces complexity and consistently chooses the highest–realism implementation option possible (Topping, 2022), Persefone.jl pursues a policy of minimum–necessary complexity, leaving out any details that are not significantly important to the modelling purpose (Sun et al., 2016). Furthermore, Persefone.jl gives a much greater priority to ease of use, transferability, and extensibility, with the intention that the software can be used by researchers independent of our own group (Meier et al., 2025).

This point is important, as another feature of Persefone.jl is its strong focus on modularity and extensibility. The model is designed to be readily adaptable in three dimensions. First, configuring the model for a new study region relies only on data that can either be self–generated (e.g. land cover) or are available as standardised variables (e.g. weather). For Germany, all required data are publicly available and the necessary preprocessing steps are almost completely automated, but the model is not principally limited to any country. Second, we use a two–tiered system to make adding new species as easy as possible: the behavioural rules for a group of species with a similar biology are implemented using a domain–specific language, which provides a succinct and readable syntax and offers a set of inbuilt functions for common tasks (Holst & Belete, 2015), whilst new members of such a species group can be added using a simple data file to configure species–specific variables (using the Type Object programming pattern; Nystrom, 2014). Third, the farm and crop submodels can be extended with new crop species and management scenarios, or even replaced with other equivalent submodels. The modular structure of the source code means that different implementations of these submodels can be slotted in, as long as they conform to a basic software interface (Vedder et al., 2024). This also opens up the possibility of coupling Persefone.jl to other agricultural, environmental, or economic models, such as models of farmer decision–making or soil processes. For all of these use cases, an emphasis on software quality and documentation is meant to ensure that the model can be successfully used by other researchers (McIntire et al., 2022; Vedder et al., 2021).

4.2 Uncertainty and limitations

Like any socio–environmental model, Persefone.jl has multiple sources of uncertainty (Ascough et al., 2008). Keeping in mind the dictum that "model validity is the adequacy of a model for its intended purpose" (Troost et al., 2023), we will here focus on the consequences of structural and parametric uncertainty for the validity of our model.

One source of structural uncertainty comes from the farm submodel, which is a highly simplified representation of real–world farm management. Following our first purpose of reproducing spatiotemporal landscape dynamics (see section 2.1), we selected crop rotations, sowing, harvest, and mowing as the most relevant management actions to model here. This of course leaves out many other management actions that also affect crops and biodiversity, including tillage, fertilisation, and pesticide application. Excluding these likely leads to greater uncertainty in the predictions of the crop model, as the latter then lacks information about the precise treatment of the crops. The exclusion is also likely to lead to an overestimation of wildlife population trends, as the listed management actions constitute ecological disturbances that can negatively affect animal species. These issues could be addressed by using a more detailed farm submodel.

In the crop submodel, the most relevant source of uncertainty is parametric. Due to the heterogeneity of local conditions, crop–growth models are generally more accurate the more specifically they can be calibrated for a given site (Kostková et al., 2021). However, the data to do so are not always available, particularly for more complex crop models with many parameters. Our approach to this challenge is two–fold: With AquaCrop, we have an intermediate–complexity crop model which can be calibrated to the district level using public data for the most common crops. For other crops, we use the simpler ALMaSS vegetation submodel, which comes with default parameters for Central Europe. The inaccuracy associated with the output of each crop model will especially impact the yield predictions (although these are currently not used further), as well as the changing composition of crop habitats over the growing season (Fig. 4). However, the results of the pattern–oriented modelling suggest that the crop model output is reliable enough to allow the animal models to replicate the tested patterns, which is the most important purpose of the crop model.

The animal models include both parametric and structural uncertainty. Parameter values for the skylark were easier to establish from the literature than for the marbled white, as the former has been extensively studied over decades. The result is that we had to keep the marbled white model very simple, and test a wider range of possible values for its parameters (Appendix C). Calibration of the marbled white model was further

hampered by the lack of published regional abundance data, so that we had to use the agglomerated national–level trend instead. In terms of structural uncertainty, we excluded several mechanisms that can have an influence on species by focusing on the ecological mechanisms with the presumed greatest general impact on population trends. For example, while insect declines have been linked to food shortages for farmland birds such as skylarks, this only appears to be demographically relevant in some regions and was therefore excluded (Püttmanns, Böttges et al., 2022). Similarly, the marbled white model excludes mechanisms such as temperature–dependent mortality in winter or insecticide exposure. As with the simplifications in the farm submodel, these omissions may lead to an overestimation of population trends. They also mean that some questions (e.g. concerning the ecological effects of pesticide use) will require an expansion of the farm and animal submodels before they can be answered using Persefone.jl.

Beyond these sources of scientific uncertainty, two practical limitations of Persefone.jl should be mentioned. First, despite our aim of keeping model complexity low, the challenges inherent in simulating three different and interacting domains (farms, crops, animals) mean that our software does have a significant technical complexity and high computational demands, which makes it quite clearly a research model intended primarily for academic use. While we are working on a graphical user interface to support the analysis and communication of model results, the given constraints mean that this software is unlikely to become a decision–support tool for practitioners. The second limitation is that, as has been stated, Persefone.jl is currently quite focused on agroecosystems, and does not consider the dynamics of non–agricultural habitats. However, this is not a necessary limitation: in the model's source code, agricultural areas are merely a type of spatial object that happens to have management and plant growth processes attached to it. Therefore, future versions of the model could make more habitat types dynamic by implementing processes such as forest phenology, urban park management, or road traffic. This means that while this first version of Persefone.jl is targeted at agroecological questions, its potential use as a modelling platform is much wider.

4.3 Potential for future research

With its integrated modelling of farm management, crop growth, and animal life cycles, Persefone.jl offers wide–ranging possibilities for agroecological research, spanning the continuum from theoretical to applied research.

In terms of theoretical and basic research, Persefone.jl can provide a platform for modelling the ecology of different animal species in agricultural landscapes. For instance, the

marbled white species model presented above could be used to further investigate the physiological mechanisms of temperature–dependence in butterfly population dynamics, ideally in a study that combines modelling with empirical work (as envisaged by Stillman et al., 2015). Other ecological modellers may also find the existing structure of Persefone.jl a useful basis to build their own species models on. Furthermore, the model's use of weather data means that questions related to climate change can be addressed by importing weather forecasts generated by climate models (Cabral et al., 2023), while the model's dynamic landscapes enable studies looking into the effects of intra–annual resource fluctuations and habitat changes (Katna et al., 2023; Schellhorn et al., 2015).

Beyond purely ecological research, Persefone.jl has the potential to become a tool for social–ecological research. Future expansions of the farm submodel (for example by coupling with existing agent-based models) can complement the model's biodiversity focus with a socio–economic perspective. This would allow pursuing new research questions, for instance tracing the impact of global market changes or behavioural factors on farmer decision–making and agroecosystems (cf. Drechsler, 2020). The simulation of crop growth in Persefone.jl also opens up the possibility of using it to study the feedback of biodiversity on food production through the action of ecosystem services, which is currently a major research challenge (Alexandridis et al., 2022; Seppelt et al., 2020).

Looking at more applied research questions, the ability to quickly set up new regions and management scenarios in Persefone.jl make it a promising instrument to assess the impacts of alternative policy scenarios. This, indeed, was one of our major reasons for creating the model, as currently there are few biodiversity models in active use in the EU policy arena (Candel, 2022; Reidsma et al., 2018). Here, the aim is to be able to provide rapid forecasts of the likely ecological effects of proposed policy changes (e.g. the 2024 derogation of the CAP's fallows regulation, or new agri–environment schemes), in order to deliver relevant inputs to the science–policy interface both during the design (Pe'er et al., 2020; Will et al., 2021) and implementation of agricultural policy (Pe'er et al., 2017; Pe'er et al., 2022).

In outlook, our two highest priorities for the further development of Persefone.jl are the implementation of further animal species models and the integration of an economic farm submodel. Our aim is to build up a portfolio of indicator species models that can be used to assess biodiversity responses to alternative policy scenarios. In addition, we will couple Persefone.jl to a socio–economic model of farmer decision–making, in order to explore some of the social–ecological interactions outlined above.

4.4 Conclusion

We present Persefone.jl, a process–based model of wildlife animal populations in dynamic agricultural landscapes. By simulating farm management, crop growth, and animal behaviour, we capture both direct and indirect effects of agriculture on species' demographics. Pattern–oriented modelling confirms that our mechanistic approach can reproduce empirically observed phenomena. We believe these components make Persefone.jl a useful tool for agroecological research and policy evaluation, and hereby make it available as an open–source research software.

Acknowledgements

The authors warmly thank the other members of the CAP4GI consortium for their excellent collaboration throughout the project, as well as our colleagues in the Department of Biodiversity and People for their support. We thank Chris Topping for his help on reimplementing the ALMaSS vegetation submodel. DV, MCM, GDI, and GP acknowledge funding through the project CAP4GI by the Federal Ministry of Education and Research (BMBF), within the framework of the Strategy, Research for Sustainability (FONA, www.fona.de/en) as part of its Social-Ecological Research funding priority, funding no. 01UT2102A. DV, MCM, and GP gratefully acknowledge the support of iDiv, funded by the German Research Foundation (DFG–FZT 118, 202548816). GP and DV acknowledge funding from the EU Horizon Europe project Agroecology–TRANSECT (contract number 101060816). The paper only reflects the opinions of the authors and not of the European Commission or other funders. Responsibility for the content of this publication lies only with the authors.

Data availability

All source code and input files are archived on Zenodo (https://doi.org/10.5281/zenodo.16993215). The development version is available at https://git.idiv.de/persefone/persefone-model.

Author contributions (CRediT)

DV: Conceptualization, Formal Analysis, Investigation, Methodology, Software, Visualization, Writing – original draft, Writing – review & editing; MCM: Formal Analysis, Investigation, Software, Writing – review & editing; GDI: Formal Analysis, Investigation, Software, Writing – review & editing; GP: Conceptualization, Methodology, Funding acquisition, Supervision, Writing – review & editing

References

Alexandridis, N., Marion, G., Chaplin-Kramer, R., Dainese, M., Ekroos, J., Grab, H., Jonsson, M., Karp, D. S., Meyer, C., O'Rourke, M. E., Pontarp, M., Poveda, K.,

Seppelt, R., Smith, H. G., Walters, R. J., Clough, Y., & Martin, E. A. (2022). Archetype models upscale understanding of natural pest control response to land-use change. *Ecological Applications*, *32*(8), e2696. https://doi.org/10.1002/eap.2696

Ascough, J. C., Maier, H. R., Ravalico, J. K., & Strudley, M. W. (2008). Future research challenges for incorporation of uncertainty in environmental and ecological decision-making. *Ecological Modelling*, *219*(3), 383–399. https://doi.org/10.1016/j.ecolmodel.2008.07.015

Augusiak, J., Van den Brink, P. J., & Grimm, V. (2014). Merging validation and evaluation of ecological models to 'evaludation': A review of terminology and a practical approach. *Ecological Modelling*, *280*, 117–128. https://doi.org/10.1016/j.ecolmodel.2013.11.009

Baguette, M., Petit, S., & Quéva, F. (2000). Population spatial structure and migration of three butterfly species within the same habitat network: Consequences for conservation. *Journal of Applied Ecology*, *37*(1), 100–108. https://doi.org/10.1046/j.1365-2664.2000.00478.x

Beckmann, M., Didenko, G., Bullock, J. M., Cord, A. F., Paulus, A., Ziv, G., & Václavík, T. (2022). Archetypes of agri-environmental potential: A multi-scale typology for spatial stratification and upscaling in Europe. *Environmental Research Letters*, *17*(11), 115008. https://doi.org/10.1088/1748-9326/ac9cf5

Bezanson, J., Edelman, A., Karpinski, S., & Shah, V. B. (2017). Julia: A Fresh Approach to Numerical Computing. *SIAM Review*, *59*(1), 65–98. https://doi.org/10.1137/141000671

Busch, M., Katzenberger, J., Trautmann, S., Gerlach, B., Dröschmeister, R., & Sudfeldt, C. (2020). Drivers of population change in common farmland birds in Germany. *Bird Conservation International*, *30*(3), 335–354. https://doi.org/10.1017/S0959270919000480

Cabral, J. S., Mendoza-Ponce, A., da Silva, A. P., Oberpriller, J., Mimet, A., Kieslinger, J., Berger, T., Blechschmidt, J., Brönner, M., Classen, A., Fallert, S., Hartig, F., Hof, C., Hoffmann, M., Knoke, T., Krause, A., Lewerentz, A., Pohle, P., Raeder, U., … Zurell, D. (2023). The road to integrate climate change projections with regional land-use–biodiversity models. *People and Nature*, *6*(5), 1716–1741. https://doi.org/10.1002/pan3.10472

Candel, J. (2022). EU food-system transition requires innovative policy analysis methods. *Nature Food*, *3*, 296–298.

Chevy, E. T., Min, J., Caudill, V., Champer, S. E., Haller, B. C., Rehmann, C. T., Smith, C. C. R., Tittes, S., Messer, P. W., Kern, A. D., Ramachandran, S., & Ralph, P. L. (2025). Population Genetics Meets Ecology: A Guide to Individual-Based Simulations in Continuous Landscapes. *Ecology and Evolution*, *15*(4), e71098. https://doi.org/10.1002/ece3.71098

Díaz Iturry, G., Matthies, M. C., Pe'er, G., & Vedder, D. (2025). AquaCrop.jl: A Process-Based Model of Crop Growth. *Journal of Open Source Software*, *10*(110), 7944. https://doi.org/10.21105/joss.07944

Donald, P. F., Evans, A. D., Muirhead, L. B., Buckingham, D. L., Kirby, W. B., & Schmitt, S. I. A. (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

Drechsler, M. (2020). Model-based integration of ecology and socio-economics for the management of biodiversity and ecosystem services: State of the art, diversity and current trends. *Environmental Modelling & Software*, *134*, 104892. https://doi.org/10.1016/j.envsoft.2020.104892

Duden, C., Nacke, C., & Offermann, F. (2024). German yield and area data for 11 crops from 1979 to 2021 at a harmonized spatial resolution of 397 districts. *Scientific Data*, *11*(1), 95. https://doi.org/10.1038/s41597-024-02951-8

Ebert, G., & Rennwald, E. (1991). *Die Schmetterlinge Baden-Württembergs, Bd.2, Tagfalter: Satyridae, Libytheidae, Lycaenidae, Hesperiidae*. Verlag Eugen Ulmer.

Eraud, C., & Boutin, J.-M. (2002). Density and productivity of breeding Skylarks Alauda arvensis in relation to crop type on agricultural lands in western France. *Bird Study*. https://doi.org/10.1080/00063650209461277

Gallagher, C. A., Chudzinska, M., Larsen-Gray, A., Pollock, C. J., Sells, S. N., White, P. J. C., & Berger, U. (2021). From theory to practice in pattern-oriented modelling: Identifying and using empirical patterns in predictive models. *Biological Reviews*, 21. https://doi.org/10.1111/brv.12729

Glutz von Blotzheim, U. N., & Bauer, K. M. (Eds.). (1985). *Handbuch der Vögel Mitteleuropas (10,1 : Passeriformes ; T. 1); [Alaudidae - Hirundinidae]*. AULA-Verl.

Grimm, V., Berger, U., Bastiansen, F., Eliassen, S., Ginot, V., Giske, J., Goss-Custard, J., Grand, T., Heinz, S. K., Huse, G., Huth, A., Jepsen, J. U., Jørgensen, C., Mooij, W. M., Müller, B., Pe'er, G., Piou, C., Railsback, S. F., Robbins, A. M., ... DeAngelis, D. L. (2006). A standard protocol for describing individual-based and agent-based models. *Ecological Modelling*, *198*(1–2), 115–126. https://doi.org/10.1016/j.ecolmodel.2006.04.023

Grimm, V., Berger, U., DeAngelis, D. L., Polhill, J. G., Giske, J., & Railsback, S. F. (2010). The ODD protocol : A review and first update. *Ecological Modelling*, *221*, 2760–2768. https://doi.org/10.1016/j.ecolmodel.2010.08.019

Grimm, V., & Railsback, S. F. (2011). Pattern-oriented modelling: A 'multi-scope' for predictive systems ecology. *Philosophical Transactions of the Royal Society B*, *367*(1586), 298–310. https://doi.org/10.1098/rstb.2011.0180

Grimm, V., Railsback, S. F., Vincenot, C. E., Berger, U., Gallagher, C., DeAngelis, D. L., Edmonds, B., Ge, J., Giske, J., Groeneveld, J., Johnston, A. S. A., Milles, A., Nabe-Nielsen, J., Polhill, J. G., Radchuk, V., Rohwäder, M.-S., Stillman, R. A., Thiele, J. C., & Ayllón, D. (2020). The ODD Protocol for Describing Agent-Based and Other Simulation Models: A Second Update to Improve Clarity, Replication, and Structural Realism. *Journal of Artificial Societies and Social Simulation*, *23*(2), 7. https://doi.org/10.18564/jasss.4259

Holst, N., & Belete, G. F. (2015). Domain-specific languages for ecological modelling. *Ecological Informatics*, *27*, 26–38. https://doi.org/10.1016/j.ecoinf.2015.02.005

Jenny, M. (1990). Territorialität und Brutbiologie der FeldlercheAlauda arvensis in einer intensiv genutzten Agrarlandschaft. *Journal für Ornithologie*, *131*(3), 241–265. https://doi.org/10.1007/BF01640998

Katna, A., Thaker, M., & Vanak, A. T. (2023). How fast do landscapes change? A workflow to analyze temporal changes in human-dominated landscapes. *Landscape Ecology*, *38*(8), 2145–2155. https://doi.org/10.1007/s10980-023-01686-y

Kostková, M., Hlavinka, P., Pohanková, E., Kersebaum, K. C., Nendel, C., Gobin, A., Olesen, J. E., Ferrise, R., Dibari, C., Takáč, J., Topaj, A., Medvedev, S., Hoffmann, M. P., Stella, T., Balek, J., Ruiz-Ramos, M., Rodríguez, A., Hoogenboom, G., Shelia, V., … Trnka, M. (2021). Performance of 13 crop simulation models and their ensemble for simulating four field crops in Central Europe. *The Journal of Agricultural Science*, *159*(1–2), 69–89. https://doi.org/10.1017/S0021859621000216

Kühn, E., Musche, M., Harpke, A., Feldmann, R., & Settele, J. (2024). Tagfalter-Monitoring Deutschland: Auswertung 2005-2023. *Oedippus*, *42*, 12–45. https://www.ufz.de/export/data/6/298835_298188_Oedippus_42_klein.pdf

le Clech, S., van Bussel, L. G. J., Lof, M. E., de Knegt, B., Szentirmai, I., & Andersen, E. (2024). Effects of linear landscape elements on multiple ecosystem services in contrasting agricultural landscapes. *Ecosystem Services*, *67*, 101616. https://doi.org/10.1016/j.ecoser.2024.101616

Lenda, M., & Skórka, P. (2010). Patch occupancy, number of individuals and population density of the Marbled White in a changing agricultural landscape. *Acta Oecologica*, *36*(5), 497–506. https://doi.org/10.1016/j.actao.2010.07.002

Marrec, R., Brusse, T., & Caro, G. (2022). Biodiversity-friendly agricultural landscapes – integrating farming practices and spatiotemporal dynamics. *Trends in Ecology & Evolution*, *37*(9), 731–733. https://doi.org/10.1016/j.tree.2022.05.004

McIntire, E. J. B., Chubaty, A. M., Cumming, S. G., Andison, D., Barros, C., Boisvenue, C., Haché, S., Luo, Y., Micheletti, T., & Stewart, F. E. C. (2022). PERFICT: A Re-imagined foundation for predictive ecology. *Ecology Letters*, *25*(6), 1345–1351. https://doi.org/10.1111/ele.13994

Meier, L., Grimm, V., & Frank, K. (2025). Model perpetuation by designing and documenting models and workflows so that they can be reused and further developed by others: The case of multiple stressors in ecology. *Ecological Modelling*, *501*, 111029. https://doi.org/10.1016/j.ecolmodel.2025.111029

Mialyk, O., Schyns, J. F., Booij, M. J., Su, H., Hogeboom, R. J., & Berger, M. (2024). Water footprints and crop water use of 175 individual crops for 1990–2019 simulated with a global crop model. *Scientific Data*, *11*(1), 206. https://doi.org/10.1038/s41597-024-03051-3

Miguet, P., Gaucherel, C., & Bretagnolle, V. (2013). Breeding habitat selection of Skylarks varies with crop heterogeneity, time and spatial scale, and reveals spatial and temporal crop complementation. *Ecological Modelling*, *266*, 10–18. https://doi.org/10.1016/j.ecolmodel.2013.06.029

mundialis GmbH & Co. KG. (2021). *Landcover classification map of Germany 2020 based on Sentinel-2 data* (Geoscientific Information). Geoscientific Information. Retrieved August 5, 2025, from https://data.mundialis.de/geonetwork/srv/eng/catalog.search#/metadata/9246503f-6adf-460b-a31e-73a649182d07

Mupepele, A.-C., Bruelheide, H., Brühl, C., Dauber, J., Fenske, M., Freibauer, A., Gerowitt, B., Krüß, A., Lakner, S., Plieninger, T., Potthast, T., Schlacke, S., Seppelt, R., Stützel, H., Weisser, W., Wägele, W., Böhning-Gaese, K., & Klein, A.-M. (2021). Biodiversity in European agricultural landscapes: Transformative societal changes needed. *Trends in Ecology & Evolution*, *36*(12), 1067–1070. https://doi.org/10.1016/j.tree.2021.08.014

Nystrom, R. (2014). *Game programming patterns*. genever benning.

