Persefone.jl: Modelling Biodiversity in Dynamic Agricultural Landscapes

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

Agricultural landscapes are highly dynamic, constantly changing in space and time due to the effects of farm management and environmental factors. To forecast the effects of changes in agricultural systems on biodiversity, we need to understand these landscape dynamics and how they impact different species. Here, we present Persefone.jl, an open-source process-based model of agricultural landscapes and animal species. The model simulates farm management, crop growth, and two wildlife animal species (skylark *Alauda arvensis* and marbled white *Melanargia galathea*) using daily time steps in three German farming regions. We describe the model and show that it replicates multiple qualitative and quantitative empirical patterns, as well as discussing design principles and limitations of the software. Persefone.jl is a modular and extensible agroecological model that can be used to study population dynamics of farmland wildlife species, and evaluate the biodiversity impacts of new policies or other changes to agricultural systems.

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

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

[2] Institute of Biodiversity, 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 Introduction

Farm management affects biodiversity in multiple ways. Some effects are direct, such as disturbances created by tillage, harvest, or pesticide application. Others are indirect, as agriculture shapes a landscape over space and time, for instance through the choice of crop rotation, the creation or removal of semi-natural habitat, or changes in nutrient availability. Together, these effects create spatiotemporal patterns of resource availability and disturbance that influence all non-domestic species living in agricultural landscapes (Vasseur et al., 2013).

In Europe, widespread declines of farmland species have been documented for taxa such as birds and butterflies (e.g. Rigal et al., 2023; van Swaay et al., 2019). These have prompted numerous conservation efforts seeking to establish "wildlife-friendly farming" (Batáry et al., 2015; Pywell et al., 2012). Many of these efforts are conducted at a policy level, using a combination of regulations and subsidies to enforce or encourage biodiversity-friendly practices. Of particular importance, due to its scope and volume, is the EU's Common Agricultural Policy (CAP), although its practical benefits have been mixed (Pe'er et al., 2014; Pe'er et al., 2020).

From an ecological perspective, there are at least three challenges associated with the design of effective agri-environmental measures. First, the measures must fit in with all the other things farmers do: they must be practicable, and not countered by other management practices (Hölting et al., 2022). Second, they must take into account the varying ecological requirements and behavioural responses of different target species (Vickery et al., 2004). Third, they should keep in mind the landscape context, as different spatial configurations may make local measures more or less effective (le Clech et al., 2024).

Simulation models can help to 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). One reason for this is that there are very few models available that simulate the impact of farm management on biodiversity (e.g. Guillem et al., 2015; Topping et al., 2003; see Vedder et al. *in review* for a review).

Here we present Persefone.jl, a model of animal populations in dynamic agricultural landscapes. 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 model the spatiotemporal population dynamics of its target species in response to environment and management. Its aim is to further ecological research into

the interactions between agriculture and biodiversity, and to provide a platform for rapid policy assessment in the context of (particularly German) agricultural landscapes.

2 Methods

2.1 Model description

Persefone.jl simulates farm management and crop growth at the landscape scale, using remote-sensing input maps for three different regions in Germany. Within these dynamically changing landscapes, the model can simulate multiple animal species. So far, the skylark, *Alauda arvensis*, and marbled white, *Melanargia galathea*, have been implemented. The model is designed to be transferable to other regions and expandable to further management scenarios and animal species.

The software is open-source and available online at https://persefone-model.eu. It is implemented in Julia, a programming language designed for performant scientific computing (Bezanson et al., 2017). 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 locally, and a simple graphical user interface is available for this purpose. For more details, see the user manual in Appendix A.

The model is constructed using a modular architecture (Vedder et al., 2024), and consists of an infrastructural core as well as four model components (Fig. 1). These will be described in the following sections.

2.1.1 World component

The world component is responsible for reading in environmental input data for the simulated region. These are satellite-based land cover maps (10 m resolution), field geometries, soil type maps, and daily weather data. All required input data are publicly available for Germany (Table 1), and the model website documents how to acquire and process them for use with the software.

Persefone.jl was developed within the research project CAP4GI, which worked together with farmers in six regions in Germany to develop recommendations for novel agri-environment measures in the CAP (Velten et al., 2023). The model has been set up to simulate the three Thuringian regions in this project: Jena, Eichsfeld, and the Thuringian Basin. These range in size between 270 km^2 and 370 km^2 and form a gradient of increasing land use intensity (based on the proportions of agricultural area and grassland; Fig.

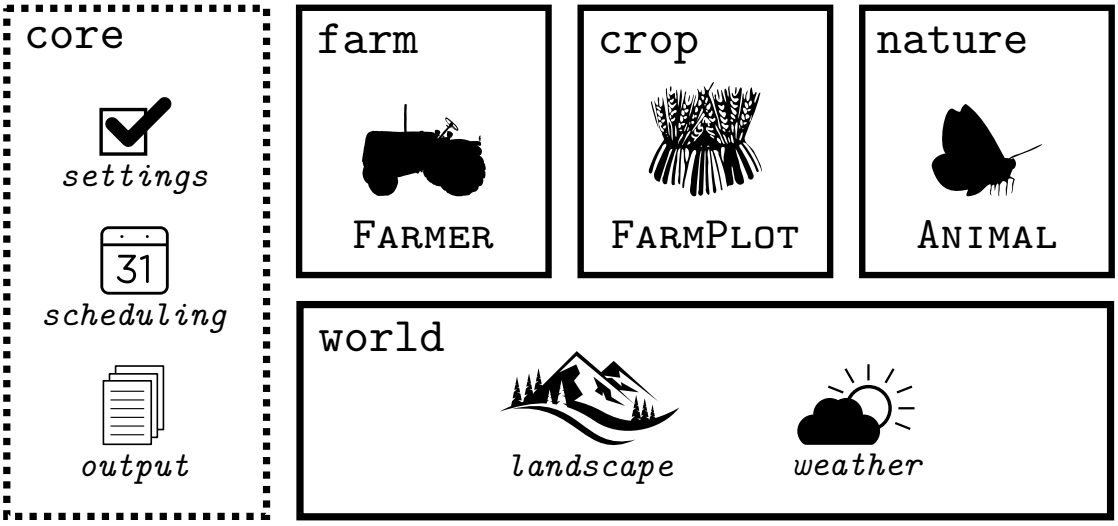

Figure 1: Model structure of Persefone.jl. The "core" component provides basic software infrastructure such as the handling of configuration files, the scheduling of processes, or utility functions for data output. The "world" component provides the necessary environmental data for the rest of the model, including maps and weather. The "farm", "crop", and "animal" components each simulate one agent type, the actions and interactions of which produce the model output.

2). The model runs with a daily time step and can simulate the years 1991 – 2023.

2.1.2 Farm component

The farm component 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.

Based on economic survey data from the three study regions (G. Theilen, *pers. comm.*), we selected a typical crop rotation that is carried out on all arable fields (with a randomised starting point): oilseed rape, winter wheat, silage maize, winter barley. 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, the management practices that are explicitly simulated are sowing and harvest/mowing. The dates on which these take place are partly taken from the literature, and partly calculated by the crop model (see below).