Pe'er, G., Bonn, A., Bruelheide, H., Dieker, P., Eisenhauer, N., Feindt, P. H., Hagedorn, G., Hansjürgens, B., Herzon, I., Lomba, Â., Marquard, E., Moreira, F., Nitsch, H., Oppermann, R., Perino, A., Röder, N., Schleyer, C., Schindler, S., Wolf, C.,

... Lakner, S. (2020). Action needed for the EU Common Agricultural Policy to address sustainability challenges (K. Gaston, Ed.). *People and Nature*, *2*(2), 305–316. https://doi.org/10.1002/pan3.10080

Pe'er, G., Finn, J. A., Díaz, M., Birkenstock, M., Lakner, S., Röder, N., Kazakova, Y., Šumrada, T., Bezák, P., Concepción, E. D., Dänhardt, J., Morales, M. B., Rac, I., Špulerová, J., Schindler, S., Stavrinides, M., Targetti, S., Viaggi, D., Vogiatzakis, I. N., & Guyomard, H. (2022). How can the European Common Agricultural Policy help halt biodiversity loss? Recommendations by over 300 experts. *Conservation Letters*, *n/a*(n/a), e12901. https://doi.org/10.1111/conl.12901

Pe'er, G., Zinngrebe, Y., Hauck, J., Schindler, S., Dittrich, A., Zingg, S., Tscharntke, T., Oppermann, R., Sutcliffe, L. M., Sirami, C., Schmidt, J., Hoyer, C., Schleyer, C., & Lakner, S. (2017). Adding Some Green to the Greening: Improving the EU's Ecological Focus Areas for Biodiversity and Farmers: Evaluation of EU's ecological focus areas. *Conservation Letters*, *10*(5), 517–530. https://doi.org/10.1111/conl.12333

Poulsen, J. G., Sotherton, N. W., & Aebischer, N. J. (1998). Comparative nesting and feeding ecology of skylarks Alauda arvensis on arable farmland in southern England with special reference to set-aside. *Journal of Applied Ecology*, *35*(1), 131–147. https://doi.org/10.1046/j.1365-2664.1998.00289.x

Püttmanns, M., Böttges, L., Filla, T., Lehmann, F., Martens, A. S., Siegel, F., Sippel, A., von Bassi, M., Balkenhol, N., Waltert, M., & Gottschalk, E. (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

Püttmanns, M., Lehmann, F., Willert, F., Heinz, J., Kieburg, A., Filla, T., Balkenhol, N., Waltert, M., & Gottschalk, E. (2022). No seasonal curtailment of the Eurasian Skylark's (Alauda arvensis) breeding season in German heterogeneous farmland. *Ecology and Evolution*, *12*(9), e9267. https://doi.org/10.1002/ece3.9267

Pywell, R. F., Heard, M. S., Bradbury, R. B., Hinsley, S., Nowakowski, M., Walker, K. J., & Bullock, J. M. (2012). Wildlife-friendly farming benefits rare birds, bees and plants. *Biology Letters*, *8*(5), 772–775. https://doi.org/10.1098/rsbl.2012.0367

Raes, D., Steduto, P., Hsiao, T. C., & Fereres, E. (2009). AquaCrop—The FAO Crop Model to Simulate Yield Response to Water: II. Main Algorithms and Software Description. *Agronomy Journal*, *101*(3), 438–447. https://doi.org/10.2134/agronj2008.0140s

Reichenau, T. G., Korres, W., Schmidt, M., Graf, A., Welp, G., Meyer, N., Stadler, A., Brogi, C., & Schneider, K. (2020). A comprehensive dataset of vegetation states, fluxes of matter and energy, weather, agricultural management, and soil properties from intensively monitored crop sites in western Germany. *Earth System Science Data*, *12*(4), 2333–2364. https://doi.org/10.5194/essd-12-2333-2020

Reidsma, P., Janssen, S., Jansen, J., & van Ittersum, M. K. (2018). On the development and use of farm models for policy impact assessment in the European Union – A review. *Agricultural Systems*, *159*, 111–125. https://doi.org/10.1016/j.agsy.2017.10.012

Reinhardt, R., Sbieschne, H., Settele, J., Fischer, U., & Fiedler, G. (2007). *Tagfalter von Sachsen*. Entomofauna Saxonica.

Reinhardt, R., Harpke, A., Caspari, S., Dolek, M., Kühn, E., Musche, M., Trusch, R., Wiemers, M., & Settele, J. (2021). *Verbreitungsatlas der Tagfalter und Widderchen Deutschlands* (1., korrigierter Nachdruck). Eugen Ulmer KG.

Rigal, S., Dakos, V., Alonso, H., Auniņš, A., Benkő, Z., Brotons, L., Chodkiewicz, T., Chylarecki, P., de Carli, E., del Moral, J. C., Domşa, C., Escandell, V., Fontaine, B., Foppen, R., Gregory, R., Harris, S., Herrando, S., Husby, M., Ieronymidou, C., … Devictor, V. (2023). Farmland practices are driving bird population decline across Europe. *Proceedings of the National Academy of Sciences*, *120*(21), e2216573120. https://doi.org/10.1073/pnas.2216573120

Röder, N., Krämer, C., Grajewski, R., Lakner, S., & Matthews, A. (2024). What is the environmental potential of the post-2022 common agricultural policy? *Land Use Policy*, *144*, 107219. https://doi.org/10.1016/j.landusepol.2024.107219

Ropella, G. E., Railsback, S. F., & Jackson, S. K. (2002). Software Engineering Considerations for Individual-Based Models. *Natural Resource Modeling*, *15*(1), 5–22. https://doi.org/10.1111/j.1939-7445.2002.tb00077.x

Roy, D. B., Rothery, P., Moss, D., Pollard, E., & Thomas, J. A. (2001). Butterfly numbers and weather: Predicting historical trends in abundance and the future effects of climate change. *Journal of Animal Ecology*, *70*(2), 201–217. https://doi.org/10.1111/j.1365-2656.2001.00480.x

Runhaar, H. (2021). Four critical conditions for agroecological transitions in Europe. *International Journal of Agricultural Sustainability*, *19*(3–4), 227–233. https://doi.org/10.1080/14735903.2021.1906055

Schellhorn, N. A., Gagic, V., & Bommarco, R. (2015). Time will tell: Resource continuity bolsters ecosystem services. *Trends in Ecology & Evolution*, *30*(9), 524–530. https://doi.org/10.1016/j.tree.2015.06.007

Seppelt, R., Arndt, C., Beckmann, M., Martin, E. A., & Hertel, T. W. (2020). Deciphering the Biodiversity–Production Mutualism in the Global Food Security Debate. *Trends in Ecology & Evolution*, *35*(11), 1011–1020. https://doi.org/10.1016/j.tree.2020.06.012

Steduto, P., Hsiao, T. C., Raes, D., & Fereres, E. (2009). AquaCrop—The FAO Crop Model to Simulate Yield Response to Water: I. Concepts and Underlying Principles. *Agronomy Journal*, *101*(3), 426–437. https://doi.org/10.2134/agronj2008.0139s

Stillman, R. A., Railsback, S. F., Giske, J., Berger, U., & Grimm, V. (2015). Making Predictions in a Changing World: The Benefits of Individual-Based Ecology. *BioScience*, *65*(2), 140–150. https://doi.org/10.1093/biosci/biu192

Sun, Z., Lorscheid, I., Millington, J. D., Lauf, S., Magliocca, N. R., Groeneveld, J., Balbi, S., Nolzen, H., Müller, B., Schulze, J., & Buchmann, C. M. (2016). Simple or complicated agent-based models? A complicated issue. *Environmental Modelling & Software*, *86*, 56–67. https://doi.org/10.1016/j.envsoft.2016.09.006

Topping, C. J., Hansen, T. S., Jensen, T. S., Jepsen, J. U., Nikolajsen, F., & Odderskær, P. (2003). ALMaSS, an agent-based model for animals in temperate European landscapes. *Ecological Modelling*, *167*(1), 65–82. https://doi.org/10.1016/S0304-3800(03)00173-X

Topping, C. J., Dalby, L., & Valdez, J. W. (2019). Landscape-scale simulations as a tool in multi-criteria decision making to support agri-environment schemes. *Agricultural Systems*, *176*, 102671. https://doi.org/10.1016/j.agsy.2019.102671

Topping, C. J. (2022). The Animal Landscape and Man Simulation System (ALMaSS): A history, design, and philosophy. *Research Ideas and Outcomes*, *8*, e89919. https://doi.org/10.3897/rio.8.e89919

Topping, C. J., & Duan, X. (2024). ALMaSS Landscape and Farming Simulation: Software classes and methods. *Food and Ecological Systems Modelling Journal*, *5*, e121215. https://doi.org/10.3897/fmj.5.121215

Troost, C., Huber, R., Bell, A. R., Van Delden, H., Filatova, T., Le, Q. B., Lippe, M., Niamir, L., Polhill, J. G., Sun, Z., & Berger, T. (2023). How to keep it adequate: A protocol for ensuring validity in agent-based simulation. *Environmental Modelling & Software*, *159*, 105559. https://doi.org/10.1016/j.envsoft.2022.105559

Tscharntke, T., Klein, A. M., Kruess, A., Steffan-Dewenter, I., & Thies, C. (2005). Landscape perspectives on agricultural intensification and biodiversity - Ecosystem service management. *Ecology Letters*, *8*(8), 857–874. https://doi.org/10.1111/j.1461-0248.2005.00782.x

Uchmański, J., & Grimm, V. (1996). Individual-based modelling in ecology: What makes the difference? *Trends in Ecology & Evolution*, *11*(10), 437–441. https://doi.org/10.1016/0169-5347(96)20091-6

Vandewoestijne, S., Martin, T., Liégeois, S., & Baguette, M. (2004). Dispersal, landscape occupancy and population structure in the butterfly *Melanargia galathea*. *Basic and Applied Ecology*, *5*(6), 581–591. https://doi.org/10.1016/j.baae.2004.07.004

van Swaay, C., Schmucki, R., Roy, D., Dennis, E., Collins, S., Fox, R., Kolev, Z. D., G. Sevilleja, C., Warren, M. S., Whitfield, A., Wynhoff, I., Arnberg, H. J., Balalaikins, M., Barea, J. M., Boe, A. M. B., Bonelli, S., Botham, M. S., Bourn, N. A., Cancela, J. P., ... Zografou, K. (2025, July 21). *EU Grassland Butterfly Index 1991-2023* (Technical report). Butterfly Conservation Europe & EMBRACE/eBMS (www.butterfly-monitoring.net) & Vlinderstichting report VS2025.014. https://doi.org/10.5281/zenodo.16367397

Vasseur, C., Joannon, A., Aviron, S., Burel, F., Meynard, J.-M., & Baudry, J. (2013). The cropping systems mosaic: How does the hidden heterogeneity of agricultural landscapes drive arthropod populations? *Agriculture, Ecosystems & Environment*, *166*, 3–14. https://doi.org/10.1016/j.agee.2012.08.013

Vedder, D., Ankenbrand, M., & Cabral, J. S. (2021). Dealing with software complexity in individual-based models. *Methods in Ecology and Evolution*, *12*(12), 2324–2333. https://doi.org/10.1111/2041-210X.13716

Vedder, D., Fischer, S. M., Wiegand, K., & Pe'er, G. (2024). Developing multidisciplinary mechanistic models: Challenges and approaches. *Socio-Environmental Systems Modelling*, *6*, 18701. https://doi.org/10.18174/sesmo.18701

Vedder, D., Matthies, M. C., Díaz Iturry, G., & Pe'er, G. (2025, August 29). *Persefone.jl* (Version 0.8). Leipzig, Germany. https://doi.org/10.5281/zenodo.16993216

Velten, S., Kewes, C., Brudler, R., Marsden, K., & Theilen, G. (2023). Multi-level Exchange Platforms for Biodiversity Conservation in Agricultural Landscapes. *Social Innovations Journal*, *22*. Retrieved January 9, 2024, from https://socialinnovationsjournal.com/index.php/sij/article/view/6968

Vickery, J. A., Bradbury, R. B., Henderson, I. G., Eaton, M. A., & Grice, P. V. (2004). The role of agri-environment schemes and farm management practices in reversing the decline of farmland birds in England. *Biological Conservation*, *119*(1), 19–39. https://doi.org/10.1016/j.biocon.2003.06.004

Will, M., Dressler, G., Kreuer, D., Thulke, H.-H., Grêt-Regamey, A., & Müller, B. (2021). How to make socio-environmental modelling more useful to support policy and management? *People and Nature*, *00*, 1–13. https://doi.org/10.1002/pan3.10207

Persefone.jl User Manual

March 5, 2026

v1.0.0

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.

Chapter 1

Graphical summary

Preprint: Vedder, D., Matthies, M. C., Iturry, G. D., & Pe'er, G. (2025). Persefone.jl: Modelling Biodiversity in Dynamic Agricultural Landscapes. doi:10.32942/X20S8K

*This documentation was last updated on 2026-03-05 for **Persefone.jl v1.0.0** (commit bb93f6d).*

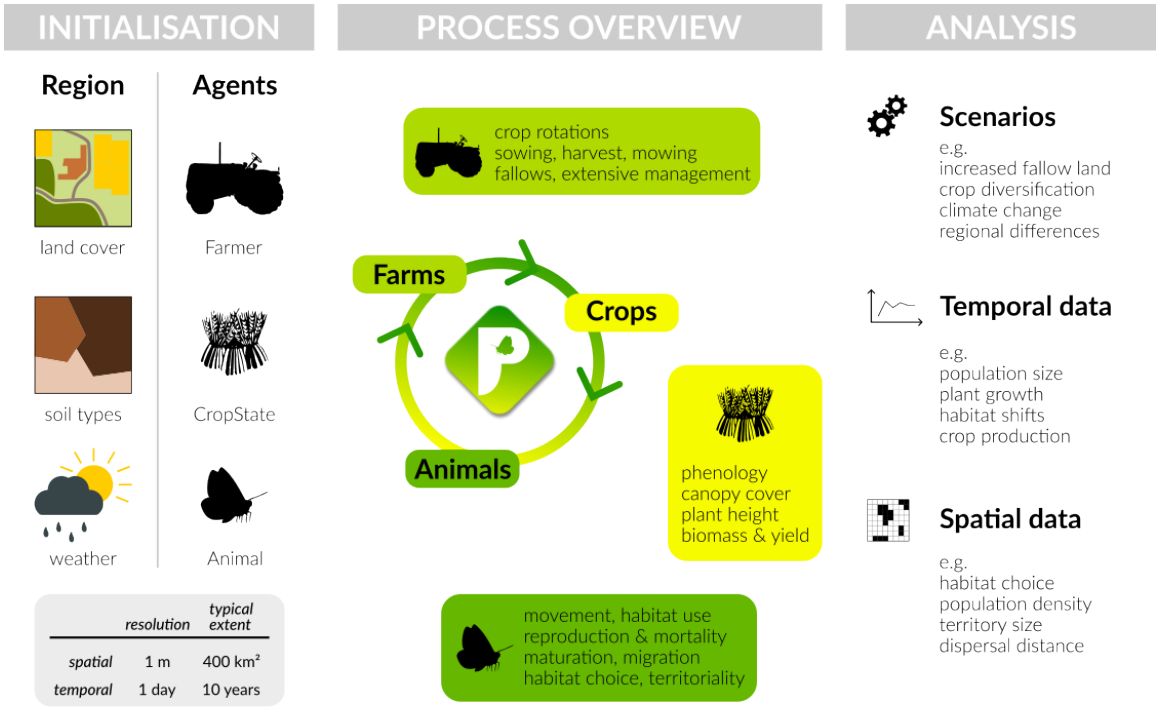

Figure 1.1: Summary graphic

Part II

User guide

Chapter 2

The Persefone.jl Package

Available user interfaces

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.

2.1 Installation

For more detailed installation instructions, see here.

Install the latest version of the Julia programming language (1.12+). The recommended editors are VSCode or Emacs. To install the package dependencies, open a Julia REPL in the repository directory and run:

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

2.2 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 <config>
```

where `<config>` specifies the configuration file to use. Make a copy of `src/parameters.toml` and edit this 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 [-s SEED] [-o OUTDIR] [-l LOGLEVEL] [--version] [-h]
              [configfile]

positional arguments:
  configfile            name of the configuration file

optional arguments:
  -s, --seed SEED       inital random seed (type: Int64)
  -o, --outdir OUTDIR   location of the output directory
```

```
  -l, --loglevel LOGLEVEL
                        verbosity: "debug", "info", or "warn"
  --version             show version information and exit
  -h, --help            show this help message and exit
```

You can also use `runparallel.jl` to launch a simulation experiment using multiple processors (when parameter scanning), or `runprofile.jl` to profile the performance of the software.