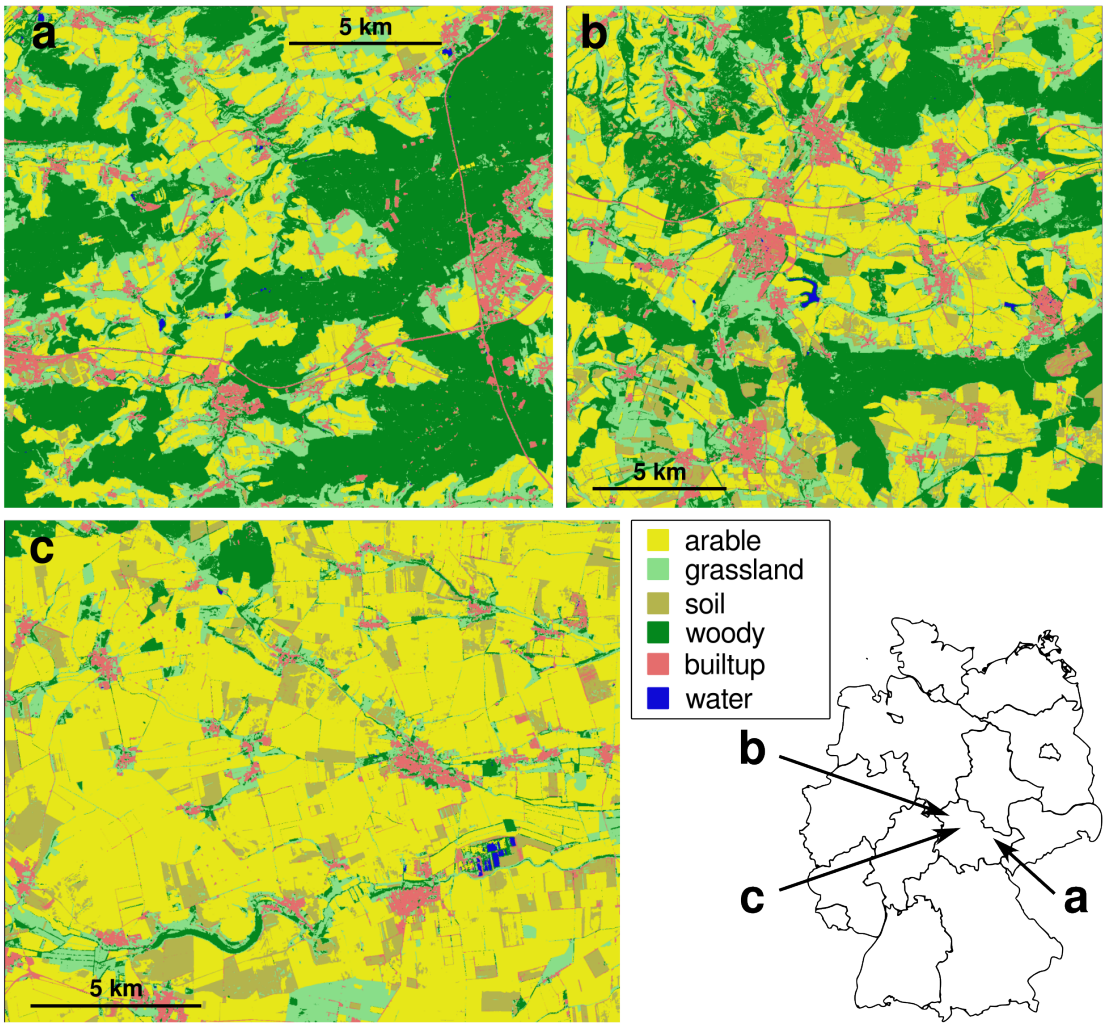

Figure 2: Regions simulated by the model: a) Jena, b) Eichsfeld, c) Thuringian Basin. Inset shows location of regions on a map of Germany. Landcover maps by mundialis GmbH & Co. KG (2021).

2.1.3 Crop component

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

The crop component provides a `FarmPlot` entity, which is initialised for every LPIS-registered field in the simulated region. The farm manager (see above) decides when the field is to be sown with a given crop, and when it is to be harvested/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 or to calculate habitat suitability.

To simulate these variables, Persefone.jl uses two different crop models, depending on the crop type. The primary crop model is AquaCrop, originally developed by the FAO and translated into Julia for the purposes of our project (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 used for numerous crops worldwide and is known to be quite reliable (Kostková et al., 2021; Mialyk et al., 2024). However, parameterising it requires regionally-specific crop growth data, which for our study regions were only available for our four main crop types (oilseed rape, winter wheat, silage maize, winter barley).

Therefore, we complemented 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 simple correlative model, predicting plant growth based on growing-degree days (i.e. temperature) and time of the year. While it is much less exact than AquaCrop, it has parameters available for a wide range of crop types, and can also simulate grassland and some non-crop vegetation types. We therefore use it for grassland, and in (currently unused) scenarios with different crop rotations to the one described above.

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

2.1.4 Animal component

It is the animal component that produces the main ecological output of Persefone.jl, namely the abundance and distribution of the several target species over time and space. To do so, it models the behaviour of individual animals in the changing landscapes created by the other components.

Each target species is represented by a separate individual-based model, though all species models can access the full state of the current simulation, and make use of a common set of utility functions. Each species is defined as a series of "life phases", i.e. functions that determine an individual's daily behaviour during one part of its life cycle (e.g. larva, or breeding adult). Individuals can interact with other individuals, including those of different species.

So far, two species have been implemented: the skylark *Alauda arvensis* and the marbled white butterfly *Melanargia galathea*. In both species, model design was kept deliberately simple, concentrating mainly on the environmental factors with the largest demographic impact, and the behavioural patterns directly affected by landscape and management. Other factors and behavioural patterns were ignored or strongly simplified, in order to keep the models to the minimum necessary complexity (Sun et al., 2016).

The skylark is a common and charismatic species of agricultural landscapes, which breeds 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 grains, 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).

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 growth affects territory and nesting site choice, and harvest/mowing is

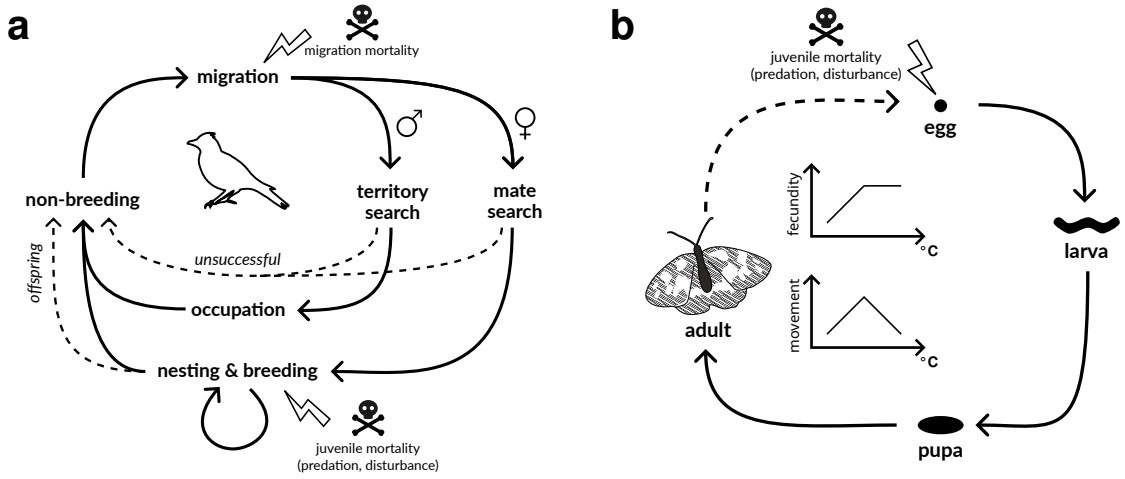

Figure 3: Animal model phase charts: a) Skylark *Alauda arvensis*, b) Marbled White *Melanargia galathea*.

an important cause of mortality.

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 intensive 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. As eggs and larvae develop on the ground, mortality from mowing is low; most is caused by predation (represented stochastically in the model). Thus, their main interaction with farm management is indirect, as they avoid grassland that has been fertilised or recently mown.

Detailed descriptions of both animal models are provided in Appendix C, following the ODD format (Grimm et al., 2006, 2020).