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 regions   # download and preprocess regional input data
```

2.3 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 3

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.

Out of date

Please note that Persefone Desktop has not yet been updated to work with model versions newer than Persefone.jl 0.8.

3.1 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.
2. Start Julia. This should launch a commandline interface/REPL.
3. Execute the following commands (copy-and-paste should work):

```
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()
```

3.2 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:

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

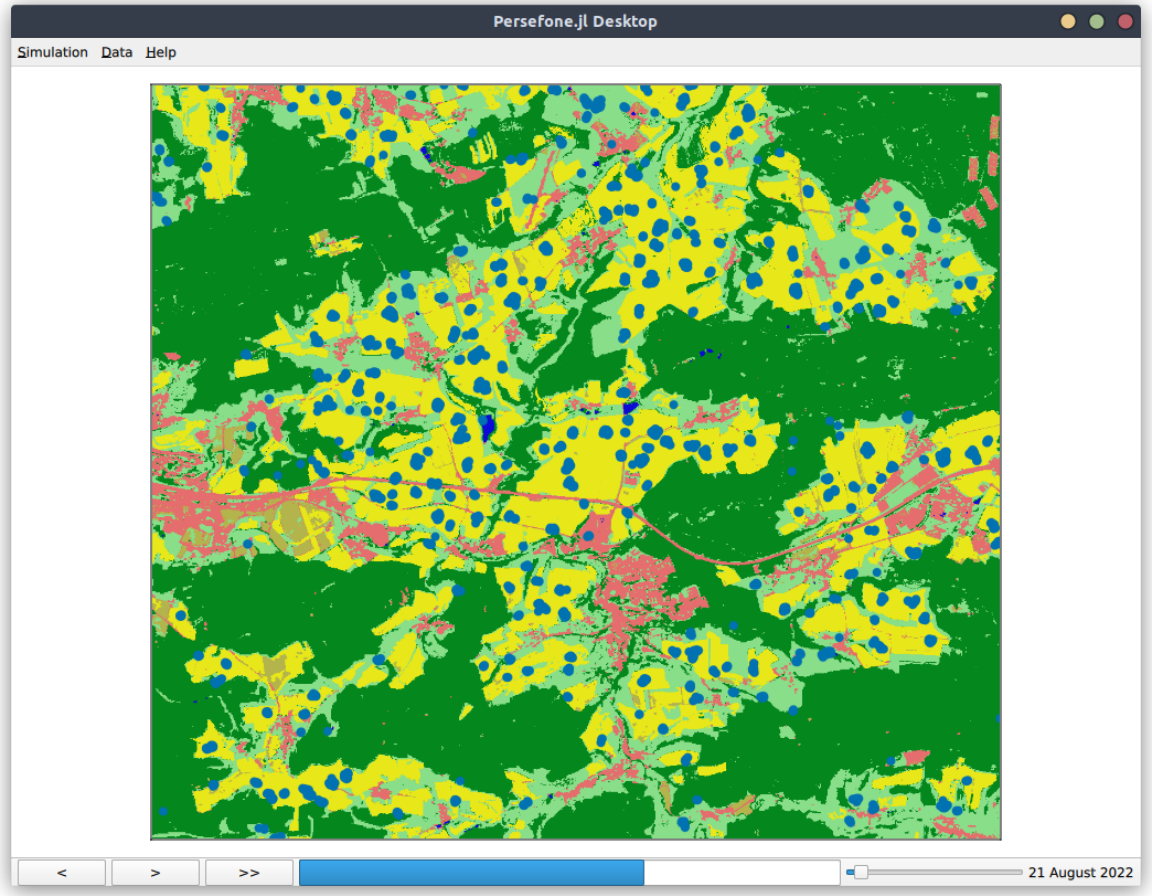

Figure 3.1: Persefone.jl Desktop screenshot

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

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

Start-up time

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

3.3 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 4

Configuration

The configuration file provides a central place for all simulation-related parameters. To set up a new simulation experiment, modify a copy of this file and pass it to the software (e.g. using `./run.jl -c <config>`). This is what the default config file looks like:

```
### 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/none/both
csvoutput = true                   # save collected data in CSV files
visualise = true                   # generate result graphs
storedata = true                   # keep collected data in memory
figureformat = "pdf"               # file format to use for graphical output
loglevel = "info"                  # verbosity level: "debug", "info", "warn"
seed = 2                           # seed value for the RNG (0 -> random value)
startdate = 2022-01-01             # first day of the simulation
enddate = 2023-12-31               # last day of the simulation

[world]
region = "jena" # the region to simulate (must be a folder in `mapdirectory`)
mapdirectory = "data/regions"   # the directory in which all geographic data are stored
landscapemap = "landscape.gpkg" # name of the landscape map in the map directory
weatherfile = "weather.csv"     # name of the weather data file in the map directory

[farm]
farmmodel = "BasicFarmer" # which version of the farm model to use
setaside = 0.04           # proportion of farm area set aside as fallow
croprotation = ["winter wheat", "winter rape", "silage maize", "winter barley"]
extensivegrassland = 0.60 # proportion of grassland managed extensively
mowingthreshold = 25      # height in cm above which intensive grassland is mown
mowingperiod = 10         # number of days in which mowing may occur on extensive grassland
fieldoutfreq = "daily"    # output frequency for crop/field data, daily/monthly/yearly/end/never
scenarios = ["thuringian_fallows"] # management scenarios to apply
```

```
[animal]
animaldirectory = "data/animals"
targetspecies = ["Skylark", "MarbledWhite"] # list of target species to simulate
popoutfreq = "daily"   # output frequency population-level data, daily/monthly/yearly/end/never
indoutfreq = "monthly" # output frequency individual-level data, daily/monthly/yearly/end/never
showhomeranges = true  # plot home ranges on output maps?

[crop]
cropmodel = "almass,aquacrop" # crop growth model to use, "almass", "aquacrop", or "simple"
cropdirectory = "data/crops/almass/,data/crops/aquacrop/" # the directory storing all data files for
↪  the selected crop model
```

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). Use `runparallel.jl` to have these simulations run on multiple processors.

Note that species-specific parameters are stored in the `data/crops` and `data/animals` directories. For instructions on how to create or modify these, see the crop calibration tutorial and the animal model documentation.

Part III

Developer guide

Chapter 5

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 <configfile>`.

Changing the region

To apply Persefone to a new region, you need to create new a new input map giving land cover and soil type information, as well as providing local weather data. (Within Germany, these steps are largely automated.) How to do so is described here.

Adding new animal species

To add a new animal species, familiarise yourself with the architecture of the animal submodel. Adding a new species to an existing species group can be done by adding a TOML file to the `data/animals` folder and setting the appropriate parameter values. To implement a new species group, use the `@create` and `@phase` macros.

Adding new crop species

To calibrate the AquaCrop crop growth model for new crop species, follow the tutorial here.

Adding new farmer behaviour or a new crop model

To implement new farmer behaviour or add another crop model, create a new subtype of Farmer or AbstractCropState, respectively. As this is somewhat more complex, read through the current implementations of the farm and crop components to understand how they work.

Adding a new submodel

To add a new submodel in addition to the existing ones (`animal`, `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 6

Developing Persefone

6.1 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`.

6.2 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 Gitlab 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. Update `CHANGELOG.md` to keep track of your changes, and plan releases as appropriate. (See semantic versioning and keep a changelog.

7. Once you're 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.

8. Repeat :-)

The Gitlab issue tracker can be used to create, discuss, and assign tasks, as well as to monitor progress towards milestones/releases.

6.3 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 docstrings 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> 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 7

Source code architecture

Chapter 8

Model components

The Persefone source code is divided into five components, three of which we call "submodels":

1. `core`: This provides the foundation of the model software, and sets up and executes simulation runs. It also reads the configuration file and provides data output functionality.
2. `world`: This manages the environment, i.e. the landscape (including reading in map data and providing spatial operations) and the weather.
3. `farm`: This is an agent-based model of farmer decision making. It provides the `Farmer` agent type, which can be subtyped to provide different decision models. Currently the only implemented farmer is very basic, only carrying out a static crop rotation and simple grassland mowing regimes.
4. `crop`: This component simulates the growth of crops in the landscape. It provides the agent type `AbstractCropState`, representing the crop plants growing on one field. Currently two different models are used here: AquaCrop for the most important crops, and ALMaSS for the rest.
5. `animal`: This is a collection of 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.

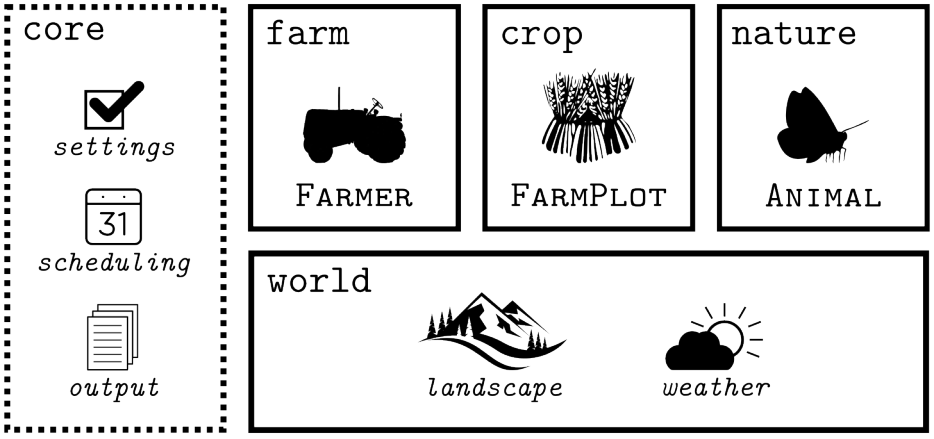

Figure 8.1: model architecture

Conceptually, `core` and `world` provide functionality that is needed by all the submodels. Decisions made by `Farmers` affect the `AstractCropStates` on their fields, and (directly or indirectly) the `Animals` in the model landscape.

Chapter 9

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 spatially-index collection of `LandscapeElement` 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

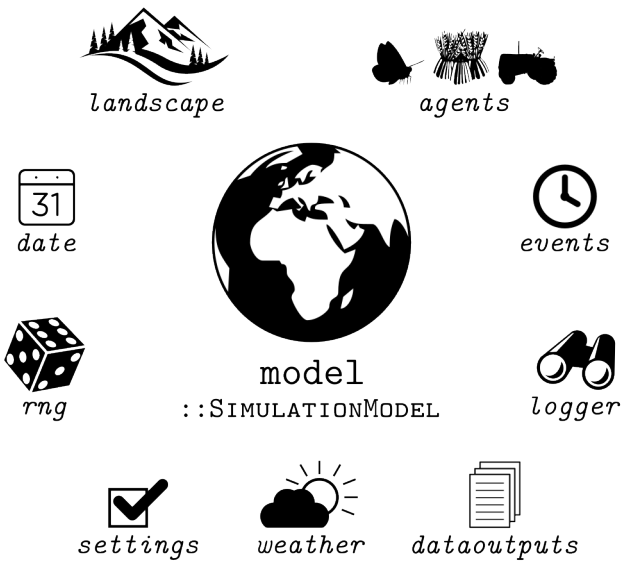

Figure 9.1: the model object

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 a `model` object is actually available. This is the case for most functions in Persefone, but not for all. Similarly, most other macros in the Persefone.jl package depend on the `model` object being available in-scope (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/animal/ecologicaldata.jl`. (See `src/core/output.jl` for details.)

Farm events

The `FarmEvent` struct is used to communicate farming-related events between submodels (e.g. sowing or harvest). An event can be triggered with `createevent!` and affects one `LandscapeElement`. (See `src/world/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 10

Maps and weather data

To simulate a region, Persefone.jl requires different input files:

1. A map file in the GeoPackage format, containing land cover polygons with an associated soil type.
2. A CSV file containing daily weather data for the simulated time period.
3. Regionally calibrated crop parameters for the AquaCrop model (if available).

How to calibrate AquaCrop is explained in detail here. If there are no regional parameters available, Persefone.jl will instead instantiate AquaCrop with national-level parameters.

This page will describe how the map and weather data are structured, obtained, and processed. All region data files are stored using the following convention:

```
data/regions/<regionname>/
-> <regionname>.geojson
-> landscape.gpkg
-> weather.csv
```

Where `<regionname>` is currently one of `bodensee`, `eichsfeld`, `hohenlohe`, `jena`, `oberrhein`, or `thueringer_becken`.

Info

There is a QGIS project file at `data/regions/auxiliary/persefone.qgz`, which can be used get an overview of the existing region boundary files and create new ones.

10.1 Generating input data

The process of downloading and processing map and weather data for the existing regions is fully automated. Simply execute `make regions`, or run the script `data/regions/auxiliary/createinputs.sh`.

If you only need to generate an individual region, or only weather or map data, you can use the scripts in the `data/regions/auxiliary` folder: `extract_weather_data.R` and `vector_maps.jl`. Note that generating map files can take a long time!

To add a new region within Germany, you must create a GeoJSON file containing a single rectangle delineating the landscape extent, and add the relevant details for the region to the two scripts listed above. Adding a region outside Germany is possible too, but in this case the input files must be created manually.

10.2 Land cover maps

Persefone.jl works with vector maps in which each geographical element is assigned to one of 32 land cover types:

```
@enum LandCover begin
    NoData                # 0
    ResidentialBuildings  # 1
    IndustrialBuildings   # 2
    PublicBuildings       # 3
    OtherBuildings        # 4
    IndustrialArea        # 5
    ParkOrGarden          # 6
    Arable                # 7
    Grassland             # 8
    Orchard               # 9
    Horticulture          # 10
    FruitPlantation       # 11
    ConiferousForest      # 12
    BroadleavedForest     # 13
    MixedForest           # 14
    TreePlantation        # 15
    Hedge                 # 16
    Scrub                 # 17
    Trees                 # 18
    Reeds                 # 19
    BareGround            # 20
    Heath                 # 21
    Peatland              # 22
    Swamp                 # 23
    RockOrCliff           # 24
    SlopeOrDike           # 25
    Stream                # 26
    Lake                  # 27
    Ocean                 # 28
    Railway               # 29
    Path                  # 30
    Road                  # 31
end
```

While these maps could be created by hand using normal GIS software, it is easier to generate them from existing official cartographical maps. For Germany, these are available in the ATKIS data provided by each state (e.g. Baden-Württemberg). These maps cover all of Germany seamlessly with an accuracy of +/- 3m and are updated regularly. However, they use a complicated classification scheme that goes beyond the requirements of Persefone.jl. Therefore, the `vector_maps.jl` script takes these data, crops them to the desired extent, and converts the ATKIS classification to the land cover types listed above. The resulting GeoPackage file can then be copied into the respective region folder and used as input to the model.

10.3 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.

The required map can be downloaded here. Its integer values map onto the Persefone.jl `SoilType` enum as follows:

```
1: Abbauflächen -> nosoil
2: Gewässer -> nosoil
3: Lehmsande (ls) -> loamy_sand
4: Lehmschluffe (lu) -> silt_loam
5: Moore -> nosoil
6: Normallehme (ll) -> loam
7: Reinsande (ss) -> sand
8: Sandlehme (sl) -> sandy_loam
9: Schluffsande (us) -> sandy_loam
10: Schlufftone (ut) -> silty_clay
11: Siedlung -> nosoil
12: Tonlehme (tl) -> clay_loam
13: Tonschluffe (tu) -> silty_clay_loam
14: Watt -> nosoil
```

The `vector_maps.jl` script will take this map and extract all relevant data, so that the resulting GPKG file contains soil as well as land cover data.

Names of soil types are based on the relative composition of clay, silt, and sand. Note that the typology used in the Bodenatlas does not map perfectly on to this international classification. Image source: Australian Environmental Education

10.4 Weather data

Currently, Persefone uses historical weather data from the closest 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 description of these data sets 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. The currently included regions have the following station codes:

- **Region Jena:** station number 02444 ("Jena (Sternwarte)")
- **Region Eichsfeld:** station number 02925 ("Leinefelde")
- **Region Thüringer Becken:** station number 00896 ("Dachwig")
- **Region Hohenlohe:** station number 03761 ("Oehringen")
- **Region Bodensee:** station number 06263 ("Singen")
- **Region Oberrhein:** station number 05275 ("Waghäusel-Kirrlach")

The `extract_weather_data.R` script can be used to download and process the data into the format needed by Persefone. This uses the `rdwd` package. To use it, simply specify the desired `region`, adding its ID to the `stationid` list if necessary. The produced CSV file can be copied into the respective region folder.

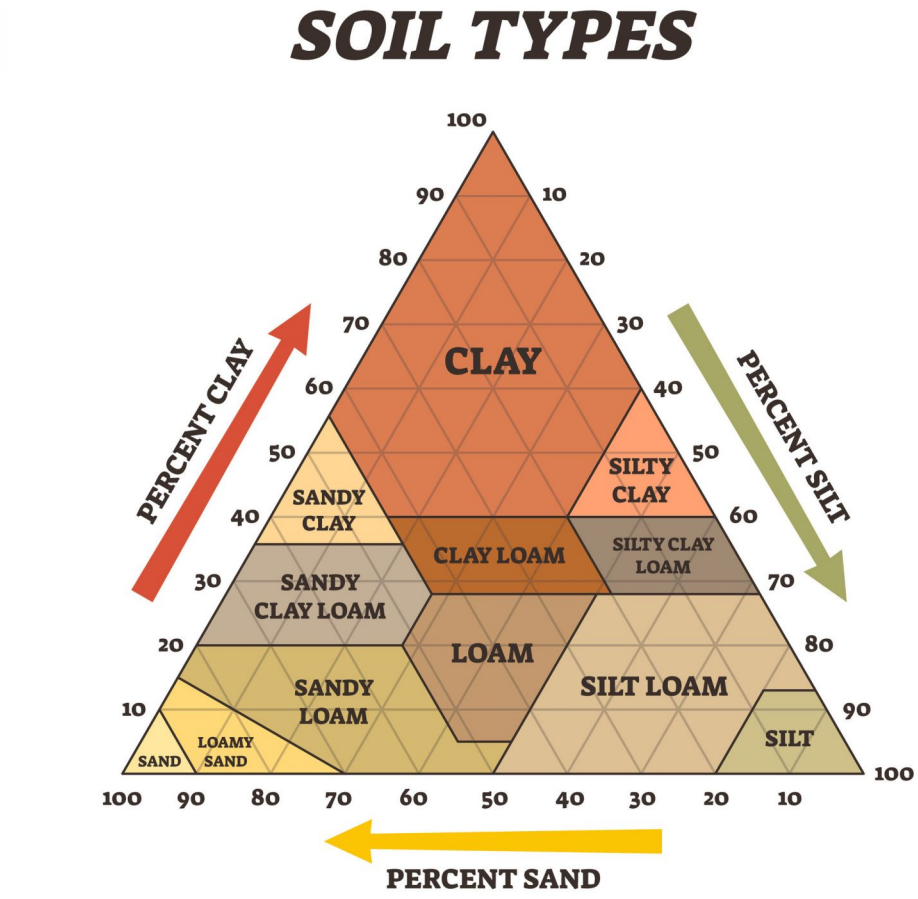

Figure 10.1: Soil types triangle

Chapter 11

Defining new species

11.1 Code structure of the animal submodel

An important aim of Persefone is to make adding new species as easy as possible, while simultaneously allowing full flexibility for modelling differences in behaviour and life cycles. To achieve this, the animal submodel works with three separate layers of code, representing different degrees of generality.

The bottom-most layer consists of functions that are needed by all species (e.g. initialisation) or that can potentially be used by many different species (e.g. movement). These are contained in the files `src/animal/individuals.jl` and `src/animal/populations.jl`. Many of these have macros available for convenient use (defined in `src/animal/macros.jl`).

The middle layer is the taxon layer. **Taxon** in this case is not a strictly scientific term in the sense of a phylogenetic clade, but rather refers to a group of species that have similar life cycles and behavioural patterns and can thus be modelled using largely the same code. At the moment, the two implemented taxa are farmland birds and grassland butterflies (in `src/animal/birds.jl` and `src/animal/butterflies.jl`). For each taxon, the animals' behaviour during different parts of the life cycle is implemented using the `@phase` macro (see below for details). A **phase** represents a period of the individual's life that is marked by specific behavioural patterns (e.g. as a larva, or when nesting).

The third and highest level of the animal submodel is the **species** layer. While behavioural rules are best represented using code, many other aspects of a species' ecology can be expressed as parameters (e.g. date of return from migration, or mean number of eggs per nest). It is assumed that species belonging to the same taxon (as defined above) will share the same behavioural rules, but differ in the parameter values that modify these rules. Following the general programming rule that data should be separated from logic, these species-specific parameters are therefore read in from TOML configuration files, stored in the `data/animals` directory. The `animal.targetspecies` configuration parameter selects which of these files to load during a model run.

Taking all this together, we see that adding a new species can be very simple, if it is a member of an already implemented taxon. In this case, one only needs to duplicate one of the existing species parameter files and change the parameter values to suit the new species. At the same, if the species shows behaviour that is not yet implemented in any taxon, one has the flexibility of either expanding the code for an existing taxon (if there is one that is similar enough), or creating a new taxon tailored to the life cycle of the new species one wants to simulate.

Summary

Behavioural logic is implemented on a taxon level using the `@phase` macro. Parameter values modifying this behaviour are defined for each species using TOML files stored in the `data/animals` directory.

11.2 The @phase macro

Let's have a look at this on a more technical level. Here is an example of a hypothetical mermaid taxon:

```
@create Mermaid begin
    self.daysofdrought = 0
    @debug "Created $(animalid(self))."
end

@phase Mermaid life begin
    @debug "$(animalid(self)) is swimming happily in a $(@landcover())."
    @respond pesticide @kill(self.pesticidemortality, "poisoning")
        if @precipitation() == 0
                self.daysofdrought += 1
                if self.daysofdrought > self.droughttolerance
                        @setphase("drought")
                end
        elseif self.daysofdrought > 0
                self.daysofdrought -= 1
        end
    if self.sex == female && @landcover() == WATER && chance(0.01)
        @reproduce()
    end
end