2.2 Model validation

Throughout the modelling process, we employed multiple techniques to ensure that the individual components and the complete model are adequate for their intended purpose (Troost et al., 2023). To this end, we define two purposes that Persefone.jl is intended to fulfil:

1. To represent the spatiotemporal landscape dynamics created by arable farming, including crop rotations, plant growth, and yield formation.
2. To predict the population dynamics of target species in response to environmental conditions and management, considering especially movement, reproduction, and mortality.

The first purpose is addressed by the world, farm, and crop components; the second by the animal component and its interactions with the rest of the model.

In addition to this scientific validation, we used software development best practices, including unit testing and code reviews, to verify the technical correctness of our software (Ropella et al., 2002; Vedder et al., 2021).

2.2.1 Landscape dynamics

The only function of the world component is to make empirical input data available to the rest of the model. As these data were previously validated by their respective creators (Table 1), we only verify the technical correctness of this component as described above, without further validation steps.

The function of the farm component is to carry out the crop rotation and sow, harvest, and mow fields at the appropriate time. This was verified using visual inspection of field-level summary statistics over time.

The crop component is intended to produce reasonable estimates of crop growth and phenology under the given environmental conditions. We used publicly available data sets of district-level yield and phenology data from our study regions to calibrate the AquaCrop model, then used cross-validation to test the robustness of the calculated parameters. The ALMaSS crop model is currently only used to generate grass growth patterns, the sufficient correctness of which we confirmed visually. For more details on the calibration and validation of the crop component, see Appendix B.

2.2.2 Population dynamics

As is common in individual-based models, we were most concerned with the structural validation of our target species models, as our aim was to predict (at least qualitative) population dynamics from individual-level mechanisms (Troost et al., 2023). For this, we used pattern-oriented modelling, identifying for each species a set of patterns at different spatial, temporal, and organisational scales and comparing model output to known values derived from literature (Grimm & Railsback, 2011).

Data	Description	Purpose	Source
Land cover	Satellite-derived raster map of six different land cover classes (10 m resolution, year 2020).	input	mundialis GmbH & Co. KG
Field geometries	Shape files of all fields registered in the EU Land Parcel Information System (LPIS; in Germany: InVeKoS).	input	Thüringer Landesamt für Landwirtschaft und Ländlichen Raum
Soil types	Shape file 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	Thüringer Landesamt für Statistik
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)

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.

For the skylarks, we looked at three different patterns. The first was the size of territories, which were generated procedurally in the model, and are known to vary depending on the landscape. The second was the choice of nesting habitat, which depends on the crops available and changes over the course of the breeding season. The third was the ecological trap described above, where the agricultural switch from spring to winter grains 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 for each region from 2011–2020.

For the marbled white, we considered several "simple" patterns: the number of eggs laid in a female's lifetime, the proportion of time spent moving through different habitats, the local population density, and the lifetime displacement distance. In addition, we selected a "complex" pattern, namely the population development as recorded by the German butterfly monitoring scheme (Kühn et al., 2024). This shows a strongly fluctuating, but overall 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 five 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

The model output shows how the internal landscape changes over time due to farm management and crop growth. Fig. 4 shows this at a landscape perspective, tracking how the proportion of different crops changes over the course of several years, and how the average plant height of each crop changes over the growing seasons. Fig. 5 gives a field-level perspective, showing the development of the other four AquaCrop output variables (canopy cover, biomass, phenological stage, yield) from sowing to harvest. Fig. 6 shows validation of the AquaCrop model for silage maize, depicting goodness-of-fit of four output variables against empirical data from the three study regions. (For more details on the crop model validation, see Appendix B.)

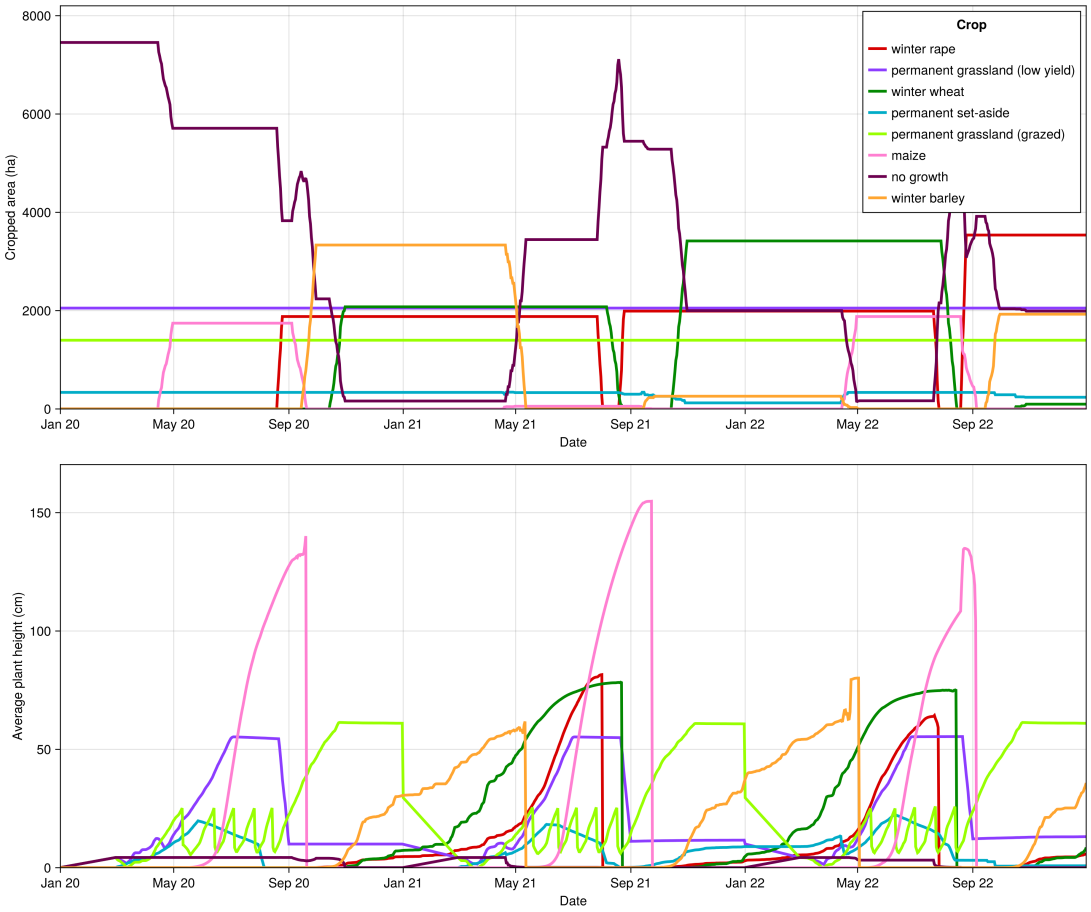

Figure 4: Field and crop dynamics over time, simulated in Jena from 2020–2022. Above: proportion of agricultural land sown with each crop type. Below: average plant height of crop. "No growth" refers to fields that are not currently sown with any crop (at the start of the simulation or between harvest and re-sowing).

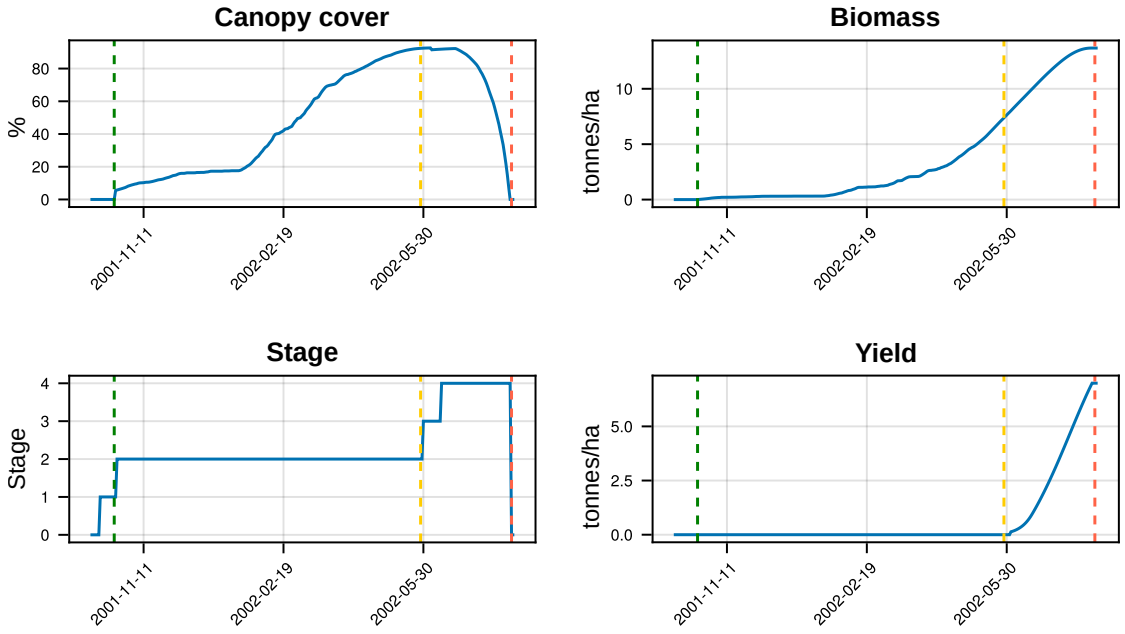

Figure 5: Output variables of the AquaCrop model, showing a simulation of winter wheat in the Jena region. Blue lines show model output over time. Dashed lines show empirically observed phenological dates from the region (green: emergence, yellow: flowering, red: harvest).

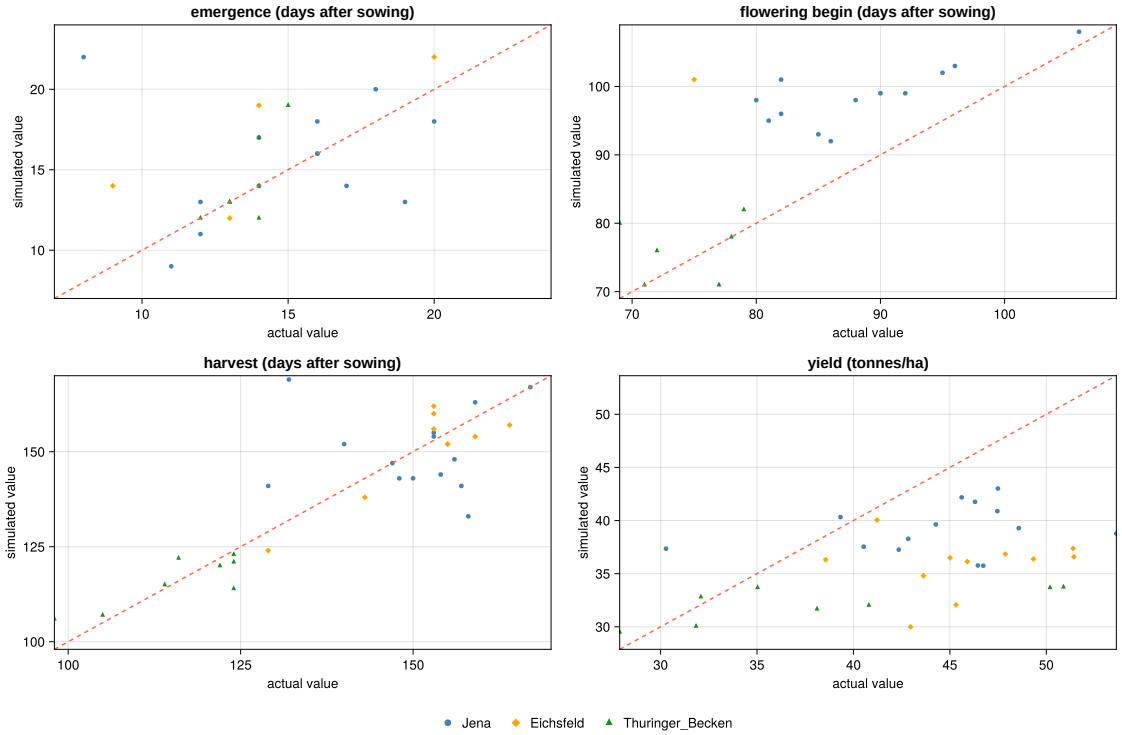

Figure 6: Output of the AquaCrop model compared to empirical data from the study regions, shown here for silage maize. The red line is the $x = y$ line, i.e. points above the line are overestimated, points below the line underestimated by the model.

The skylark model conforms well to the patterns against which we tested it. Territory sizes in the most intensive scenario ranged from 0.38–24.76 ha, with a median of 1.09 ha and an interquartile range of 0.81–1.56 ha. This compares favourably with the observations listed by Glutz von Blotzheim and Bauer (1985), which range between 0.17–46 ha, and are most commonly around 0.5–1.5 ha. We also observe the effect that territory sizes in extensively used farmland are smaller—the scenario with the lowest land use intensity gave an interquartile range of 0.74–1.26 ha and a median of 0.96 ha.

Likewise, the choice of nesting habitat over the breeding season (Fig. 7) closely follows the description of Jenny (1990). In his observations, as in our model, grassland is always a favoured habitat; winter barley is almost never used, as it grows too quickly; winter wheat still occurs in the early breading season, but not in the late; 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 (Fig. 8). Across regions, skylark population grow in the scenario with mostly extensive grassland usage and spring-sown crops, while they decline 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: skylarks in the almost entirely arable Thuringian Basin respond very strongly to spring or winter crops, but little to grassland usage intensity, while the response is more mixed in the other regions.

For the marbled white, the collected lifetime variables also correspond well to known literature values (Fig. 9). Fecundity peaks at around 120 eggs/female, which is in the range given by Reinhardt et al. (2007). Lifetime displacement is usually below 1 km, but can reach up to 8 km, which agrees with the results of capture-mark-recapture studies (e.g. Vandewoestijne et al., 2004). In terms of movement, unmanaged and extensively managed grassland are the primary habitats 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 replicates the Germany-wide trends to a certain extent (Fig. 10). As with the monitoring data, the model data too show an initial period of population decline, followed by stabilisation and (partly) increase. The effect of the 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. However, the regionally-simulated populations do not follow the national monitoring data in detail: the year-to-year fluctuations are less pronounced in the model, and the degree