@phase Mermaid drought begin
    @debug "$(animalid(self)) is experiencing drought."
        if @precipitation() > 0
                self.daysofdrought -= 1
                @setphase("life")
        end
end

requireparams!(:taxon, "Mermaid", [:pesticidemortality, :droughttolerance])
addspeciesparam!(:pesticidemortality, "Chance of a Mermaid dying from pesticide exposure", Float64)
addspeciesparam!(:droughttolerance, "Days without rain before a Mermaid goes into drought phase",
↪  Int)
```

As stated above, the most important thing here is that each taxon must implement one or more `@phase` blocks, defining its behaviour during a certain phase of life. Under the hood, `@phase` creates a function that will be called daily, so long as the individual's phase variable is set to this phase. Code within this function has access to the `model` object (which holds all runtime data) as well as a `self` object, which is the currently active `Animal` agent. Using `self`, the code can access any parameter values defined in the species files. 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 (see `src/nature/macros.jl`).

A special phase is defined using the `@create` macro. This is purely a utility wrapper - `@create Mermaid` is equivalent to `@phase Mermaid create`. This phase is called when initialising an individual at the start of a model run or when it is born, and can be used to instantiate life-time variables (like `self.daysofdrought` in the example above).

Read the source

The simplest way to understand how this all works is probably to read the source code of the already implemented taxa. If you have questions, ask the Persefone developers for help.

11.3 Species parameters

So that the model can check at start-up whether all species files include all needed parameters, the animal submodel provides an internal documentation mechanism. This is demonstrated in the final lines of the example above. Each taxon should use the `addspeciesparam!()` and `requirespeciesparams!()` functions to record which parameters it uses, and under which conditions these are required (as some parameters are only needed if another parameter has a certain value). These functions can also be used to specify lists of valid values, and conversion functions for converting types that cannot be natively parsed by TOML.

Here is the list of parameters that is always required for all species:

Parameter	Description
`species`	The official (Latin) name of the species.
`species_en`	The English name of this species.
`taxon`	The taxon that this species belongs to.
`migrate`	Whether this species migrates in winter or not.
`birthphase`	The phase assigned to individuals at birth.
`initialphase`	The phase assigned to individuals at model initialisation.
`initialiseas`	The sex assigned to individuals at model initialisation.
`initialdensity`	The habitat area per individual at initialisation (in ha).
`initialhabitat`	A HabitatExpression describing where this species can live.

Note that if `migrate` is `true`, several other parameters are also required, namely `migrationdeparture`, `migrationarrival`, `migrationdelayfemales`, and `migrationmortality`.

The function `checkspeciesdict()` can be used to check the consistency and completeness of a species parameter file. This will flag any wrong types or values, and output a list of parameters that ought to be defined but aren't (based on the given parameter values).

11.4 Habitat descriptors

A final feature of the animal submodel that still needs to be explained are the habitat descriptors. This is a string format that can be used to define which types of habitat are usable by species in different contexts. For example, the `initialhabitat` parameter in the table above expects such a descriptor for use during model initialisation.

Habitat descriptors are created using `parsehabitat()` and are stored internally as a `HabitatExpression`. As a short form, they can be created using the string macro `@h_str`, simply by prepending a habitat descriptor string with an h.

Here are some examples to demonstrate the salient properties:

```
parsehabitat("Fallow")
parsehabitat("Arable OR Grassland")
parsehabitat("Grassland OR (GRAIN AND NATURAL)")
h"WinterWheat AND height>5"
h"ExtensiveGrassland AND (height<=50 AND height>20)"
h"GRAIN AND (!SilageMaize AND !CornMaize)"
```

To explain: habitat descriptors consist of one or more expressions. Expressions are linked using the boolean operators `AND` or `OR`, with parentheses used to specify priority order. Each expression is either a single word, in which case it must be a `LandCover`, `LandCoverCategory`, `CropName`, or `CropGroup`. If it is a single word, a prepended `!` excludes the given value. Alternately, expressions can take the form `<aspect><comparator><value>`, where `<aspect>` is either `height` or `cover` (i.e. plant height / crop cover), and comparator is one of `==`, `!=`, `>`, `<`, `>=`, `<=`.

While such habitat descriptors can of course also be implemented in code (using, for example, the `@landcover()` or `@cropcover()` macros), having them as strings gives the advantage that they can be included in species parameter files, and makes them easier to read.

Part IV

Software API

Chapter 12

Simulation

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

12.1 Persefone.jl

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

`Persefone.MAPRESOLUTION` – Constant.

The spatial resolution of the currently used input maps.

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} or a String can be automatically converted to an AnnualDate, allowing this syntax: `nestingend::AnnualDate = (August, 15)`, or `nestingend::AnnualDate = "15 August"`.

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.camelcase` – Method.

```
camelcase(string)
```

Return a CamelCaseVersion of the string (each word capitalised and joined to the next).

source

`Persefone.cycle!` – Function.

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

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

source

`Persefone.snakecase` – Method.

```
snakecase(string)
```

Return a snake*case*version of the string (all small letters, words joined by underscores).

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.uncamelcase` – Method.

```
uncamelcase(string)
```

Reverse `camelcase()` (e.g. "WinterWheat" -> "Winter Wheat").

source

`Persefone.unsnakecase` – Method.

```
unsnakecase(string)
```

Reverse `snakecase()` (e.g. "winter_wheat" -> "winter wheat").

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.@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.@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.@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

`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

12.2 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 SimulationModel 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.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

12.3 landcover.jl

This file lists the land cover types represented on the model maps.

12.4 landscape.jl

This file manages the landscape maps that underlie the model.

`Persefone.HomeRange` – Type.

```
HomeRange
```

Container for home range / territory polygons.

source

`Persefone.Landscape` – Type.

```
Landscape
```

A vector-based landscape representation.

source

`Persefone.LandscapeElement` – Type.

```
LandscapeElement
```

Container for landscape features (landcover, soil, optional crop state).

source

`Persefone.SoilType` – Type.

The soil type of a LandscapeElement

source

`Persefone._closest_point_on_curve` – Method.

```
_closest_point_on_curve(pos, curve)
```

Return the closest point on a curve to pos and the squared distance. The closing segment is included if the curve is not explicitly closed.

source

Persefone._closest_point_on_geom – Method.

```
_closest_point_on_geom(pos, geom)
```

Return the closest point on the polygon boundary of `geom` to `pos` and the squared distance between them. Holes are included as candidates.

source

Persefone._closest_point_on_segment – Method.

```
_closest_point_on_segment(pos, a, b)
```

Return the closest point on the segment a-b to `pos` and the squared distance.

source

Persefone._geom_to_geoms – Method.

```
_geom_to_geoms(geom)
```

Convert an ArchGDAL geometry into one or more polygon geometries (splitting multipolygons).

source

Persefone.add_homerange! – Method.

```
add_homerange!(ls::Landscape, hr::HomeRange)
```

Adds a `HomeRange` to a `Landscape` and update the spatial index.

source

Persefone.add_land! – Method.

```
add_land!(ls::Landscape, le::LandscapeElement)
```

Adds a `LandscapeElement` to a `Landscape`, and assigns `le.id` a new `LandElemID` (therefore modifying both arguments).

source

Persefone.allelements – Method.

```
allelements(model)
```

Return all elements in the model landscape.

source

Persefone.coverage_stats – Method.

```
coverage_stats(polys, boundary; precision_eps=0.0)
```

Compute coverage, gap area, and overlap area for a set of polygon geometries within a boundary polygon.

source

Persefone.coverage_stats – Method.

```
coverage_stats(ls::Landscape; boundary=nothing, precision_eps=0.0)
```

Compute coverage statistics for a Landscape. If no boundary is provided, the landscape extent is used as a rectangular boundary.

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 – Function.

```
directionto(pos, habitatfunction, model, maxradius=1km, stepsize::Length=100m)
```

Calculate the direction from the given location to the closest landscape element matching the habitat descriptor function. Returns a bearing (0-359°) or nothing if no matching landscape element is found.

source

Persefone.distanceto – Function.

```
distanceto(pos, habitatfunction, model, maxradius=1km, stepsize::Length=100m)
```

Return the distance to the closest landscape element matching the habitat descriptor function. Is a thin wrapper around findnearest().

source

Persefone.distancetoedge – Method.

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

Calculate the distance from the given location to the edge of the local landscape element.

source

Persefone.findnearest – Function.

```
findnearest(pos, habitatfunction, model, maxradius=1km, stepsize::Length=100m)
```

Find the closest landscape element matching the habitat function, if there is one within the maxradius. Searches in concentric circles determined by stepsize. Returns a tuple of the found landscape element (or nothing) and the distance to it.

source

Persefone.gap_polygons – Method.

```
gap_polygons(polys, boundary; precision_eps=0.0)
```

Return a vector of gap polygons (boundary minus union of polys).

source

Persefone.gap_polygons – Method.

```
gap_polygons(ls::Landscape; boundary=nothing, precision_eps=0.0)
```

Return gap polygons for a Landscape. If no boundary is provided, the landscape extent is used as a rectangular boundary.

source

Persefone.get_homeranges_at_coord – Method.

```
get_homeranges_at_coord(landscape::Landscape, coord::Coord)
```

Find any HomeRanges containing the given coordinate. Uses the spatial index for fast candidate filtering, then precise geometric test.

Returns a vector of HomeRanges found (empty if none found).

source

Persefone.get_homeranges_in_extent – Method.

```
get_homeranges_in_extent(landscape::Landscape, ext::Extent2D)
```

Find all HomeRanges whose extents intersect with the given extent using the spatial index for efficient O(log n) lookup.

Returns a lazy iterator over HomeRange objects.

source

Persefone.get_land_elem_at_coord – Method.

```
get_land_elem_at_coord(landscape::Landscape, coord::Coord)
```

Find the LandscapeElement containing the given coordinate. Uses the spatial index for fast candidate filtering, then precise geometric test.

Returns the LandscapeElement if found, or nothing otherwise.

source

Persefone.get_land_elems_in_extent – Method.

```
get_land_elems_in_extent(landscape::Landscape, ext::Extent2D)
```

Find all LandscapeElements whose extents intersect with the given extent using the spatial index for efficient O(log n) lookup.

Returns a lazy iterator over LandscapeElement objects.

source

Persefone.inextent – Method.

```
inextent(pos, model)
```

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

source

Persefone.initlandscape – Function.

```
initlandscape(gpkgpath="landscape.gpkg";
              landcover_field=:landcover,
              soil_field=:soil_type)
```

Initialise a Landscape from a GeoPackage at gpkgpath. Each feature is converted to a LandscapeElement, keeping only the Persefone landcover value.

Arguments

- landcover_field: Field name containing landcover values (default: :landcover)
- soil_field: Field name containing soil type values (optional; default: :soil_type)

source

Persefone.isvalid – Method.

```
isvalid(ls::Landscape)
```

Return true if all geometries in the landscape are topologically valid.

source

Persefone.landcover – Method.

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

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

source

Persefone.landelem – Method.

```
landelem(position, model)
```

Return the LandscapeElement at this position (Coord).

source

Persefone.landelem – Method.

```
landelem(id, model)
```

Return the LandscapeElement with this ID.

source

Persefone.nearby_land_elements – Function.

```
nearby_land_elements(landscape, landscapeelement, buffer=5m)
```

Expand the given landscape elements extent by the buffer distance, then return an iterator over all landscape elements in this extent.

source

Persefone.remove_homerange! – Method.

```
remove_homerange!(ls::Landscape, hr::HomeRange)
```

Remove the given homerange from the landscape and update the spatial index.

source

Persefone.safebounds – Method.

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

Clamp a position to the landscape extent.

source

12.5 weather.jl

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

`Persefone.Weather` – Type.

```
Weather
```

Holds the weather information for the whole simulation period.

source

`Persefone.check_missing_weatherdata` – Method.

```
check_missing_weatherdata(dataframe)
```

Check the weather input data for missing values in columns where input values are required.

source

`Persefone.cloudcover` – Method.

```
cloudcover(weather, date)
cloudcover(model, date)
cloudcover(model)
```

Return the average cloudcover in eigths on `date`.

source

`Persefone.daynumber` – Method.

```
daynumber(weather, date)
```

Returns the number of days, counting `weather.firstdate` as day 1.

source

`Persefone.evapotranspiration` – Method.

```
evapotranspiration(weather, date)
evapotranspiration(model, date)
evapotranspiration(model)
```

Return the potential evapotranspiration (ETo) on `date`.

source

`Persefone.findspans` – Method.

```
findspans(predicate_fn, array) -> Vector{UnitRange{Int}}
```

Returns spans of indices in a 1-d `array` where a `predicate_fn` returns `true`. The spans are returned as a `Vector` of `UnitRange{Int}`, where each range is of the form `start_index:end_index`.

source

`Persefone.humidity` – Method.

```
humidity(weather, date)
humidity(model, date)
humidity(model)
```

Return today's average vapour pressure in %.

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/regions/auxiliary/extract_weather_data.R`.

source

`Persefone.maxtemp` – Method.

```
maxtemp(weather, date)
maxtemp(model, date)
maxtemp(model)
```

Return the maximum temperature in °C on `date`.

source

`Persefone.meantemp` – Method.

```
meantemp(weather, date)
meantemp(model, date)
meantemp(model)
```

Return the mean temperature in °C on `date`.

source

`Persefone.mintemp` – Method.

```
mintemp(weather, date)
mintemp(model, date)
mintemp(model)
```

Return the minimum temperature in °C on `date`.

source

`Persefone.precipitation` – Method.

```
precipitation(weather, date)
precipitation(model, date)
precipitation(model)
```

Return the total precipitation in mm on `date`.

source

`Persefone.sunshine` – Method.

```
sunshine(weather, date)
sunshine(model, date)
sunshine(model)
```

Return the sunshine duration in hours on `date`.

source

`Persefone.windspeed` – Method.

```
windspeed(weather, date)
windspeed(model, date)
windspeed(model)
```

Return the average windspeed in m/s on `date`.

source

`Persefone.@cloudcover` – Macro.

```
@cloudcover(date=model.date)
```

Return the average cloudcover in eigths today/on the specified date. Can only be used in scopes where `model` is available.

source

`Persefone.@evapotranspiration` – Macro.

```
@evapotranspiration(date=model.date)
```

Return the potential evapotranspiration (ETo) today/on the specified date. Can only be used in scopes where `model` is available.

source

`Persefone.@humidity` – Macro.

```
@humidity(date=model.date)
```

Return the average vapour pressure (in %) today/on the specified date. Can only be used in scopes where `model` is available.

source

`Persefone.@maxtemp` – Macro.

```
@maxtemp(date=model.date)
```

Return the maximum temperature today/on the specified date. Can only be used in scopes where `model` is available.

source

`Persefone.@meantemp` – Macro.

```
@meantemp(date=model.date)
```

Return the mean temperature today/on the specified date. Can only be used in scopes where `model` is available.

source

`Persefone.@mintemp` – Macro.

```
@mintemp(date=model.date)
```

Return the minimum temperature today/on the specified date. Can only be used in scopes where `model` is available.

source

`Persefone.@precipitation` – Macro.

```
@precipitation(date=model.date)
```

Return the total precipitation in mm today/on the specified date. Can only be used in scopes where `model` is available.

source

`Persefone.@sunshine` – Macro.

```
@sunshine(date=model.date)
```

Return the sunshine duration in hours today/on the specified date. Can only be used in scopes where `model` is available.

source

`Persefone.@windspeed` – Macro.

```
@windspeed(date=model.date)
```

Return the average windspeed in m/s today/on the specified date. Can only be used in scopes where `model` is available.

source

Chapter 13

General utility

These file contain functions for background tasks, such as reading in model configurations, geometric functions, and printing or plotting any output.

13.1 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.resolve_input_dir` – Method.

```
resolve_input_dir(path; configdir=nothing)
```

Return an absolute path to a directory if it exists, trying common roots (`pkgdir`, the directory of the config file, and the current working directory) in that order. Returns `nothing` if the directory cannot be found.

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`, `animal`, `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

13.2 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. This can be any of: "daily", "monthly", "yearly", "end", "never", or a date (e.g. "15 June"). - 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

13.3 types.jl

`Persefone.AbstractCropState` – Type.

```
AbstractCropState
```

The abstract supertype of all crop states in the model. Each crop model has to define a type `CropState <: AbstractCropState`.

source

`Persefone.AbstractCropType` – Type.

```
AbstractCropType
```

The abstract supertype of all crop types in the model. Each crop model has to define a type `CropType <: AbstractCropType`.

source

`Persefone.ModelAgent` – Type.

```
ModelAgent
```

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

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

13.4 geometry.jl

`Persefone.Coord` – Type.

```
Coord
```

Continuous 2D coordinate in projected map units (e.g. meters).

This is aliased to `GeometryBasics.Point{2,Float64}` for compatibility with GeoInterface-aware geometry types.

source

`Persefone.Extent2D` – Type.

```
Extent2D
```

Axis-aligned extent with X/Y bounds, backed by `Extents.Extent`.

source

`Persefone.Extent2D` – Method.

```
Extent2D(geom::Geom)
```

Compute the X/Y extent of a GDAL geometry.

source

`Persefone.Geom` – Type.

```
Geom
```

Geometry handle returned by GDAL/ArchGDAL. For polygons this is typically an `ArchGDAL.IGeometry` of type `wkbPolygon` or `wkbMultiPolygon`.

source

`Persefone.areaof` – Method.

```
areaof(geom)
```

Return the area of a geometry in square meters.

source

`Persefone.bearingfrompoint` – Method.

```
bearingfrompoint(origin, destination)
```

Get the distance and the bearing of the destination point as seen from the origin.

source

`Persefone.circle` – Function.

```
circle(center, radius, coarseness=10)
```

Return a quasi-circular geom with the given properties. Coarseness defines how many degrees lie between two points of the resulting polygon (i.e. if 10°, the "circle" will consist of 36 points).

source

`Persefone.fuzzyposition` – Function.

```
fuzzyposition(pos, maxdist, rng=default_rng())
```

Return a random position that is at most `maxdist` away from `pos`.

source

`Persefone.geom_from_coords` – Method.

```
geom_from_coords(coords)
```

Create a GDAL polygon geometry from a list of coordinates (outer ring only).

source

`Persefone.point_in_polygon` – Method.

```
point_in_polygon(coord, geom)
```

Test whether the given point lies within the given geometry.

source

`Persefone.pointfrombearing` – Method.

```
pointfrombearing(origin, bearing, distance)
```

Calculate the coordinates of the point at the given bearing (in °) and distance from the origin.

source

`Persefone.randompoint` – Function.

```
randompoint(geom, rng)
```

Return a random position that lies within the given geometry.

source

`Persefone.rectangle` – Method.

```
rectangle(center, width, height)
```

Return a rectangular geom with the given properties.

source

`Persefone.square` – Method.

```
square(center, width)
```

Return a square geom with the given properties.

source

`Persefone.@fuzzyposition` – Macro.

```
@fuzzyposition(pos, maxdist)
```

Return a random position that is at most `maxdist` away from `pos`. This is a utility wrapper that can only be used where `model` is available.

source

`Persefone.@randompoint` – Macro.

```
@randompoint(place)
```

Return a random point within a landscape element or a home range. This is a utility wrapper that can only be used where `model` is available.

source

13.5 utils.jl

13.6 makieplots.jl

`Persefone.birdpopulation` – Method.

```
birdpopulation(speciesname, model)
```

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

source

`Persefone.birdstats` – Method.

```
birdstats(speciesname, model)
```

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

source

`Persefone.butterflyabundanceplot` – Method.

```
butterflyabundanceplot(speciesname, model)
```

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

source

`Persefone.butterflylifestats` – Method.

```
butterflylifestats(speciesname, model)
```

Plot various statistics from the marbled white model: fecundity, movement, habitat use

source

`Persefone.butterflytrendsplot` – Method.

```
butterflytrendsplot(speciesname, model)
```

... Returns a Makie figure object.

source

`Persefone.croptrends` – Method.

```
croptrends(model)
```

Plot a dual line graph of cropped area and average plant height and cover 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.visualisemap` – Function.

```
visualisemap(model, date, landscape)
```

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 pre-loaded `Landscape` object. Returns a Makie figure object.

source

Chapter 14

Animal submodel

Unused source files

There are two source files in the `src/animal` directory that are currently not used: `insects.jl` and `energy.jl`. The first defines a function `insectbiomass()` that provides a very rough estimate of insect population density in a given pixel. The second is a (still incomplete) implementation of Dynamic Energy Budgets. Both were begun in the expectation that they would be needed, but then set aside for the time being. We note their existence here should they become useful again (with the caveat that both require testing).

14.1 animal.jl

This file is responsible for managing the animal modules.

`Persefone.Animal` – Type.

```
Animal
```

This is the generic agent type for all animals. Animal behaviour is defined for each taxon using `@create` and `@phase`. Species-specific parameter values are loaded from data files, as specified using the `animal.animaldirectory` and `animal.targetspecies` configuration parameters.

In addition to the taxon-specific fields defined by `@create`, all animals have the following fields:

- `id` An integer unique identifier for this individual.
- `species` A string giving the name of the species.
- `sex` male, female, or hermaphrodite.
- `pos` An (x, y) coordinate tuple of the individual's current position.
- `landelem` A pointer to the landscape element this individual currently occupies.
- `age` The age of the individual in days.
- `phase` The name of the update function to be called during the current life phase.
- `homerange` A geometry delimiting this individual's home range or territory.

source

`Persefone.addanimal!` – Function.

```
addanimal!(speciesname, sex, position, model, phase="")
```

Initialise a new animal individual. Instantiates the object and calls the species' "create" function on it. If phase is empty, it is set to the configured birth phase. Then adds the new animal to the model and returns the animal object.

source

`Persefone.initanimals!` – Method.

```
initanimals!(model)
```

Initialise the model with all simulated animal populations.

source

`Persefone.stepagent!` – Method.

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

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

source

`Persefone.updateanimals!` – Method.

```
updateanimals!(model)
```

Run processes that affect all animals.

source

14.2 taxa.jl

This file contains the constants, macros, and functions needed to define new animal taxa.

`Persefone.PHASES` – Constant.

PHASES is a dict that is used to store all phases defined in the model code (by taxon).

source

`Persefone.@create` – Macro.

```
@create(taxon, body)
```

This is a special case of `@phase` that is used to instantiate animal individuals at birth. Use this to define taxon-specific variables and to initialise their values.

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

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.@phase` – Macro.

```
@phase(taxon, 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. Species sharing a similar life cycle are grouped together in a taxon. Each phase definition is used for all species belonging to this taxon, with species-specific behaviour encoded using user-configured parameters.

@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 file, as well as all standard `Animal` variables (e.g. `self.age`, `self.sex`, `self.offspring`).
- `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`.

After the phase definition, use `phasedoc` to document which parameters this phase uses (this is needed to check the validity of species parameter files).

source

14.3 macros.jl

This file contains many of the macros that can be used by the taxa (i.e. within @phase).

`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.@checkhabitat` – Macro.

```
@checkhabitat(pos=self.landelem, habitatdescriptor)
```

Check whether the given habitatdescriptor fits the given position. This is a utility wrapper that can only be used nested within `@phase` (if used with no arguments), or wherever `model` is available.

source

`Persefone.@cropcover` – Macro.

```
@cropcover(pos=self.landelem)
```

Returns the local cropcover. This is a utility wrapper that can only be used nested within `@phase` (if used with no arguments), or wherever `model` is available.

source

`Persefone.@cropgroup` – Macro.

```
@cropgroup(pos=self.landelem)
```

Returns the local cropgroup. This is a utility wrapper that can only be used nested within `@phase` (if used with no arguments), or wherever `model` is available.

source

`Persefone.@cropheight` – Macro.

```
@cropheight(pos=self.landelem)
```

Returns the local cropheight. This is a utility wrapper that can only be used nested within `@phase` (if used with no arguments), or wherever `model` is available.

source

`Persefone.@cropname` – Macro.

```
@cropname(pos=self.landelem)
```

Returns the local cropname. This is a utility wrapper that can only be used nested within `@phase` (if used with no arguments), or wherever `model` is available.

source

`Persefone.@destroynest` – Macro.

```
@destroynest(reason)
```

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

source

`Persefone.@directionto` – Macro.

```
@directionto(pos=self.pos)
```

Calculate the direction to a position, 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` (if used with no arguments), or wherever `model` is available.

source

`Persefone.@distanceto` – Macro.

```
@distanceto(pos=self.pos)
```

Calculate the distance to a position, 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` (if used with no arguments), or wherever `model` is available.

source

`Persefone.@distancetoedge` – Macro.

```
@distancetoedge(pos=self.pos)
```

Calculate the distance to the closest neighbouring habitat. This is a utility wrapper that can only be used nested within `@phase` (if used with no arguments), or wherever `model` is available.

source

`Persefone.@findnearest` – Macro.

```
@findnearest(pos=self.pos)
```

Calculate the direction to a position, 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` (if used with no arguments), or wherever `model` is available.

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.@inextent` – Macro.

```
@inextent(pos)
```

Check whether the given position is within the extent of the landscape.

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(geom)
```

Test whether this geometry 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(pos=self.landelem)
```

Returns the local landcover. This is a utility wrapper that can only be used nested within `@phase` (if used with no arguments), or wherever `model` is available.

source

`Persefone.@landelem` – Macro.

```
@landelem(pos=self.pos)
```

Return the landscape element at this position, or the element with `id == pos`. This is a utility wrapper that can only be used nested within `@phase` (if used with no arguments), or wherever `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.@neighbours` – Macro.

```
@neighbours([radius], conspecifics=true)
```

Return an iterator over all (by default conspecific) animals in the animal's landscape element or within a given radius around its position, excluding itself. This can only be used nested within `@phase`.

source

`Persefone.@occupy` – Macro.

```
@occupy(geom)
```

Set the given geometry as the animal's home range. Use `@vacate` to remove the home range again. This 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.@vacate` – Macro.

```
@vacate()
```

Remove this animal's home range. This can only be used nested within `@phase`.

source

`Persefone.@walk` – Macro.

```
@walk(direction, distance)
```

Walk the animal in a given direction, which is either specified by a bearing (0-359°), or by a direction string (e.g. "northwest", or "random"). This is a utility wrapper that can only be used nested within `@phase`.

source

14.4 individuals.jl

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

`Persefone.followanimal!` – Method.

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

Move the following animal to a location at most `distance` away from the leading animal.

source

`Persefone.isoccupied` – Method.

```
isoccupied(geometry, species, model)
```

Check whether the given geometry could be inserted as a home range without overlapping any other home ranges of the given species.

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!` – Function.

```
occupy!(animal, model, geom, exclusive=true)
```

Create a new HomeRange for the animal with the given geometry. If `exclusive` is `true`, this function fails if the given Geom overlaps with an existing HomeRange belonging to a conspecific. Returns `true` if successful, or `false` in case of conflicts.

source

`Persefone.reproduce!` – Function.

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

Remove the animal's home range.

source

`Persefone.walk!` – Method.

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

The animal moves to the point defined by the bearing and distance from its current location.

source

`Persefone.walk!` – Method.

```
walk!(animal, model, direction, distance)
```

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

source

14.5 populations.jl

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

`Persefone.animal` – Method.

animal(id, model)

Return the animal with the given ID, or nothing if it's no longer alive.

source

`Persefone.countanimals` – Function.

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

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

source

`Persefone.countanimals` – Function.

```
countanimals(model, landelem, species="")
```

Return the number of animals in this landscape element, optionally filtering by species.

source

`Persefone.homerangesize` – Method.

```
homerangesize(animal)
```

Calculate the size of this animal's home range in m² (utility wrapper).

source

`Persefone.initindividuals!` – Method.

```
initindividuals!(species, landelem, model)
```

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

source

`Persefone.initpopulation!` – Method.

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

Initialise the population of the given species, based on the configured initialisation parameters.

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 home range of an animal of the given species.

source

`Persefone.nearby_animals` – Function.

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

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

source

`Persefone.nearby_animals` – Function.

```
nearby_animals(model, landelem, species="")
```

Return a list of animals in this landscape element, optionally filtering by species.

source

`Persefone.neighbours` – Function.

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

Return a list of animals in the same landscape element as this animal, excluding itself. By default, only return conspecific animals.

source

`Persefone.neighbours` – Function.

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

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

source

`Persefone.populationdensity` – Function.

```
populationdensity(landelem, model, species="")
```

Calculate the population density for a given species ("" -> all animals) in a given patch. Returns individuals/hectare.

source

`Persefone.populationdensity` – Function.

```
populationdensity(model, landelem, species="")
```

Return the population density in individuals/ha of this landscape element, optionally filtering by species.

source

14.6 ecologicaldata.jl

This file contains the code to extract and summarise data from the animal submodel for later analysis.

`Persefone.birdabundance` – Method.

```
birdabundance(speciesname, model)
```

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

source

`Persefone.butterflyabundance` – Method.

```
butterflyabundance(speciesname, model)
```

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

source

`Persefone.butterflytrends` – Method.

```
butterflytrends(speciesname, model)
```

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

source

`Persefone.initecologicaldata` – Method.

```
initecologicaldata()
```

Create output files for each data group collected by the animal 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 `animal.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 `animal.popoutfreq`.

source

14.7 speciesparams.jl

This file handles the metadata associated with all possible species parameters.

`Persefone.SPECIESPARAMS` – Constant.

SPECIESPARAMS is a dict that stores all species parameters used by phases in the code.

source

`Persefone.SpeciesParameter` – Type.

```
SpeciesParameter
```

A struct to define a species parameter. This contains not just the parameter name and a short docstring, but also information about the expected type and a function to convert the type as returned by the TOML input file into the required type. In addition, it can contain a dict that returns further species parameters that are required depending on the values chosen for this parameter, as well as a list of value options for this parameter.

source

`Persefone.addspeciesparam!` – Function.

```
addspeciesparam!(name, docstring, type=String, convert=(x)->x; options=[])
```

Add a new SpeciesParameter to the internal reference system.

source

`Persefone.checkspeciesdict` – Method.

```
checkspeciesdict(params)
```

Verify that the species dict contains all needed parameters. Returns a vector of missing parameters, or an empty vector if the check is successful. (Note: call this after `convertparametertypes!()`, as it needs the correct parameter types.)

source

`Persefone.convertparametertypes!` – Method.

```
convertparametertypes!(speciedict)
```

Convert all parameter values to their specified types, using the metadata stored in `SpeciesParameter`. Gives a warning when conversion fails. Returns the converted dict (with keys changed from strings to symbols for performance).

source

`Persefone.initspeciesdicts` – Method.

```
initspeciesdicts(animaldir, targetspecies)
```

Load the species definition files into a dict.

source

`Persefone.requiredparams` – Function.

```
requiredparams(baseparam, value="")
```

Return the list of parameters that must defined if the baseparam has the given value.

source

`Persefone.requireparams!` – Method.

```
requireparams!(baseparam, value, requiredparams, condition=(x)->true)
```

Stipulate species parameters that must be defined if the baseparam takes on the given value. Pass an empty string as the value to denote parameters that are always required.

source

14.8 habitatdescriptors.jl

This file implements a simple string format for describing habitats.

`Persefone.HabitatExpression` – Type.

```
HabitatExpression
```

The internal type representing habitat descriptors (as created by `parsehabitat()` and `@h_str`). Creates a tree structure, consisting of one or two branches joined by a boolean link. The nodes of the tree are `HabitatProperty`s or `nothing`.

source

`Persefone.HabitatProperty` – Type.

```
HabitatProperty
```

A helper struct for HabitatExpression, storing information about which aspect of the environment is to be tested, what the reference value is, and which comparison to apply.

source

`Persefone.checkhabitat` – Method.

```
checkhabitat(position, habitatexpression, model)
```

Evaluate the given habitat expression with regards to a given position in the model landscape. Returns true or false.

source

`Persefone.checkhabitat` – Method.

```
checkhabitat(landelem, habitatexpression)
```

Evaluate the given habitat expression on a landscape element, recursively evaluating all component expressions and linking them with the relevant boolean operators. Returns true or false.

source

`Persefone.checkhabitat` – Method.

```
checkhabitat(landelem, habitatproperty)
```

Evaluate the given habitat property, returning true if it applies to the given position in the model landscape and false if not.

source

`Persefone.directionto` – Function.

```
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` – Function.

```
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.parsehabitat` – Method.

```
parsehabitat(descriptor)
```

Parse a habitat string and return the resulting HabitatExpression. HabitatExpressions can consist of multiple subexpressions, joined by either AND or OR. At the lowest level, each subexpression is parsed using `parsehabitatproperty()`.

Examples: "Fallow", "Orchard OR FruitPlantation", "Grassland OR (GRAIN AND NATURAL)", "ExtensiveGrassland AND (cropheight<=50cm AND cropheight>20cm)", "GRAIN AND (!SilageMaize AND !CornMaize)"

source

`Persefone.parsehabitatproperty` – Method.

```
parsehabitatproperty(descriptor)
```

Parse a HabitatProperty from a string (helper function for parsehabitat()). HabitatProperties may consist of either a single word (which must be a valid instance of a LandCover, LandCoverCategory, CropName, or CropGroup), or a word specifying an aspect and the associated value, separated by one of ==, !=, >=, <=, >, < (and no spaces!). The aspect may be one of "landcover", "landcovercategory", "cropname", "cropgroup", "cropheight", "cropcover". If the property is a single word, a preceding exclamation point may be used to negate it.

Examples: "Arable", "!WinterWheat", "cropgroup==LEGUME", "cropheight<50"

source

`Persefone.@h_str` – Macro.

```
@h_str(s)
```

A string macro for constructing habitat descriptors. Allows syntax like the following: `h"Arable OR Grassland"`, `h"WinterWheat AND height>5cm"` Calls `parsehabitat()`.

source

Chapter 15

Species models

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

15.1 Birds

`Persefone.#1358#fun` – Function.

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

source

`Persefone.#1382#fun` – Function.

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

source

`Persefone.#1383#fun` – Function.

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

source

`Persefone.#1430#fun` – Function.

Once a male has found a territory, he remains in it until the breeding season is over. (It should adjust it to new conditions when and as necessary, but this is not yet implemented.)

source

`Persefone.#1431#fun` – Function.

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

source

`Persefone.#1501#fun` – Function.

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

source

`Persefone.#1525#fun` – Function.

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.allowsnestingskylark` – Method.

```
allowsnestingskylark(animal, model, pos)
```

Check whether the given position is suitable for nesting for skylarks.

source

`Persefone.birdrandommove!` – Function.

```
birdrandommove!(self, model, ttl=10)
```

The bird checks random spots within its movement range and moves to the first one that satisfies its forage habitat requirements. If `ttl` attempts are all unsuccessful, it stays where it is. Returns `true` if the move is successful, `false` if not.

source

`Persefone.circularterritory` – Function.

```
circularterritory(animal, model, increment=5m, gridspacing=2m)
```

Attempt to build a territory surrounding the bird's current location. The territory is constructed as a circle covering the species' minimum territory size, and expanded incrementally until the required effective size (i.e. area weighted by habitat quality) is reached. If this cannot be done without infringing on another territory, return `nothing`, otherwise return the resulting geometry.

source

`Persefone.destroynest!` – Method.

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

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

source

`Persefone.dynamicterritory` – Function.

```
dynamicterritory(animal, model, blocksize=10m)
```

Attempt to build a territory surrounding the bird's current location by successively adding square blocks concentrically around it, as long as they don't infringe on another territory. Keeps adding territory blocks until the required effective territory size (area weighted by habitat quality) is reached, then returns the geometry. Returns `nothing` if there is insufficient space.

source

`Persefone.effectivesize` – Function.

```
effectivesize(self, model, territory, gridspacing=2m)
```

Calculate the functional area of the territory, weighting the absolute size by the local forage quality. Works by superimposing a grid of points over the territory and summing up the quality rating of each point.

source

`Persefone.foragequalityskylark` – Method.

```
foragequalityskylark(animal, model, pos)
```

Calculate the relative quality of the habitat at this position for foraging for a skylark. 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

15.2 Butterflies

`Persefone.#1693#fun` – Function.

Initialise a butterfly individual. Mainly defines the time this individual will spend in each phase. (This ought to be temperature-dependent rather than random, but I don't have data for that.)

source

`Persefone.#1786#fun` – Function.

Juvenile individuals (i.e. eggs, larvae, pupae) simply wait for the eclosing day.

source

`Persefone.#1787#fun` – Function.

Juvenile individuals (i.e. eggs, larvae, pupae) simply wait for the eclosing day.

source

`Persefone.#1788#fun` – Function.

Juvenile individuals (i.e. eggs, larvae, pupae) simply wait for the eclosing day.

source

`Persefone.#1789#fun` – Function.

Adult butterflies (we only simulate females) fly around more or less randomly on days with good weather, laying eggs on suitable habitat.

Movement

Adults move a given number of steps each day, depending on the temperature (see below). Each step, an individual moves to a random location within its perceptual range that matches its habitat requirements (i.e. either arable or extensive grassland with plant heights within the required range of 30-60cm). Here, the individual lays an egg if it has not yet laid all its eggs for that day. Locations that are not suitable habitat may still be chosen to move to with a certain probability (given by the `habitatpreference` parameter), but without subsequent oviposition.

Temperature

Temperature affects both the distance moved and the number of eggs laid each day. The optimal temperature is taken to be the midway point between the species' minimum and maximum temperatures (i.e. 24°C). Outside the species' temperature range (18-30°C), neither movement nor oviposition take place. Within that range, the number of steps each day peaks at the optimum temperature and declines linearly on either side of it. (cf. Evans et al., 2019). The number of eggs laid declines linearly if the temperature is below the optimum, but stays stable above it (cf. Gotthard et al., 2007). The daily mean temperatures are

used as the basis for calculation (using the maximum temperature produces wrong model results during heat waves).

Note: this is quite specific to Melanargia galathea and likely needs to be adjusted in future.

source

`Persefone.butterflyrandommove!` – Function.

```
butterflyrandommove!(butterfly, model)
```

The butterfly randomly inspects points within its field of view and moves to the first suitable spot it finds. Depending on the `habitatpreference` parameter, spots that are not suitable habitat may also be selected. Spots with higher population densities are more likely to be avoided (avoidance increases linearly up to 100% at `maxdensity`). Returns `true` if the move is successful. Otherwise, the butterfly moves to any spot in its perception range and returns `false`.

source

`Persefone.habitatcategory` – Method.

```
habitatcategory(butterfly, model)
```

Return the habitat category of the butterfly's current location (using a taxon-specific classification).

source

`Persefone.recordlifestats` – Method.

```
recordlifestats(butterfly, model)
```

Save this butterfly's life stats to file.

source

Chapter 16

Crop submodel

Persefone reimplements two different crop models for different purposes. AquaCrop is a well-established crop growth model developed by the FAO, which provides quite reliable estimates of plant growth but is data-intensive to parameterise. The vegetation submodel of ALMaSS is much simpler, but also less reliable. Accordingly, we use AquaCrop for our main crop types, and ALMaSS for the rest (especially grass growth).

16.1 cropnames.jl

This provides a unified name list of crop types that can potentially be simulated by Persefone.jl.

`Persefone.cropgroup` – Method.

```
cropgroup(cropname)
```

Return the CropGroup that this crop belongs to.

source

16.2 farmplot.jl

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

`Persefone.harvest!` – Method.

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

Harvest the crop of this farm plot (i.e. a landscape polygon with a cropstate).

source

`Persefone.sow!` – Method.

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

Sow the specified crop on a farm plot (i.e. a landscape polygon with a cropstate).

source

`Persefone.stepagent!` – Method.

```
stepagent!(le::LandscapeElement, model)
```

Update the vegetation in a landscape polygon using the crop model (if applicable).

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

16.3 cropmodels.jl

This initialises the crop models and the farmplots at the beginning of the simulation.