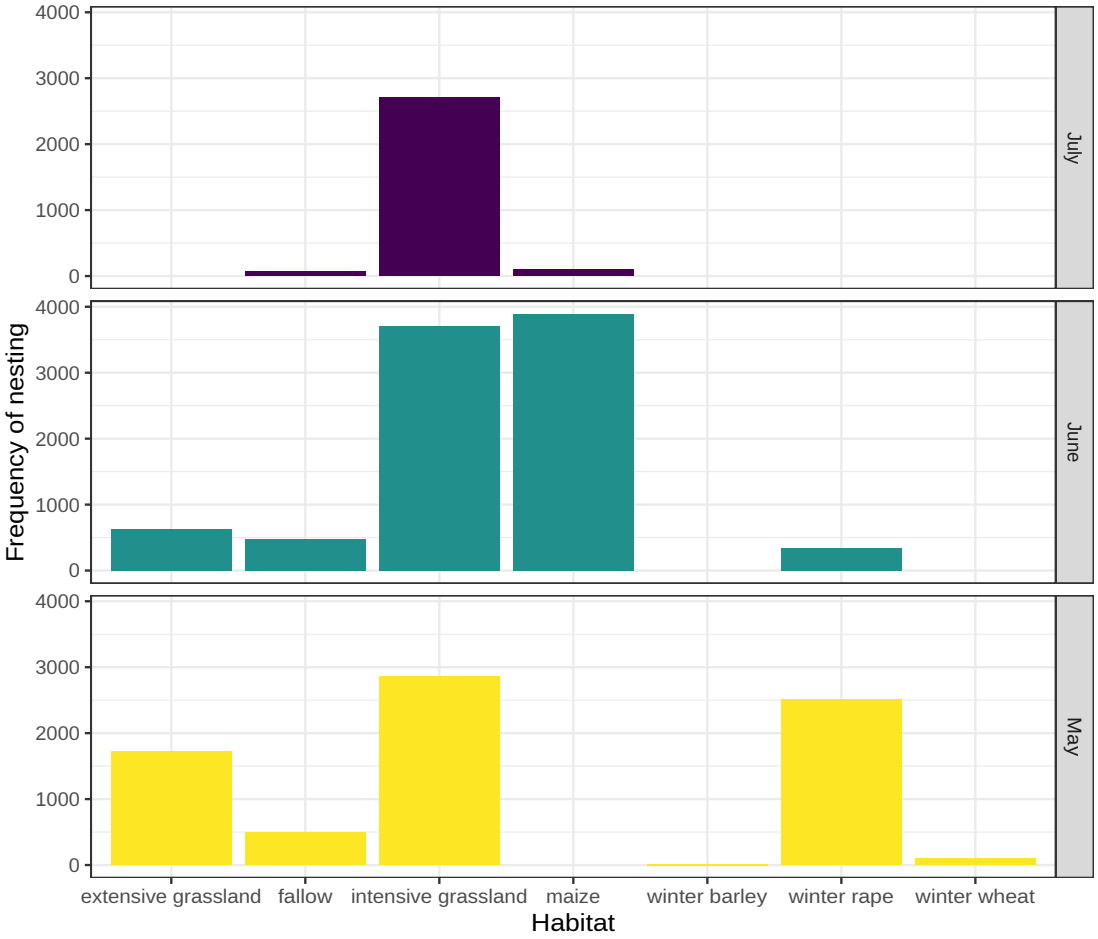

Figure 7: Habitat usage by nesting skylarks over the summer months. Data from a 10-year simulation run (2011-2020) in Jena under the intensive grassland / winter grain scenario (cf. Fig. 8).

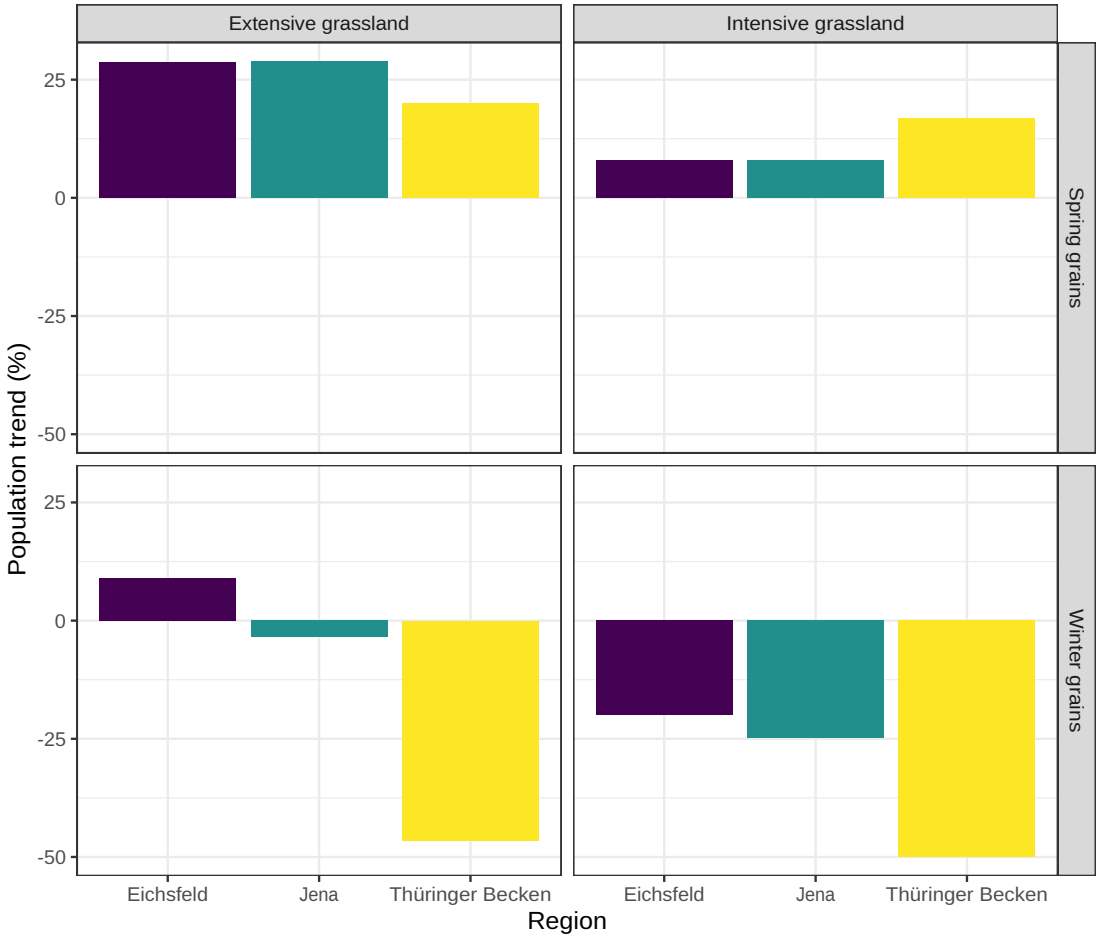

Figure 8: Skylark population trends in four different land-use scenarios in the three model regions. Trends are given as percentage increase/decrease after 10 simulated years (2011-2020).

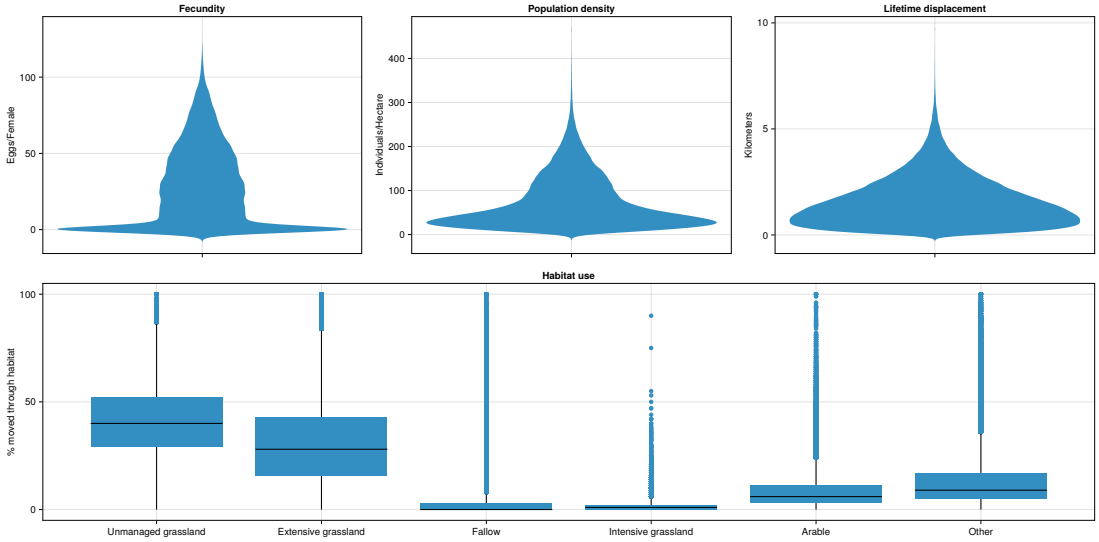

Figure 9: Pattern-testing for the marbled white model, showing several lifetime variables. Top left: Number of eggs laid by each female. Top center: Population density experienced by each individual (i.e. number of conspecifics in the surrounding hectare). Top right: Distance of the location at death from the location at birth for each individual. Bottom: Proportion of movement steps taken in different habitat types.

of stabilisation or recovery after 2015 diverges quite widely. Indeed, differences in the weather in the three regions (Eichsfeld is coolest, Jena warmest) lead to quite different population trajectories.

4 Discussion

4.1 Model purpose and design principles

Persefone.jl's primary purpose is to evaluate the impact of agricultural practice on wildlife population dynamics in Germany. To achieve this, it simulates the spatiotemporal dynamics of agricultural landscapes together with the behaviour and life cycle of target animal species. The intention is to use the combined model for *ex ante* and *ex post* assessments of changes in agricultural policy, but also to make it available as a software platform for agroecological research. To make these use cases feasible, also for other researchers, we have put great effort into making Persefone.jl reliable and extendable.

Reliability refers to both the scientific and technical trustworthiness of the model software (Vedder et al., 2021). We seek to achieve scientific credibility through the avoidance of

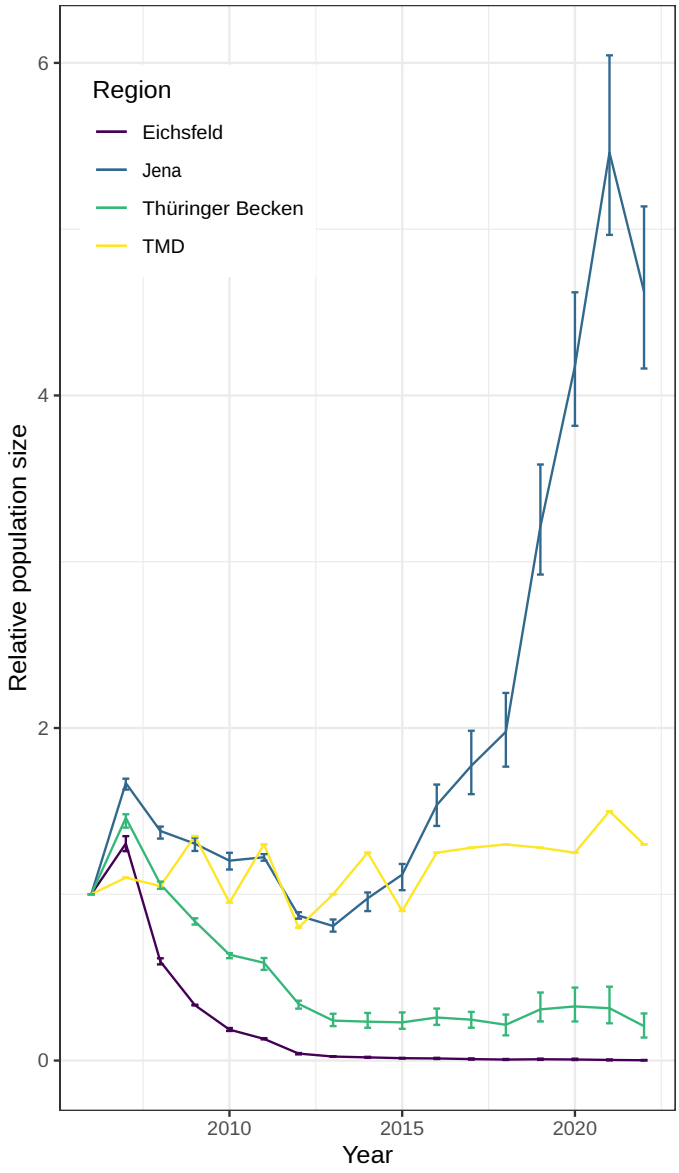

Figure 10: Marbled white population development in the three model regions from 2006 to 2022, compared with the national trend collected by the German butterfly monitoring scheme (TMD). Each line shows mean population size of five replicate simulation runs, relative to the population size in 2006; error bars show maximum/minimum values. TMD data from Kühn et al. (2024).

unnecessary complexity, the use of empirical data and knowledge wherever possible, and the continuous application of pattern-oriented modelling (cf. Grimm et al., 2014). To maintain technical reliability, we provide extensive documentation along with the open source code, and conduct code reviews and apply unit and integration testing and to catch mistakes (Balaban et al., 2021; Ropella et al., 2002).

Extensibility is part of the design of Persefone.jl in three directions. First, adding new regions to be simulated requires no code changes, merely the provision of the required map and weather input files. For Germany, all required data are publicly available, and the process required to import them into Persefone.jl is fully documented and partly automated. Second, adding new animal species can be done without changes to the model core, by adding a new species definition file that makes use of the existing interface functions in the animal component. To make these species definition files as clear and succinct as possible, we have used Julia's macro system to create a custom domain-specific language for this purpose, documented in the manual (Holst & Belete, 2015). Third, both the farm and the crop component are designed to be replaceable by equivalent modules. As the communication between the model components takes place using defined function interfaces, new component implementations can be slotted in to replace or augment the current ones, as demonstrated by the two different crop models in use at the moment.

4.2 Model evaluation

Persefone.jl is a complex model with many interacting parts. Still, as the results above show, these parts both individually and together can emulate many dynamics of agricultural landscapes and ecosystems. The landscapes simulated by the model are based on high-resolution maps of real regions, whose temporal dynamics are captured using simulated farm management based on typical crop rotations. We use an established crop-growth model (AquaCrop) to track how different habitat parameters (such as plant height and canopy cover) change over time, and calibrated it to our study regions for maximum accuracy. The remaining uncertainty in the output is in line with what is to be expected from comparable models (Kostková et al., 2021).

Pattern-oriented modelling shows that the animal models capture the relevant ecological mechanisms well. In particular the skylark model clearly replicates three well-known patterns, each of which requires the interplay of multiple ecological processes, without having needed extensive calibration and parameter testing. The results of the simulation of different land use scenarios show that Persefone.jl can indeed be used to evaluate the impact of changes in agricultural management on species populations, and that its output

aligns with empirical observations. The marbled white model also captures most of its patterns well, and qualitatively replicates the observed population trend over the past 20 years. However, the strong quantitative divergence between the regions suggest that there are still secondary influences on the population dynamics which this species model does not yet capture.

4.3 Limitations

We are well aware that Persefone.jl has a number of important limitations; some fundamental, some practical. A fundamental and unremovable limitation concerns the uncertainty of the model output, due to the inherent complexity and stochasticity of ecosystems. While we have integrated the best available knowledge on our target species' ecology into our model, and the validation shows that the model can reproduce multiple empirical patterns, (agro-)ecological dynamics are known to be highly context sensitive (e.g. le Clech et al., 2024). Thus, our model output is not to be understood as precise predictions, but rather as an expected trend, given current knowledge.

Another fundamental limitation, which may or may not be remediable in future, has to do with the land cover map. We used the highest-resolution map that is available for all of Germany (mundialis GmbH & Co. KG, 2021), to ensure the feasibility of simulating multiple regions. However, its spatial resolution of 10 m is still too coarse for a number of linear landscape features (such as flowering strips), which are ecologically very important. In addition, it only provides six land cover classes (Fig. 2), making the differentiation between ecologically distinct types of semi-natural habitat effectively impossible (e.g. calcareous grassland vs. peat land, hedgerow vs. tree line). For our current purposes, we consider the map to be good enough and the best available, but we are looking into possible ways to address these shortcomings in future releases.

A current limitation that will be improved in future is the minimal implementation of the farm model. At the moment, the only farm management actions we simulate are sowing, harvest, and mowing, while crop rotations are fixed. This will be expanded significantly in a follow-up research project, to include not only more management actions but also dynamic, agent-based farmer decision making.