`Persefone.initcropmodels` – Method.

```
initcropmodels(cropmodels, cropdirectory; region)
```

Initialise the crop models given as a comma-delimited string (e.g. "almass,aquacrop"). The crop model parameters are read from the `cropdirectories`, also a comma-delimited string. Returns the crop types available in the simulation.

source

`Persefone.initfields!` – Method.

```
initfields!(model)
```

Identify all arable and grassland landscape polygons and save their IDs for quick reference.

source

16.4 almass.jl

This file reimplements the ALMaSS vegetation submodel.

`Persefone.ALMaSS.temperature_to_solar_conversion_c3` – Constant.

Temperature to solar conversion factor for C3 plants.

source

`Persefone.ALMaSS.temperature_to_solar_conversion_c4` – Constant.

Temperature to solar conversion factor for C4 plants.

source

`Persefone.ALMaSS.CropCurveParams` – Type.

```
CropCurveParams
```

The values in this struct define one crop growth curve.

source

Persefone.ALMaSS.CropState – Type.

```
CropState
```

The state data for an ALMaSS vegetation point calculation.

source

Persefone.ALMaSS.CropType – Type.

```
CropType
```

The type struct for all crops. Currently follows the crop growth model as implemented in ALMaSS.

source

Persefone.ALMaSS.GrowthPhase – Type.

```
GrowthPhase
```

ALMaSS crop growth curves are split into five phases, triggered by seasonal dates or agricultural events.

source

Base.tryparse – Method.

```
Base.tryparse(type, str)
```

Extend `tryparse` to allow parsing GrowthPhase values. (Needed to read in the CSV parameter file.)

source

Persefone.ALMaSS.buildgrowthcurve – Method.

```
buildgrowthcurve(data)
```

Convert a list of rows from the crop growth data into a CropCurveParams object.

source

Persefone.ALMaSS.readcropparameters – Method.

```
readcropparameters(cropdirectory)
```

Parse a CSV file containing the required parameter values for each crop (as produced from the original ALMaSS file by `convert_almass_data.py`).

source

Persefone.ALMaSS.setphase! – Method.

```
setphase!(cropstate, phase)
```

Set the growth phase of an ALMaSS `cropstate`.

source

`Persefone.ALMaSS.solar_conversion_c3` – Method.

```
solar_conversion_c3(temperature)
```

Solar conversion factor (no units) for C3 plants.

source

`Persefone.ALMaSS.solar_conversion_c4` – Method.

```
solar_conversion_c3(temperature)
```

Solar conversion factor (no units) for C4 plants.

source

`Persefone.harvest!` – Method.

```
harvest!(cropstate, model)
```

Harvest the crop of this cropstate.

source

`Persefone.sow!` – Method.

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

Change the cropstate to sow the specified crop.

source

`Persefone.stepagent!` – Method.

```
stepagent!(cropstate, model)
```

Update a farm plot by one day.

source

16.5 aquacrop.jl

This file integrates the AquaCrop model (implemented in a separate pacakge) into Persefone.

`Persefone.AquaCropWrapper.readcropparameters` – Method.

```
readcropparameters(cropdirectory; region)
```

Read parameters needed for the AquaCrop crop module in Persefone:

- a CSV file crop_data.csv containing some parameters required to map

AquaCrop crop names to Persefone crop names, as well as additional crop data needed for Persefone (cropgroup, minsowdate, maxsowdate)

- modified crop parameters for each region, e.g. the file `data/crops/aquacrop/regions/jena/winter_wheat.toml` can be used to overload the parameters of the Persefone crop "winter wheat" for the "jena" region (falls back to Germany-wide parameters, then defaults)

source

Persefone.AquaCropWrapper.weightedmean – Method.

```
weightedmean(weights, params)
```

Weighted mean for AquaCrop parameter containers (AbstractParametersContainer), combining numeric fields and recursing into nested containers.

source

Persefone.harvest! – Method.

```
harvest!(cropstate, model)
```

Harvest the crop of this cropstate.

source

Persefone.sow! – Method.

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

Change the cropstate to sow the specified crop.

source

Persefone.stepagent! – Method.

```
stepagent!(cropstate, model)
```

Update a crop state by one day.

source

Chapter 17

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.

17.1 farmevents.jl

This file provides the interface for handling management events (e.g. tillage, harvest).

`Persefone.FarmEvent` – Type.

```
FarmEvent
```

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

source

`Persefone.Management` – Type.

The types of management event that can be simulated

source

`Persefone.updateevents!` – Method.

```
updateevents!(model)
```

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

source

17.2 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.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

`Persefone.updatefarmsandfields!` – Method.

```
updatefarmsandfields!(model)
```

Call the step function on all farms and fields, and run any scenarios.

source

17.3 farmdata.jl

This file collects relevant output data from the farm model.

`Persefone.initfarmdata` – Method.

```
initfarmdata()
```

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

source

`Persefone.savefielddata` – Method.

```
savefielddata(model)
```

Return a data table (to be printed to `fields.csv`), giving the current date, and the area and average height and cover of each crop in the landscape. May be called never, daily, monthly, yearly, or at the end of a simulation, depending on the parameter `farm.fieldoutfreq`.

source

17.4 scenarios.jl

This file contains management scenarios that can modify the functioning of the farm component.

`Persefone.applyscenarios` – Method.

```
applyscenarios(model)
```

Calls functions that can change settings or otherwise manipulate the model to implement different simulation scenarios.

source

`Persefone.sc_thuringian_fallows` – Method.

```
sc_thuringian_fallows(model)
```

This scenario changes the amount of set-aside/fallow area over time, based on historical developments in Thuringia following changes in the CAP (2007: abolishment of set-asides (used as a market instrument); 2015: introduction of Greening). Note: does not include the most recent changes (CAP post-2022).

Based on data from Thüringer Landesamt für Statistik, via Jungmann (2018): https://wirtschaft.thueringen.de/fileadmin/Landwirtschaft/Agrarpolitik/oevf*biodiversitat*thueringen_nov2018.pdf

source

Appendix B: Crop model descriptions, parameter calibration, and cross-validation

1 Crop model descriptions

1.1 AquaCrop (FAO Crop Water Productivity Model)

AquaCrop (Steduto et al., 2009; Raes et al., 2009) is a process-based crop growth model designed to predict crop yield response to water conditions. It simulates plant growth over time — including canopy cover, phenology, biomass, yield, and other variables — based on physical and physiological processes. The model also incorporates management practices and environmental input data. Inside Persefone, the AquaCrop.jl (Díaz Iturry et al., 2025) implementation of the AquaCrop model was used, which is a direct translation of the original AquaCrop Fortran code to the Julia programming language.

The AquaCrop model focuses on the yield response to water availability. It is designed to use a relatively low number of parameters, which are expected to be easy to estimate. The model emphasises the fundamental processes involved in crop productivity and the responses to water deficits, both from physiological and agronomic perspectives. In addition to crop parameters, AquaCrop also relies on climate input data and soil type characterisation. Temperature data are used to track crop development through the calculation of growing degree days (GDD). Rainfall and soil properties are used to estimate the soil water content within the root zone. Based on these calculations, the model estimates the crop's canopy cover (CC). Subsequently, using reference evapotranspiration (ETo) and the water productivity (WP) parameter, it estimates biomass production. Finally, the harvest index (HI) is applied to convert biomass into yield, as described in (Steduto et al., 2009; Raes et al., 2009). Daily solar radiation is not an explicit AquaCrop model input, but its effect enters indirectly via the reference evapotranspiration. We extended the model by adding functionality to calculate plant height from biomass.

Model inputs. To simulate with AquaCrop and predict plant yields for given conditions, the following data are needed:

- climate data: min./max. daily air temperature, rainfall, reference evapotranspiration
- soil type: five horizons, each with hydraulic conductivity, water content at saturation, field capacity, and permanent wilting point. These parameters can be set from preset soil types such as "silty loam" etc.
- crop type and parameters
- sowing date and sowing density

Model outputs. The outputs relevant to Persefone are:

- canopy cover
- dry biomass and yield

Model calibration. To calibrate crop parameters to empirical crop data, the following is needed:

- input data as above
- biomass or yield per crop type
- phenological phase dates per crop type (date of germination, flowering, etc)

1.2 ALMaSS vegetation model

ALMaSS (Animal, Landscape and Man Simulation System) is an agent-based landscape simulation framework developed to study fauna and management in agricultural environments (Topping, Hansen, et al., 2003; Topping and Duan, 2024). It includes a vegetation/crop growth sub-model that provides daily vegetation state (e.g., height, biomass, fractional cover) to the animal agents. The crop module is a simple, semi-mechanistic light-use-efficiency (LUE) model driven primarily by solar radiation and temperature (growing degree-days), with stage changes driven by crop management (sowing, harvest) or calendar time (1 January, 1 March). The model does not consider water availability or transpiration, and assumes that an adequate amount of water is available. In Persefone, the ALMaSS vegetation model has been re-implemented in the Julia programming language, following the original publication and source code.

Each crop stage has its own set of three piecewise-linear growth curves that determine the daily change in plant height, green and total leaf area index of the plant in terms of growing degree-days.

The amount of radiation absorbed by the plant canopy can be calculated from the green leaf area index with the Beer-Lambert law of extinction. Finally, the change in dry matter is calculated from the amount of solar energy with a simple multiplicative model of crop- and temperature-dependent factors.

The total accumulated dry matter over the course of a simulation can be calculated as the sum of daily changes, which are determined by the leaf area index, temperature, incoming radiation:

$$W = \sum_{d=1}^{n} \varepsilon \, f(T(d)) \, \phi(L(d)) \, R(d) \, p(d)$$

- W: accumulated dry matter (g/m^2) at day n
- ε: radiation use efficiency (g/MJ), depends on plant species
- $f(T)$: effect of temperature T (°C) on radiation use efficiency, depends on plant species
- ϕ: fraction of incoming light intercepted by canopy; estimated as $\phi(L(d)) = 1 - e^{-kL(d)}$ from leaf area index L, with extinction coefficient $k = 0.4$
- R: incoming daily radiation (MJ/m^2)
- p: effect of fertiliser use

Model inputs. A simulation with the ALMaSS vegetation model needs the following input data:

- climate data: min./max. daily air temperature, incoming daily radiation
- crop parameters: growth curves for height and green/total leaf area index in terms of growing-degree days (GDD), radiation use efficiency for crop type,
- sowing date

Model outputs.

- canopy height, green leaf area index, total leaf area index
- accumulated dry matter (biomass)

Table 1: List of crop types currently available in Persefone.jl, and which crop model is used to simulate them (AquaCrop.jl or ALMaSS).

Crop name	**Model used**
Winter wheat	AquaCrop
Spring wheat	AquaCrop
Winter barley	AquaCrop
Spring barley	AquaCrop
Triticale	ALMaSS
Spring oat	ALMaSS
Winter rye	ALMaSS
Spring rape	ALMaSS
Winter rape	AquaCrop
Silage maize	AquaCrop
Potato	ALMaSS
Carrot	ALMaSS
Sunflower	ALMaSS
Lucerne	ALMaSS
Fodder beet	ALMaSS
Peas	ALMaSS
Grass clover mix	ALMaSS
Grass ley	ALMaSS
Intensive grassland	ALMaSS
Extensive grassland	ALMaSS
Natural grass	ALMaSS
Fallow	ALMaSS
No crop	ALMaSS

2 AquaCrop height-biomass regression

AquaCrop does not predict canopy height, which is required by our ecosystem simulator Persefone. We therefore infer height from dry biomass by fitting a saturating rational function to observations of plant height and biomass for major field crops.

Data and variables. We used the multi-site crop monitoring dataset of (Reichenau et al., 2020), pooling observations across four monitored sites in western Germany (Hürtgenwald, Merken, Selhausen, Merzenhausen) for the following crops: maize (`MA`), winter wheat (`WW`), winter barley (`WB`), and rapeseed (`RA`). The response variable is the measured canopy height (cm). The predictor is dry biomass per plant ($\mathrm{g\,plant^{-1}}$), computed as the sum of dry mass compartments divided by the observed plant density:

$$x \;=\; \frac{\mathrm{DW}_{\text{green leaves}} + \mathrm{DW}_{\text{brown leaves}} + \mathrm{DW}_{\text{green stems}} + \mathrm{DW}_{\text{brown stems}} + \mathrm{DW}_{\text{fruit}}}{n_{\text{plants / m}^2}}, \tag{1}$$

where the five dry-weight compartments correspond to the dataset columns `DW_green_leaves`, `DW_brown_leaves`, `DW_green_stems`, `DW_brown_stems`, `DW_fruit`, and $n_{\text{plants / m}^2}$ is the observed plant density (column `num_plants_m2`). All crop-specific fits pool data over sites and dates (no site or date effects are modelled).

Model. For each crop, canopy height h as a function of dry biomass per plant x is modelled by a three-parameter rational function:

$$h(x) \;=\; \frac{a\,x^{\,b}}{c + x^{\,b}}, \qquad a, b, c > 0. \tag{2}$$

This form is monotone and saturating with interpretable parameters: the asymptotic maximum height is $\lim_{x\to\infty} h(x) = a$; the half-saturation biomass is

$$x_{50} \;=\; c^{\,1/b} \quad \text{such that} \quad h(x_{50}) = \tfrac{1}{2}a, \tag{3}$$

and the exponent b controls how sharply height increases with biomass around x_{50}. For small x, $h(x) \approx (a/c)\,x^{\,b}$.

The saturating form of the fitting function ensures realistic capping of height and realistic behavior when biomass is large.

Estimation. Parameters (a, b, c) were estimated by nonlinear least squares for each crop separately, minimizing the sum of squared residuals in height. We used the Levenberg-Marquardt algorithm as implemented in `LsqFit.jl`, with starting values $(a, b, c) = (100,\ 2,\ 4)$ chosen to reflect typical cereal/maize canopy heights and a sigmoidal rise with biomass. Goodness

of fit is summarized by the coefficient of determination

$$R^2 \;=\; 1 - \frac{\sum_i \big(h_i - \hat{h}(x_i)\big)^2}{\sum_i \big(h_i - \bar{h}\big)^2}, \tag{4}$$

where h_i are observed heights, $\hat{h}(x_i)$ are model predictions from (2), and $\bar{h}$ is the sample mean height. Fits were performed independently for maize, winter wheat, winter barley, and rapeseed using all available observations for which (1) was defined.

Resulting functional fits.

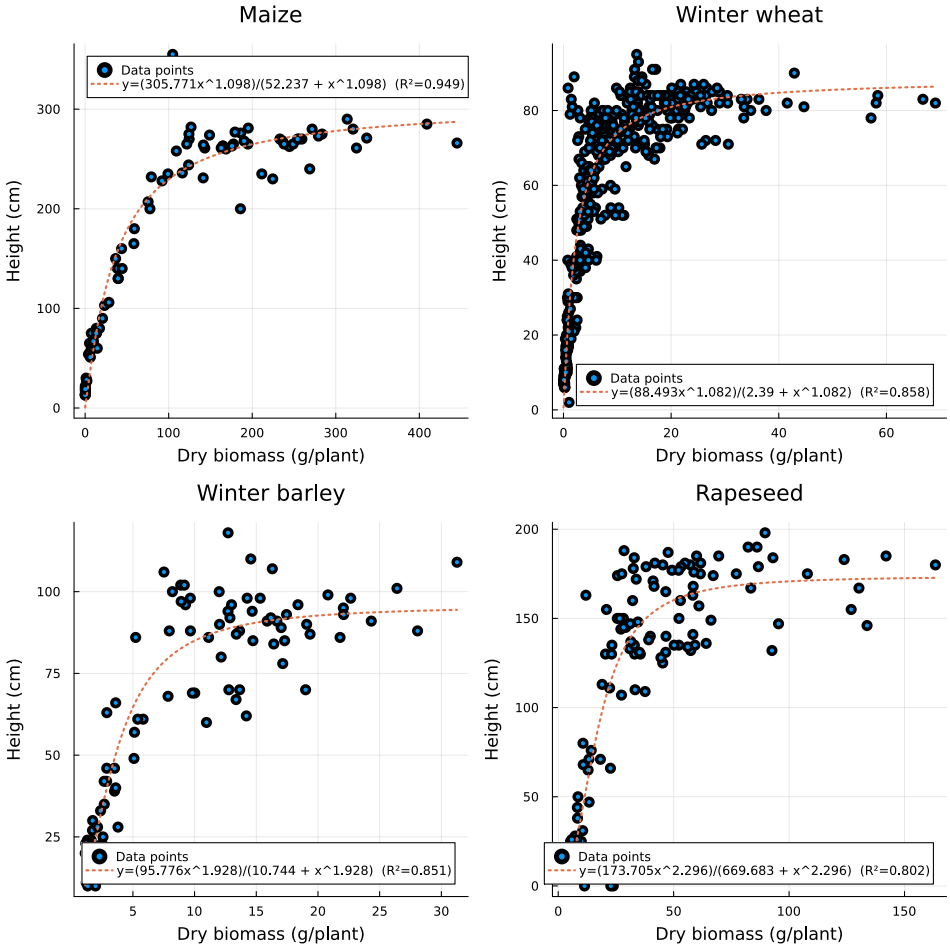

Figure 1: Data and rational function fits for predicting plant height from dry biomass

Use in Persefone. In the simulator, crop height is obtained by evaluating (2) with the fitted, crop-specific parameters from this dataset $(\hat{a}, \hat{b}, \hat{c})$ and the current AquaCrop dry biomass per plant x in the simulation.

Code availability. The notebook with the code for the height-biomass nonlinear regression can be found in the Persefone distribution in the path `docs/notebook-aquacrop-height-biomass-regression.jl`.

3 AquaCrop parameter calibration

This section summarises the procedure used to calibrate crop parameters for the `AquaCrop.jl` model—a Julia re-implementation of the FAO AquaCrop model—as integrated in the Persefone ecosystem simulator. The calibration was performed for three regions in Thuringia (Eichsfeld, Jena, and Thüringer Becken) and four crops (silage maize, winter wheat, winter barley, and winter rapeseed).

Calibration was carried out independently for each (region, crop) pair to obtain region-specific parameter sets that reflect local climate and soils.

All calibrations rely on publicly available climate and statistical yield data (Deutscher Wetterdienst via `rdwd` (Barry, 2025); Statistical Office of Thuringia (Statistical Office of Thuringia, 2025)).

3.1 Data used

The following inputs are required by the model and were assembled for calibration:

- **Climate:** Daily temperature and precipitation (and other meteorological variables as needed for reference evapotranspiration), sourced from the German Weather Service (DWD) via the `rdwd` interface.
- **Soil:** Regional soil type/texture information to parameterise water holding characteristics of the root zone.
- **Management:** Crop identity, sowing date, and (where available) sowing density and reported phenological phase dates.
- **Targets for calibration/validation:** Annual yield statistics by crop and region from the Statistical Office of Thuringia; observed phenological phase dates (when available).

3.2 Calibration workflow

Calibration followed a stepwise sequence that follows the structure of the AquaCrop model. Parameters were tuned in three stages, moving from development timing to canopy/roots to biomass and yield:

(1) Phenology and development timing. Observed phase dates were used, together with climate data, to refine GDD requirements for the emergence–flowering–maturity stages of each crop. The aim was to align simulated stage timing with observations, thereby stabilising subsequent steps that depend on accurate development calendars.

(2) Canopy and root-zone dynamics. Parameters controlling canopy expansion, canopy senescence/decay, and root depth expansion were adjusted to match the observed seasonal evolution of the canopy under local water supply. Conceptually, the root profile determines the effective soil water reservoir, which modulates water stress; this stress feeds back on canopy growth and transpiration, so a consistent canopy/root parameterisation is required before biomass can be reliably fit.

(3) Biomass formation and yield conversion. With timing and canopy dynamics stabilised, water productivity (WP), temperature-stress responses, potential transpiration scaling, and harvest index (HI) parameters were tuned to reproduce observed fresh yields, with the exception of silage maize, where the target variable was fresh biomass instead of fresh yield. The emphasis is on matching interannual variability and mean levels by adjusting the parameters that directly govern biomass accumulation and the partitioning of biomass to harvested yield.

3.3 Parameter groups adjusted

Table 2 lists the principal parameter groups targeted during calibration. Exact parameter symbols follow `AquaCrop.jl` conventions.