4.4 Contribution to the current research context

A recent review showed that many existing individual-based models of biodiversity in agricultural landscapes are very conceptual in their approach and often don't consider

the effects of agricultural management (Vedder et al., *in review*). With Persefone.jl, we wanted to build an applied model that simulates real species in real landscapes, and considers both the direct and indirect impacts of agricultural practice.

The most similar model available today 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 design of Persefone.jl and ALMaSS are quite similar (and we make use of their vegetation submodel), we understand our model to be a complement to ALMaSS in three ways. First, it is important for a research community to have multiple models studying the same question, as this leads to more robust understanding and predictions (Hooftman et al., 2022; Rosenzweig et al., 2013). Second, our modelling approaches differ: while ALMaSS embraces complexity and consistently chooses the highest-realism implementation option possible, Persefone.jl pursues a policy of minimum-necessary complexity, leaving out any details that are not significantly important to the modelling purpose. And third, 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.

Beyond the immediate modelling context, Persefone.jl contributes to agroecological research by drawing attention to the importance of temporal dynamics. While spatial composition and configuration have long been the focus of landscape ecology, temporal patterns and effects have received much less attention (Marrec et al., 2022). This is particularly critical in agroecosystems, where farm management creates regular disturbances and resource fluctuations in constantly changing landscapes (Vasseur et al., 2013). These dynamics are not visible on the annual land cover maps used for many ecological studies, and capturing them empirically is not trivial (Katna et al., 2023). Thanks to its high temporal and spatial resolution, Persefone.jl is well-placed as a tool to study the effects of these spatiotemporal dynamics on a range of animal species. In addition, its farm and crop components could be used in joint empirical-modelling studies as a way of interpolating the state of a landscape over the course of a year.

4.5 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 therefore make Persefone.jl available as a tool for policy

evaluation and a platform for agroecological research.

Acknowledgements

The authors warmly thank the other members of the CAP4GI consortium for their excellent collaboration throughout the project. DV, MM, and GDI are funded 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. Responsibility for the content of this publication lies with the authors. DV, MM, and GP gratefully acknowledge the support of iDiv, funded by the German Research Foundation (DFG–FZT 118, 202548816).

Data availability

The Persefone.jl source code and relevant 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

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

Balaban, G., Grytten, I., Rand, K. D., Scheffer, L., & Sandve, G. K. (2021). Ten simple rules for quick and dirty scientific programming. *PLOS Computational Biology*, *17*(3), e1008549. https://doi.org/10.1371/journal.pcbi.1008549

Batáry, P., Dicks, L. V., Kleijn, D., & Sutherland, W. J. (2015). The role of agri-environment schemes in conservation and environmental management. *Conservation Biology*, *29*(4), 1006–1016. https://doi.org/10.1111/cobi.12536

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

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

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., Augusiak, J., Focks, A., Frank, B. M., Gabsi, F., Johnston, A. S., Liu, C., Martin, B. T., Meli, M., Radchuk, V., Thorbek, P., & Railsback, S. F. (2014). Towards better modelling and decision support: Documenting model development, testing, and analysis using TRACE. *Ecological Modelling*, *280*, 129–139. https://doi.org/10.1016/j.ecolmodel.2014.01.018

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., & 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

Guillem, E., Murray-Rust, D., Robinson, D., Barnes, A., & Rounsevell, M. (2015). Modelling farmer decision-making to anticipate tradeoffs between provisioning eco-

system services and biodiversity. *Agricultural Systems*, *137*, 12–23. https://doi.org/10.1016/j.agsy.2015.03.006

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

Hölting, L., Busse, M., Bülow, S., Engler, J. O., Hagemann, N., Joormann, I., Kernecker, M. L., Larondelle, N., Sturm, A., Turkelboom, F., Wätzold, F., & Cord, A. F. (2022). Co-design: Working with farmers in Europe to halt the loss of biological diversity. *Ecological Solutions and Evidence*, *3*(3). https://doi.org/10.1002/2688-8319.12169

Hooftman, D. A., Bullock, J. M., Jones, L., Eigenbrod, F., Barredo, J. I., Forrest, M., Kindermann, G., Thomas, A., & Willcock, S. (2022). Reducing uncertainty in ecosystem service modelling through weighted ensembles. *Ecosystem Services*, *53*, 101398. https://doi.org/10.1016/j.ecoser.2021.101398

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

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

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

Pe'er, G., Dicks, L. V., Visconti, P., Arlettaz, R., Báldi, A., Benton, T. G., Collins, S., Dieterich, M., Gregory, R. D., Hartig, F., Henle, K., Hobson, P. R., Kleijn, D., Neumann, R. K., Robijns, T., Schmidt, J., Shwartz, A., Sutherland, W. J., Turbé, A., ... Scott, A. V. (2014). EU agricultural reform fails on biodiversity. *Science*, *344*(6188), 1090–1092. https://doi.org/10.1126/science.1253425

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

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

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 prop-

erties 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* (B. Klausnitzer & R. Reinhardt, **typeredactors**). 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

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

Rosenzweig, C., Jones, J. W., Hatfield, J. L., Ruane, A. C., Boote, K. J., Thorburn, P., Antle, J. M., Nelson, G. C., Porter, C., Janssen, S., Asseng, S., Basso, B., Ewert, F., Wallach, D., Baigorria, G., & Winter, J. M. (2013). The Agricultural Model Intercomparison and Improvement Project (AgMIP): Protocols and pilot studies. *Agricultural and Forest Meteorology*, *170*, 166–182. https://doi.org/10.1016/j.agrformet.2012.09.011

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

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

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., & 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

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. A. M., Dennis, E. B., Schmucki, R., Sevilleja, C., Balalaikins, M., Botham, M., Bourn, N., Brereton, T., Cancela, J., Carlisle, B., Chambers, P., Collins, S., Dopagne, C., Escobés, R., Feldmann, R., Fernández-García, J., Fontaine, B., Gracianteparaluceta, A., Harrower, C., … Roy, D. B. (2019). *The EU Butterfly Indicator for Grassland species: 1990-2017* (Technical Report). Butterfly Conservation Europe. www.butterfly-monitoring.net

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

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

Persefone.jl User Manual

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

http://persefone-model.eu

August 29, 2025

v0.8.0

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 available on Gitlab.

*This documentation was last updated on 2025-08-29 for **Persefone.jl v0.8.0** (commit 33b037c).*

Part II

User guide

Chapter 1

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.

1.1 Installation

For more detailed installation instructions, see here.

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

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

1.2 Running from the command line

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

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

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

The full list of commandline arguments is:

```
usage: run.jl [-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 release   # create a release
```