Table 2: Principal parameter groups adjusted during calibration.

Group	**Purpose / examples**
Development (GDD)	Align emergence, flowering, and maturity timing with observed phase dates.
Canopy dynamics	Control CC expansion and decay (e.g., maximum CC, growth rate, senescence rate).
Root expansion	Set root depth trajectory to determine the active soil water reservoir.
Water productivity (WP)	Scale biomass production per unit transpired water.
Stress response scalars	Represent temperature/water stress effects on growth and transpiration.
Harvest index (HI)	Convert biomass to yield; adjust HI level and dynamics.

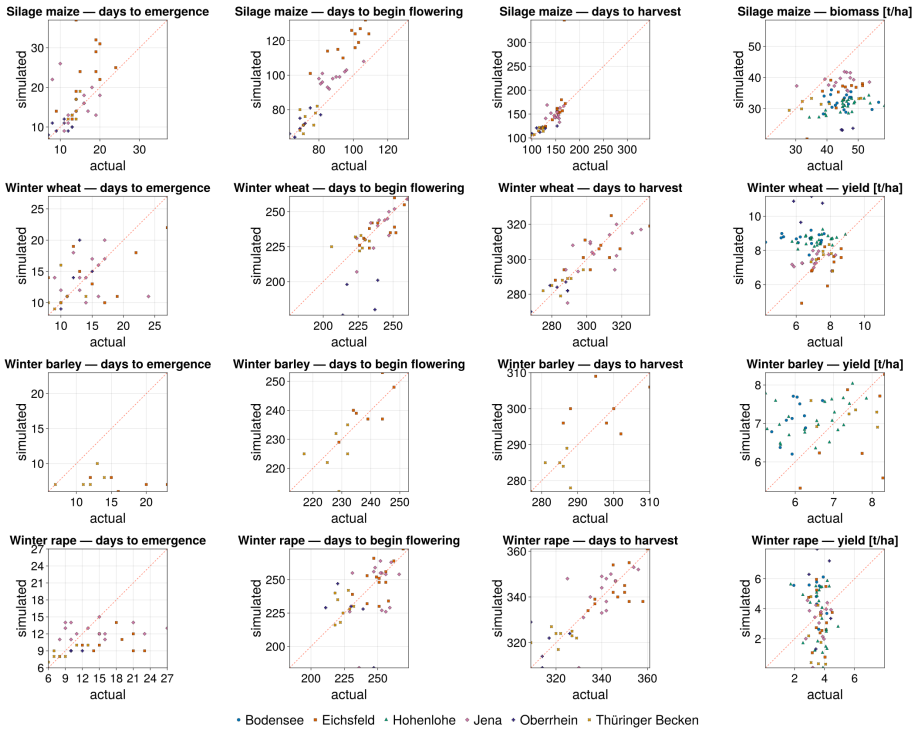

Figure 2: Output of the AquaCrop model compared to empirical data from the study regions, shown here for the four main crop types. The red line is the $x = y$ line, i.e. points above the line are overestimated, points below the line underestimated by the model.

3.4 Results and diagnostics

Goodness of fit was evaluated by comparing simulated vs. observed (i) dates of key phenological stages and (ii) annual yields. In practice, diagnostics included time series plots of canopy cover, stage timing overlays, and simulated-vs-observed yield comparisons. Successful calibration is indicated by close agreement in stage timing and substantially reduced yield discrepancies at the regional scale.

4 Cross-validation of the AquaCrop parameter calibration

We performed cross-validation on the calibrated crop parameters from section 3 with a forward-in-time hold-out design applied independently to each pair of regions and crops. Earlier seasons form the fitting block, and later seasons form the validation block.

The cross-validation uses the Julia MLJ.jl (Blaom et al., 2020) package

with the custom fitting procedure described in section 3. As the parameter fitting procedure uses a randomised optimisation procedure, the validation was performed 100 times for each fold and results were averaged.

Reported metrics. We summarise validation with three complementary metrics: the root mean squared *proportional* error (RMSPE, %), the root mean squared error (RMSE, in $\mathrm{kg\,ha^{-1}}$), and Willmott's index of agreement (D, unitless).

RMSPE (%). This scale-free metric enables direct comparison across crops and sites:

$$\mathrm{RMSPE}\ (\%) \;=\; 100 \times \sqrt{\frac{1}{n}\sum_{i=1}^{n}\left(\frac{\hat{y}_i - y_i}{y_i}\right)^2}. \tag{5}$$

In the implementation, a small tolerance ε is added where needed to avoid division by zero (as implemented in MLJ.jl's `RootMeanSquaredProportionalError`).

RMSE ($\mathbf{kg\,ha^{-1}}$). This magnitude-sensitive metric is reported on the original scale of the response:

$$\mathrm{RMSE} \;=\; \sqrt{\frac{1}{n}\sum_{i=1}^{n}\big(\hat{y}_i - y_i\big)^2}. \tag{6}$$

Unlike RMSPE, RMSE is not scale-free; it quantifies average absolute error size in the same units as y (here, $\mathrm{kg\,ha^{-1}}$).

Willmott's index of agreement (IA). This bounded, unitless index measures the degree to which predictions approach observed values, relative to the variability in the observations:

$$\mathrm{IA} \;=\; 1 \;-\; \frac{\displaystyle\sum_{i=1}^{n}\big(\hat{y}_i - y_i\big)^2}{\displaystyle\sum_{i=1}^{n}\Big(\,\big|\hat{y}_i - \bar{y}\big| + \big|y_i - \bar{y}\big|\,\Big)^2}, \qquad \bar{y} \;=\; \frac{1}{n}\sum_{i=1}^{n} y_i. \tag{7}$$

Values range from 0 (no agreement) to 1 (perfect agreement). In practice, IA complements RMSE/RMSPE by normalising errors to the observed variability.

5 Discussion

Across stations, model performance varies considerably by crop. Silage maize and winter wheat achieve moderate agreement and proportional error (e.g., IA around 0.47–0.52 for maize and 0.46–0.54 for wheat; RMSPE

Table 3: Cross-validation results for RMSE, Willmott's index of agreement (IA), and RMSPE by station and crop

Station	Crop	RMSE (kg/ha)	Index of agreement (IA)	RMSPE (%)
Eichsfeld	Silage maize	13 117.5	0.358	27.1
	Winter barley	—	—	—
	Winter rapeseed	1783.2	0.334	53.1
	Winter wheat	2647.1	0.184	33.2
Jena	Silage maize	6051.7	0.471	14.9
	Winter barley	—	—	—
	Winter rapeseed	626.3	0.775	19.6
	Winter wheat	671.3	0.460	10.2
Thüringer Becken	Silage maize	11 213.3	0.521	23.8
	Winter barley	680.8	0.648	8.8
	Winter rapeseed	10 603.8	0.003	296.4
	Winter wheat	608.1	0.542	8.0

— indicates cross-validation failed.

typically 10–27%), suggesting that inter-annual variability is captured to a useful degree on these crops. Winter barley performs well where evaluated (low RMSPE at Thüringer Becken), but cross-validation did not converge at two stations, indicating sensitivity to data density or local management assumptions. In contrast, winter oilseed rape exhibits very poor fit (RMSPE ∼300% at Thüringer Becken and IA ≈ 0), possibly pointing to structural issues with the model or the parameter-fitting procedure, and/or missing drivers that are crop-specific (e.g., cultivar turnover, pest/disease stress, or nitrogen dynamics).

These patterns are broadly consistent with a Central European multi-model assessment of four crops (Kostková et al., 2021), which reported the highest accuracy for silage maize and the lowest for winter oilseed rape by the index of agreement, and also highlighted that yield prediction performance is markedly lower than for phenology. Their ensemble median (EnsMED) yield RMSEs were about 1861 $\mathrm{kg\,ha^{-1}}$ (silage maize), 1365 $\mathrm{kg\,ha^{-1}}$ (winter wheat), 1105 $\mathrm{kg\,ha^{-1}}$ (spring barley) and 969 $\mathrm{kg\,ha^{-1}}$ (winter oilseed rape), while for winter wheat the individual-model IA spanned roughly 0.24–0.60 with EnsMED near 0.48, ranges that bracket our wheat results and mirror our difficulty with oilseed rape. Differences in sites, periods, and inputs (including yield basis) limit quantitative comparability, but the qualitative ranking of crops and the gap between phenology and yield skill are aligned with that study's conclusions.

References

Barry, S. (2025). *rdwd: Direct Access to the German Weather Service (DWD) Climate Data*. https://github.com/brry/rdwd. R package / GitHub repository. Accessed 2024–2025.

Blaom, Anthony D. et al. (2020). "MLJ: A Julia package for composable machine learning". In: *Journal of Open Source Software* 5.55, p. 2704. DOI: 10.21105/joss.02704. URL: https://doi.org/10.21105/joss.02704.

Díaz Iturry, Gabriel et al. (2025). "AquaCrop.jl: A Process-Based Model of Crop Growth". In: *Journal of Open Source Software* 10.110, p. 7944. DOI: https://doi.org/10.21105/joss.07944.

Kostková, Markéta et al. (2021). "Performance of 13 crop simulation models and their ensemble for simulating four field crops in Central Europe". In: *The Journal of Agricultural Science* 159.1-2, pp. 69–89. DOI: 10.1017/S0021859621000216.

Raes, D. et al. (2009). "AquaCrop—The FAO crop model to simulate yield response to water: II. Main algorithms and software description". In: *Agronomy Journal* 101.3, pp. 438–447. DOI: 10.2134/agronj2008.0140s.

Reichenau, T. G. et al. (2020). "A comprehensive dataset of vegetation states, fluxes of matter and energy, weather, agricultural management, and soil properties from intensively monitored crop sites in western Germany". In: *Earth System Science Data* 12.4, pp. 2333–2364. DOI: 10.5194/essd-12-2333-2020.

Statistical Office of Thuringia (2025). *Crop Area and Yield Statistics (online database)*. https://statistik.thueringen.de/datenbank/TabAnzeige.asp?tabelle=kr000516%%7C%%7C. Accessed 2024–2025.

Steduto, P. et al. (2009). "AquaCrop—The FAO crop model to simulate yield response to water: I. Concepts and underlying principles". In: *Agronomy Journal* 101.3, pp. 426–437. DOI: 10.2134/agronj2008.0139s.

Topping, C. J. and X. Duan (2024). "ALMaSS Landscape and Farming Simulation: software classes and methods". In: *Food and Ecological Systems Modelling Journal* 5, e121215. DOI: https://doi.org/10.3897/fmj.5.121215.

Topping, C. J., T. S. Hansen, et al. (2003). "ALMaSS, an agent-based modelling system for animals in agricultural landscapes". In: *Ecological Modelling* 167.1–2, pp. 65–82. DOI: 10.1016/S0304-3800(03)00173-4.

Appendix C: Animal Species Models

Part I.

Skylark (*Alauda arvensis*)

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

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

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.

2. Entities, state variables, and scales

2.1. Landscape

The simulated landscape consists of a vector map, each element of which is assigned one of 32 land cover classes as well as a soil type. Landscape elements may be attached to a crop model instance, 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. The maps have an accuracy of +/- 3 m and an extent of 270–1560 km^2 (depending on the modelled region).

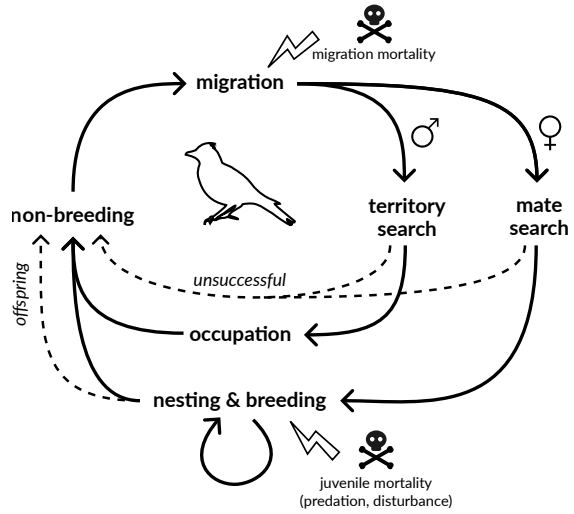

Figure 1: Phase diagram of the skylark model.

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 or 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 polygon defining the area 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.

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 stays in its territory until the breeding season is over. Then it changes its phase to nonbreeding. (Note: real 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 likelihood 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.

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 (for justification, 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 often 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 (Poulsen et al., 1998).
- **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, with each next step chosen by selecting a random suitable habitat within a radius of 500 m. 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.

Persefone.jl includes a seed parameter, which is used to initialise the random number generator. This can be used to ensure reproducibility (simulation runs of the same model version with the same parameter values will be identical in outcome). The model saves the configuration file used for a simulation run alongside the run's output data, so that all runs can be repeated if necessary.

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

The model collects three sets of data. The first set gathers daily abundance data, listing the number of individuals currently alive in each life phase. The second gathers information about each nesting attempt, with the date, habitat choice, and territory size. The third records the cause of death of all animals that die during the simulation. All data are saved as CSV files, with several figures automatically created from these at the end of the run.

5. Initialisation

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

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

6. Input data

The general input to Persefone.jl (i.e. land use maps and weather data) is described in the user manual and online documentation. The species parameters associated with the skylark model are listed and explained in Table 1.

7. Submodels

7.1. Territory formation

In the `territorysearch` phase, male skylarks look for a breeding territory upon returning from migration. If they still have a designated territory from the previous year, they occupy this again. Otherwise, they take random steps through the landscape until they find a territory (one step per day of `movementrange` length). Each step, they inspect the

Table 1: Species parameters and their values for the *Alauda arvensis* model.

Parameter	Value used	Explanation	References	Values tested
eggtime	11	Days spent in each	Glutz	
nestlingtime	9	juvenile life stage.	von Blotzheim	
fledglingtime	21		and Bauer (1985)	
egg-predation-mortality	0.03	Daily probability of dying in each juvenile life stage.	Delius (1965) and Jenny (1990)	
nestling-predation-mortality	0.03			
fledgling-predation-mortality	0.01			
firstyear-mortality	0.38	Probability of dying in the first year of life after fledging.	Delius (1965)	
migration-mortality	0.33	Probability of dying during on each winter migration.	Delius (1965)	
migration--departure	15 Sep – 1 Nov	Time period in which	Glutz	
migration--arrival	15 Feb – 1 Mar	adults leave for / arrive back from winter migration.	von Blotzheim and Bauer (1985)	
migrationdelay-females	15	Number of days which females return later than males from winter migration.	Glutz von Blotzheim and Bauer (1985)	
minimum-territory	5000 m^2	Minimum required size of a territory in ideal habitat.	Delius (1965)	
foragehabitat	Grassland OR BareGround OR Arable	Land cover types usable for foraging.	Glutz von Blotzheim and Bauer (1985)	
edgestructures	FOREST OR BUILDING OR Hedge OR Trees OR RockOrCliff	Land cover types with tall vertical structures that are avoided.	Glutz von Blotzheim and Bauer (1985)	

Table 2: Species parameters and their values for the *Alauda arvensis* model. (cont.)

Parameter	Value used	Explanation	References	Values tested
`mindistance-toedge`	60 m	Minimum required distance of suitable habitat to vertical structures.	Jenny (1990)	
`maxforageheight` `maxforagecover`	50 cm 70 %	Maximum preferred plant height and canopy cover for foraging.	Püttmanns et al. (2022)	
`nestingheight` `nestingcover`	15–45 cm 20–100 %	Range of preferred plant height and canopy cover for nesting.	Jenny (1990)	15–20 / 15-60 cm 20–50 %
`matefaith-fulness`	50 %	Probability that a female retains her partner from the previous year.	Jenny (1990)	
`nestingbegin` `nestingend`	10–20 Apr 15 Aug	Time in which nesting takes place.	Glutz von Blotzheim and Bauer (1985)	
`nestbuilding-time`	4–5	Number of days needed to build a nest.	Glutz von Blotzheim and Bauer (1985)	
`eggsperclutch`	2–5	Number of eggs laid per clutch.	Jenny (1990)	
`movement-range`	500 m	Movement distance per day when looking for territories, mates, or foraging outside the breeding season.		
`perception`	300 m	Visual range while moving (used in the `matesearch` and `nonbreeding` phases).		
`flocking-distance`	30 m	Approximate distance between animals in a flock (in the `nonbreeding` phase).		
`offfield-nesting`	false	Allow skylarks to nest in unmanaged areas of the landscape?		true

landscape in concentric circles around their location, adding square blocks of habitat to their putative territory until they have accumulated a sufficiently large effective habitat area. The area that each block contributes to the effective area is weighted by its forage quality, which is calculated as a function of its openness (bare ground is best, quality decreases as vegetation height and cover increase). If the required effective habitat area cannot be reached because of other territories in the surrounding, or an insufficiently large contiguous habitat area, the search breaks off and is repeated in a different location the next day.

7.2. Juvenile mortality

Juvenile mortality comes from two different sources: predation and disturbance (i.e. management). Predation is an age–dependent constant probability applied daily before chicks become independent, then one–off for the first summer upon independence. (Skylarks are only instantiated as code objects at independence, therefore applying first–year mortality as a one–off probability means that only those need to be instantiated that will survive the year, this saves processing time.) Apart from the constant background mortality of predation, the management actions of tillage and harvest also lead to juvenile death (100 % probability). Except for the first–year mortality, which is individual–specific, all other causes of juvenile mortality affect the whole brood, i.e. the whole nest is lost at once.

8. Testing & validation

The only critical parameters we found were `nestingheight` and `nestingcover`, as these determine where skylarks can build their nests. The original values tested here (15–25 cm and 20–50 %) resulted in rapid population loss, as the crop models predict values outside this range for most of the breeding season. After another look at the literature, we therefore raised the maximum nesting height to 45 cm, as different studies report different values here. We also raised the maximum nesting cover to 100 %, as AquaCrop and ALMaSS both predict very high canopy cover proportions very quickly, which would have precluded any nesting otherwise. (In addition, there were concerns about methodological mismatches between the values generated by our crop models and those measured in empirical studies, leading us to deprioritise this parameter.)

Two other parameters we changed from the initial settings where `offfieldnesting` and `limitterritory`. We disabled the former because we were observing a lot of skylarks

nesting in areas that were not under agricultural management in the model; something that we judged to be a modelling artefact rather than a real ecological effect. The latter we had initially enabled to curtail the creation of unusually large territories (>40 ha); however, a closer inspection of the literature showed that these sizes can in fact be reached, so we turned this setting off again.

With these described parameter adjustments, and good calibration of the AquaCrop model, the skylark model reproduced a range of complex ecological patterns without further changes needed.

Part II.
Marbled White (*Melanargia galathea*)

Melanargia galathea is a common grassland specialist butterfly. This animal model is one component of the animal submodel of Persefone.jl.

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

1. Purpose

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

2. Entities, state variables, and scales

2.1. Landscape

The simulated landscape consists of a vector map, each element of which is assigned one of 32 land cover classes as well as a soil type. Landscape elements may be attached to a crop model instance, 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. The maps have an accuracy of +/- 3 m and an extent of 270–1560 km^2 (depending on the modelled region).

The landscape is also associated with daily weather data, taken from the nearest weather station. This provides the daily mean temperature and daily precipitation as input to this species model (other variables are available, but not used).

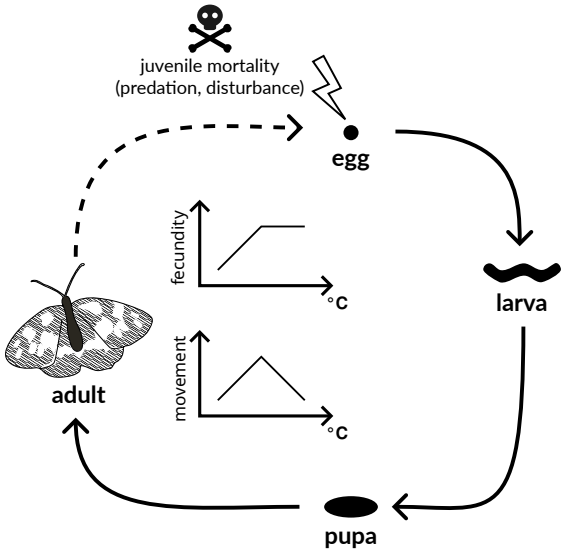

Figure 2: Phase diagram of the marbled model.

2.2. Animals

The simulated individuals (a.k.a. agents) are female marbled whites. (It is assumed that all females mate, and males therefore do not need to be simulated to capture population dynamics.) Each individual is characterised by the following variables:

- `ID` A unique identifier for this individual, which can be used to link it to its parent and its offspring.
- `phase` The individual's current stage in the life cycle. May be one of: egg, larva, pupa, adult.
- `age` The individual's age in days.
- `position` The individual's position in the simulated landscape.

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:

- Juveniles (phases egg, larva, and pupa) check whether they are in a field that has been tilled or harvested, and if so, die with a certain probability (100 % for tillage, by default 0 % for harvest; the latter probability is configurable). They also check whether they are old enough to advance to the next phase. No movement or other

activity takes place.

- Adults move around quasi–randomly and lay eggs. Distance moved and number of eggs laid is temperature–dependent (for details, see below). Individuals die once they reach their maximum age, which is determined using a linear distribution at birth.

4. 4. Design concepts

4.1. Basic principles

This model assumes that marbled white distribution is primarily shaped by habitat availability, while abundance is most affected by weather. Thus, the model's representation of their biology focuses on habitat requirements and activity levels in response to weather. Accordingly, the only behavioural mechanisms that are simulated in the model are movement and oviposition.

Movement is assumed to be largely random, although with a strong preference for habitat that is suitable for oviposition (i.e. open semi–natural areas, extensively managed grassland, and fallows; cf. section 7.2). The representation of movement chosen here is coarser than that found in dedicated movement models of butterflies (e.g. Evans et al., 2019), as we are primarily interested in larger–scale processes (over seasons and kilometres) for which very fine–grained movement decisions (over seconds and meters) are less relevant.

The model assumes that marbled white population growth is limited by fecundity not mortality. Fecundity in turn is understood to be limited by the available time for oviposition, as determined by the daily weather (Gotthard et al., 2007). Therefore, a central part of the model is the calculation of the number of eggs that can be laid in a given day (section 7.1).

Agricultural management is of secondary importance to the modelled population trends, but can still influence them on two levels. The first concerns the availability of habitat, as the amount of fallow land and the proportion of extensively managed grassland can vary in different model scenarios. The second is additional mortality caused by tillage and harvest/mowing, which can be configured but is generally assumed to be negligible (Ebert & Rennwald, 1991).

4.2. Emergence

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

1. **Individual lifetime stats:** The combination of individual behaviour, weather, and landscape structure lead to characteristic distributions of different measured individual-level variables (cf. section 4.11), which can be compared to known distributions of these variables from the literature (e.g. Baguette et al., 2000; Vandewoestijne et al., 2004). These variables include the individual fecundity (i.e. total eggs laid per female), the per–patch population density experienced by each individual, the lifetime displacement (i.e. distance of the location at death from the birth location), and the relative use of different habitats while moving through the landscape.
2. **Population trends:** Kühn et al. (2024) show the population trends of *Melanargia galathea* in Germany for the period 2006–2023. In the first time period (2006–2015), this shows a strongly fluctuating, overall slightly decreasing trend, followed by a marked increase after 2015. Comparing this to the summer temperatures over these years shows that the abundance in one year is correlated with the mean temperature of the previous summer (at least until 2016), a pattern that was also shown for British butterflies by Roy et al. (2001). In addition, the introduction of the CAP Greening measures in 2015 may have contributed to the positive trend over the past years.

4.3. Adaptation

Marbled whites respond to the landscape by mostly restricting their movement to suitable habitat. The movement rules were chosen to reproduce lifetime and landscape–scale patterns, rather than conform to mechanistic principles of movement behaviour (see sections 7.2, 8.2).

4.4. Objectives

Marbled whites' only "objective" is to stay on or close to suitable habitat, in order to allow them to lay their eggs (see section 7).

4.5. Learning

The model includes no learning by individuals.

4.6. Prediction

The model includes no predictions by individuals.

4.7. Sensing

Marbled whites can perceive the landscape structure in a given radius around them (land cover, crop type, vegetation height and cover). They can see nearby conspecifics and sense the day's weather (temperature and precipitation).

4.8. Interaction

By default, there are no interactions between individual marbled whites. Optionally, a maximum local population density can be set, with individuals less likely to visit a habitat patch as the number of conspecifics already on it approaches this maximum.

4.9. Stochasticity

Stochasticity is used when modelling mortality and movement. Juvenile mortality is modelled as a one–time probability of death, applied when an adult butterfly lays an egg. (Only eggs that pass this probability check and will therefore mature into adults are actually instantiated, in order to save computational resources.) The amount of time in days that an individual spends in each development phase is drawn at random during its initialisation, using a normal distribution for the larval phases and a linear distribution for the adult lifetime. Movement proceeds in quasi–random jumps (for details see below).

Persefone.jl includes a seed parameter, which is used to initialise the random number generator. This can be used to ensure reproducibility (simulation runs of the same model version with the same parameter values will be identical in outcome). The model saves the configuration file used for a simulation run alongside the run's output data, so that all runs can be repeated if necessary.

4.10. Collectives

The model includes no collectives.

4.11. Observation

The model collects three sets of data. The first set gathers daily abundance data, listing the number of individuals currently alive in each life phase. The second set is updated every time an individual dies, and shows that individual's lifetime values of fecundity, displacement, and habitat use, as well as the local population density at its last location. The third set is updated annually and lists that year's adult abundance, average fecundity, and average temperature. All data are saved as CSV files, with several figures automatically created from these at the end of the run.

5. Initialisation

The simulation is initialised with one individual per hectare placed at random on a grassland habitat. The starting population density can be modified with the `initialdensity` parameter. Individuals are created as eggs (whether at birth or at initialisation), and then calculate how much time they will spend in each juvenile phase in order to ensure eclosure during the known flying period. (As juvenile phases are functionally identical in the model, the amount of time in each is irrelevant for model outcomes.)

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

6. Input data

The general input to Persefone (i.e. land use maps and weather data) is described in the user manual and online documentation. The species parameters associated with the marbled white model are listed and explained in Table 1.

7. Submodels

7.1. Weather

Temperature affects both the distance moved and the number of eggs laid each day. The optimal temperature is taken to be the midway point between the species' minimum and maximum temperatures (i.e. 24°C). Outside the species' temperature range (18–30 °C), neither movement nor oviposition take place. Within that range, the number of steps each day peaks at the optimum temperature and declines linearly on either side of it (cf. Evans et al., 2019). The number of eggs laid declines linearly if the temperature is

Table 1: Species parameters and their values for the *Melanargia galathea* model.

Parameter	**Value used**	**Explanation**	**References**	**Values tested**
`activetemp`	16–32 °C	Temperature range suitable for activity (oviposition and movement).	Ebert and Rennwald (1991), Evans et al. (2019) and Gotthard et al. (2007)	18–30 °C, 20–28 °C
`rainactive`	true	Allow activity on rainy days?		false
`maxsteps-perday`	100	Number of movement steps per day under optimum weather conditions.		50
`habitat-preference`	0.95	Probability of declining to move to a non-habitat location.		
`perception`	100 m	Maximum distance for a single movement step.	Cant et al. (2005)	
`maxdensity`	2000	Maximum local population density; probability of avoiding a location increases linearly up to this value.	Schulte et al. (2007)	10, 100
`maxeggs-perday`	5	Number of eggs laid per day under optimal weather conditions.	Reinhardt et al. (2007)	3–12
`initial-habitat`	Grassland	Habitat that individuals are created in at simulation start.		

Table 2: Species parameters and their values for the *Melanargia galathea* model. (cont.)

Parameter	Value used	Explanation	Justification	Values tested
flying-habitat	ExtensiveGrassland OR Fallow OR FlowerStrip OR Hedge OR Scrub OR Heath OR Orchard OR NaturalGrass	Description of habitat suitable for moving through.	Ebert and Rennwald (1991)	
oviposition-habitat	(ExtensiveGrassland AND height>=30cm AND height<=60cm) OR Fallow OR FlowerStrip OR NaturalGrass	Description of habitat suitable for oviposition.	Vandewoestijne et al. (2004)	
oviposition	"linear"	Oviposition-weather algorithm to use (see sections 7.1, 8.1).		"constant"
eggtime	18–22	Range of possible times (in days) spent in each juvenile phase.	Reinhardt et al. (2007)	
larvatime	290–320			
pupatime	16–29			
maxadult-time	34 days	Maximum life expectancy after eclosure.	Reinhardt et al. (2007)	
maturation-time	5–8 days	Range of possible times between eclosure and first oviposition.	Reinhardt et al. (2007)	
juvenile-mortality	0.96	Probability of individual dying before eclosure.	Dennis (1992)	0.88–0.99
mowing-mortality	0.0	Probability of an individual dying if its location is mown or harvested.	Ebert and Rennwald (1991)	0.1, 0.5
eclosure-period	15th June – 15th August	Range of dates in which eclosure takes place.	Ebert and Rennwald (1991)	
enforce-flyingperiod	true	Ensure all individuals eclose in the empirically observed flying period.		false

below the optimum, but stays stable above it (cf. Gotthard et al., 2007). The daily mean temperatures are used as the basis for calculation, as using the maximum temperature was shown to produce wrong model results during heat waves. In addition, the model can be configured so that no activity takes place on days with rainfall (i.e. precipitation > 0, configured with `rainactive`).

7.2. Movement

Adults move a given number of steps each day, depending on the temperature (see above). Each step, an individual selects a random location within its perceptual range of 100 m. If the location is suitable habitat, i.e. either semi–natural or extensive grassland with plant heights within the required range (30–60 cm), it moves there. In this case, the individual also lays an egg if it has not yet laid all its eggs for that day. If the location is not suitable habitat, it may nevertheless move there with a certain probability (given by the `habitatpreference` parameter). Otherwise, it looks at the next randomly chosen location in its perceptual range and repeats the procedure.

8. Testing & validation

8.1. Alternate weather parameters

We tested different temperature ranges (16–32 °C, 18–30 °C, 20–28 °C), with and without rain sensitivity. We also tested an alternative calculation, in which the number of eggs per day is assumed to stay constant across the acceptable temperature range. However, in each of these scenarios, most populations died out due to insufficient reproduction. Thus, the option 16–32 °C without rain sensitivity was found to reproduce the selected patterns best.

8.2. Alternate movement

We tested an alternative movement submodel, but this gave worse pattern fits than the random movement described above:

Adults move a given number of steps each day, depending on the temperature (see below). Each step, an individual scans its surroundings in concentric circles, looking for the closest spot that offers suitable habitat which it hasn't visited today. (Depending on the `habitatpreference` and `selfavoidance` parameters, spots that are not suitable habitat or have been visited before may also be selected.) Spots with higher population densities

are more likely to be avoided (avoidance increases linearly up to 100% at `maxdensity`). If no suitable spot is found, the individual moves to a random location on the periphery of its vision. After each step, if it is in a suitable habitat, the individual lays an egg, up to the number determined by the temperature for that day.

8.3. Other parameters

We also tested different values of the parameters `habitatpreference`, `juvenilemortality`, `mowingmortality`, and `maxeggsperday`. Juvenile mortality proved to have the strongest influence and the highest sensitivity. For each parameter, we selected values to give the optimal fit in the pattern-oriented modelling process (see main text).

As explained in the main text, the Oberrhein region produced an order of magnitude more population growth than any of the other regions, and was therefore excluded from the main analysis as an outlier. Fig. 3 shows the results of the analysis with the Oberrhein included.

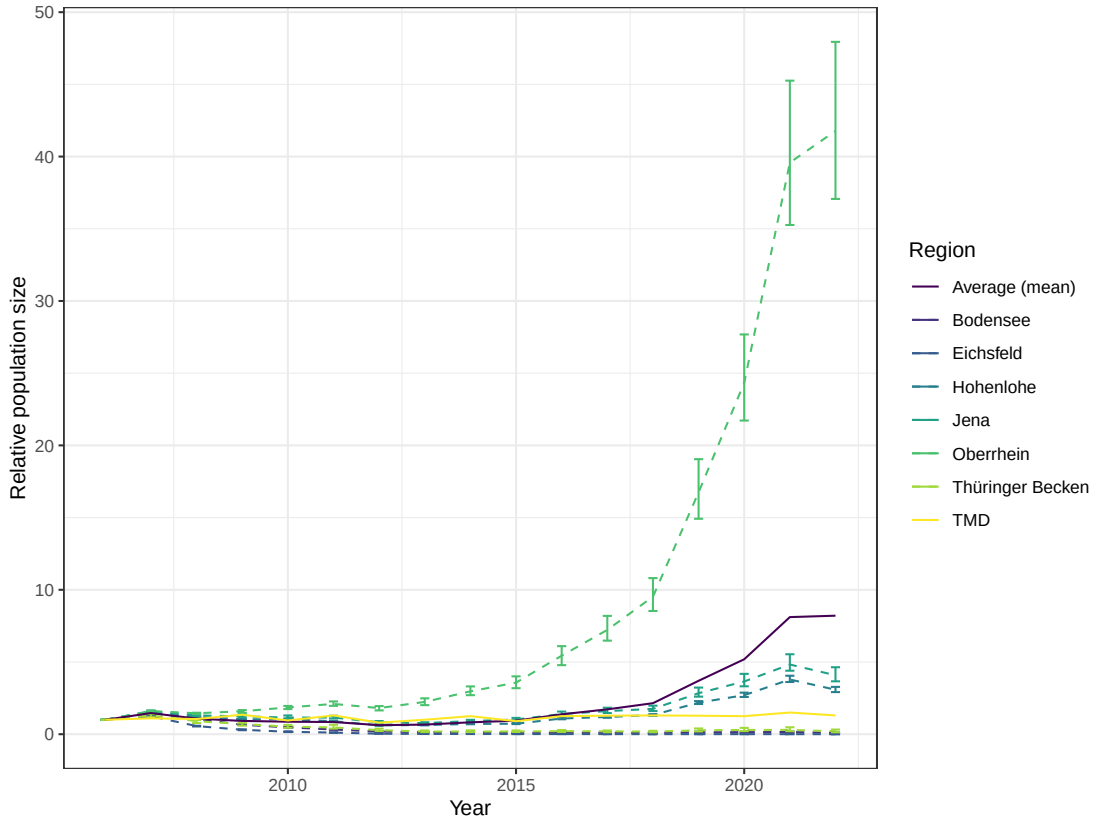

Figure 3: Results of the butterfly simulation experiment including the Oberrhein region. This corresponds to Fig. 8 in the main text, where Oberrhein was excluded as an outlier.

References

Baguette, M., Petit, S., & Quéva, F. (2000). Population spatial structure and migration of three butterfly species within the same habitat network: Consequences for conservation. *Journal of Applied Ecology*, *37*(1), 100–108. https://doi.org/10.1046/j.1365-2664.2000.00478.x

Cant, E., Smith, A., Reynolds, D., & Osborne, J. (2005). Tracking butterfly flight paths across the landscape with harmonic radar. *Proceedings of the Royal Society B: Biological Sciences*, *272*(1565), 785–790. https://doi.org/10.1098/rspb.2004.3002

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

Dennis, R. L. (Ed.). (1992). *The ecology of butterflies in Britain*. Oxford University Press.

Ebert, G., & Rennwald, E. (1991). *Die Schmetterlinge Baden-Württembergs, Bd.2, Tagfalter: Satyridae, Libytheidae, Lycaenidae, Hesperiidae*. Verlag Eugen Ulmer.

Evans, L. C., Sibly, R. M., Thorbek, P., Sims, I., Oliver, T. H., & Walters, R. J. (2019). Integrating the influence of weather into mechanistic models of butterfly movement. *Movement Ecology*, *7*(1), 24. https://doi.org/10.1186/s40462-019-0171-7

Glutz von Blotzheim, U. N., & Bauer, K. M. (Eds.). (1985). *Handbuch der Vögel Mitteleuropas (10,1 : Passeriformes ; T. 1); [Alaudidae - Hirundinidae]*. AULA-Verl.

Gotthard, K., Berger, D., & Walters, R. (2007). What Keeps Insects Small? Time Limitation during Oviposition Reduces the Fecundity Benefit of Female Size in a Butterfly. *The American Naturalist*, *169*(6), 768–779. https://doi.org/10.1086/516651

Grimm, V., Berger, U., Bastiansen, F., Eliassen, S., Ginot, V., Giske, J., Goss-Custard, J., Grand, T., Heinz, S. K., Huse, G., Huth, A., Jepsen, J. U., Jørgensen, C., Mooij, W. M., Müller, B., Pe'er, G., Piou, C., Railsback, S. F., Robbins, A. M., ... DeAngelis, D. L. (2006). A standard protocol for describing individual-based and agent-based models. *Ecological Modelling*, *198*(1–2), 115–126. https://doi.org/10.1016/j.ecolmodel.2006.04.023

Grimm, V., Berger, U., DeAngelis, D. L., Polhill, J. G., Giske, J., & Railsback, S. F. (2010). The ODD protocol : A review and first update. *Ecological Modelling*, *221*, 2760–2768. https://doi.org/10.1016/j.ecolmodel.2010.08.019

Grimm, V., Railsback, S. F., Vincenot, C. E., Berger, U., Gallagher, C., DeAngelis, D. L., Edmonds, B., Ge, J., Giske, J., Groeneveld, J., Johnston, A. S. A., Milles, A., Nabe-Nielsen, J., Polhill, J. G., Radchuk, V., Rohwäder, M.-S., Stillman, R. A., Thiele, J. C., & Ayllón, D. (2020). The ODD Protocol for Describing Agent-Based and Other Simulation Models: A Second Update to Improve Clarity, Replication,

and Structural Realism. *Journal of Artificial Societies and Social Simulation*, *23*(2), 7. https://doi.org/10.18564/jasss.4259

Jenny, M. (1990). Territorialität und Brutbiologie der FeldlercheAlauda arvensis in einer intensiv genutzten Agrarlandschaft. *Journal für Ornithologie*, *131*(3), 241–265. https://doi.org/10.1007/BF01640998

Kühn, E., Musche, M., Harpke, A., Feldmann, R., & Settele, J. (2024). Tagfalter-Monitoring Deutschland: Auswertung 2005-2023. *Oedippus*, *42*, 12–45. https://www.ufz.de/export/data/6/298835_298188_Oedippus_42_klein.pdf

Poulsen, J. G., Sotherton, N. W., & Aebischer, N. J. (1998). Comparative nesting and feeding ecology of skylarks Alauda arvensis on arable farmland in southern England with special reference to set-aside. *Journal of Applied Ecology*, *35*(1), 131–147. https://doi.org/10.1046/j.1365-2664.1998.00289.x

Püttmanns, M., Böttges, L., Filla, T., Lehmann, F., Martens, A. S., Siegel, F., Sippel, A., von Bassi, M., Balkenhol, N., Waltert, M., & Gottschalk, E. (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

Reinhardt, R., Sbieschne, H., Settele, J., Fischer, U., & Fiedler, G. (2007). *Tagfalter von Sachsen*. Entomofauna Saxonica.

Roy, D. B., Rothery, P., Moss, D., Pollard, E., & Thomas, J. A. (2001). Butterfly numbers and weather: Predicting historical trends in abundance and the future effects of climate change. *Journal of Animal Ecology*, *70*(2), 201–217. https://doi.org/10.1111/j.1365-2656.2001.00480.x

Schulte, T., Eller, O., Niehuis, M., & Rennwald, E. (2007, July 12). *Die Tagfalter der Pfalz - Band 2*. Gesellschaft für Naturschutz und Ornithologie Rheinland-Pfalz.

Vandewoestijne, S., Martin, T., Liégeois, S., & Baguette, M. (2004). Dispersal, landscape occupancy and population structure in the butterfly *Melanargia galathea*. *Basic and Applied Ecology*, *5*(6), 581–591. https://doi.org/10.1016/j.baae.2004.07.004