1.3 Running from within Julia

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

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

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

Chapter 2

Graphical User Interface

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

2.1 Quick start

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

1. Download the Julia programming language and install it on your computer.
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()
```

2.2 Running from the repo

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

To install: Install Julia and download/clone the repository. Open a Julia REPL in the downloaded folder and execute the following to install all dependencies:

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

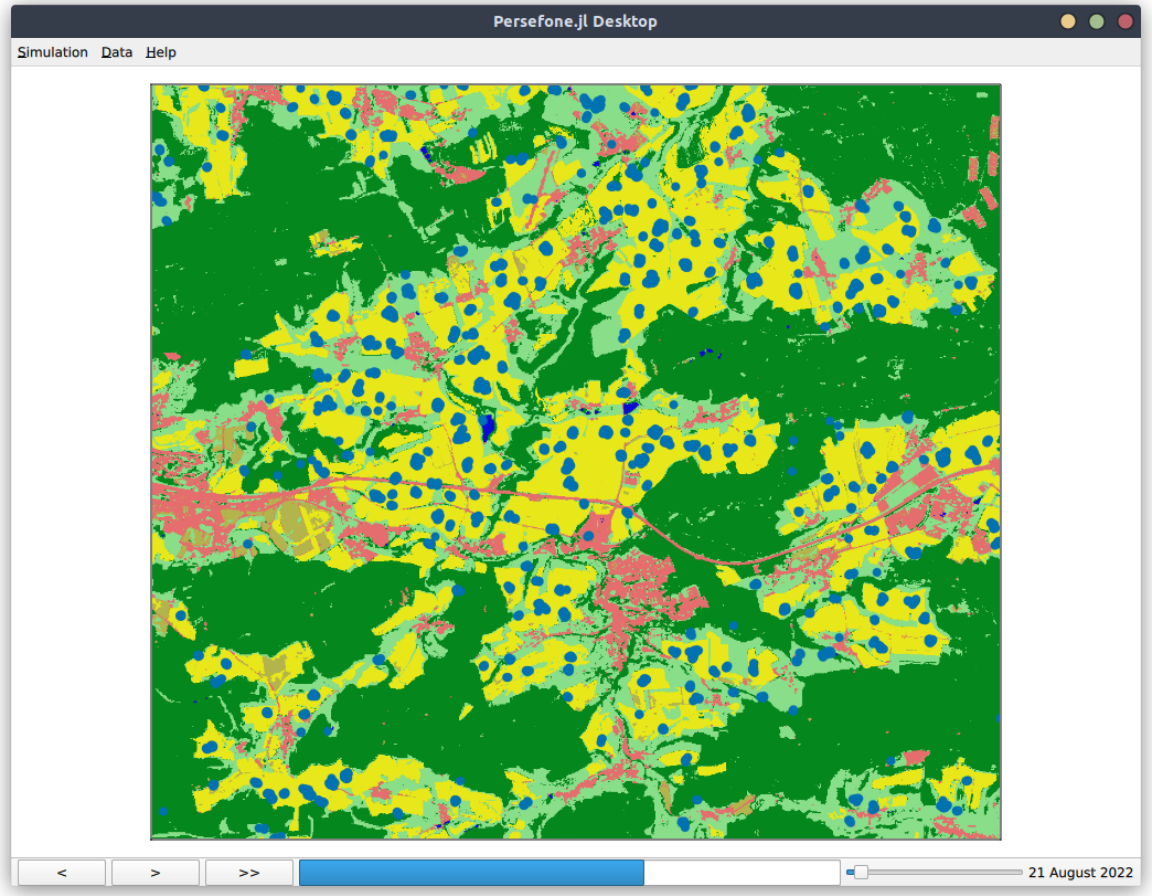

Figure 2.1: Persefone.jl Desktop screenshot

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

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

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

2.3 User interface

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

Control bar

- **Back button:** Rewind the simulation by one day.
- **Step button:** Advance the simulation by one day.
- **Run button:** Run the simulation until the button is pressed again or the end date is reached.
- **Progress bar:** Shows the percentage of time elapsed between the start and end dates of the simulation.
- **Speed slider:** Set the time delay between each simulation step when running.
- **Date:** Shows the simulation date currently displayed on the map.

Menu bar

Simulation:

- **New simulation:** Reset the model and start over.
- **Configure simulation:** Change the model settings (*not yet implemented*).
- **Load saved state:** Load a model object file saved by a previous simulation run.
- **Save current state:** Save a model object file for later use.
- **Quit:** Close the application.

Data:

- **Show population graph:** Show a window with a graph of population sizes over time in the current model run.
- **Save simulation output:** Save the model output data to file (saves both raw CSV data and generated graphics).

Help:

- **Documentation:** Open the Persefone.jl online documentation in a browser.
- **Website:** Open the main Persefone.jl website in a browser.
- **About:** Show a window with core information about the application.

Chapter 3

Configuration

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

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

[core]
configfile = "src/parameters.toml" # location of the configuration file
outdir = "results" # location and name of the output folder
overwrite = "ask" # overwrite the output directory? (true/false/"ask")
logoutput = "both" # log output to screen/file/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 = 2020-01-01 # first day of the simulation
enddate = 2022-12-31 # 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
mapresolution = 10 # map resolution in meters
landcovermap = "landcover.tif" # name of the landcover map in the map directory
farmfieldsmap = "fields.tif" # name of the field geometry map in the map directory
soiltypesmap = "soil.tif" # name of the soil type map in the map directory
weatherfile = "weather.csv" # name of the weather data file in the map directory
fixlandcover = true # correct misclassified landcover pixels

[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", "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

[nature]
targetspecies = ["MarbledWhite", "Skylark"] # 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

# MarbledWhite parameters - see src/nature/species/marbled_white.jl for details
marbledwhite_initialdensity = 1 # individuals/hectare at initialisation
marbledwhite_mintemp = 16
marbledwhite_maxtemp = 32
marbledwhite_rainactive = false
marbledwhite_movement = "random"
marbledwhite_maxstepsperday = 100
marbledwhite_selfavoidance = 0.9
marbledwhite_habitatpreference = 0.95
marbledwhite_maxindperpixel = inf
marbledwhite_maxeggsperday = 5
marbledwhite_oviposition = "linear"
marbledwhite_juvenilemortality = 0.96
marbledwhite_mowingmortality = 0.0
marbledwhite_recordeggstats = false # record habitat of eggs laid rather than movement?

# Skylark parameters - see src/nature/species/skylark.jl for details
skylark_initialdensity = 3 # ha/individual at initialisation
skylark_minimumterritory = 5000
skylark_limitterritory = false
skylark_movementrange = 500
skylark_mindistancetoedge = 60
skylark_maxforageheight = 50
skylark_maxforagecover = 70
skylark_minnestingheight = 15
skylark_maxnestingheight = 45
skylark_minnestingcover = 20
skylark_maxnestingcover = 100
skylark_offfieldnesting = true
skylark_firstyearmortality = 0.38
skylark_migrationmortality = 0.33
skylark_matefaithfulness = 0.5

[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
use_region_specific_params = true # use calibrated crop parameters, or the defaults?
```

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.

Part III

Developer guide

Chapter 4

Developing Persefone

4.1 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`.

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

7. Repeat :-)

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

4.3 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 5

Source code architecture

Chapter 6

Model components

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

1. `core` and `world`: These two directories provide the foundation of the model software, which sets up and executes simulation runs. It also reads all input files (the configuration file, landscape maps, and weather data), and provides data output functionality.
2. `nature`: This is 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.
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 `FarmPlot`, representing one field and its associated extent and crop type. Currently two different models are used here: AquaCrop for the most important crops, and ALMaSS for the rest.

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

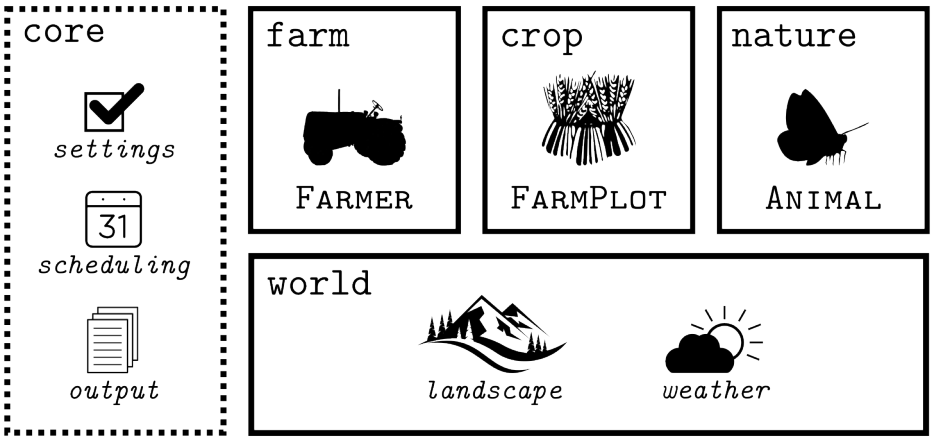

Figure 6.1: "model architecture"

Chapter 7

Important implementation details

The `model` object

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

Model configuration/the `@param` macro

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

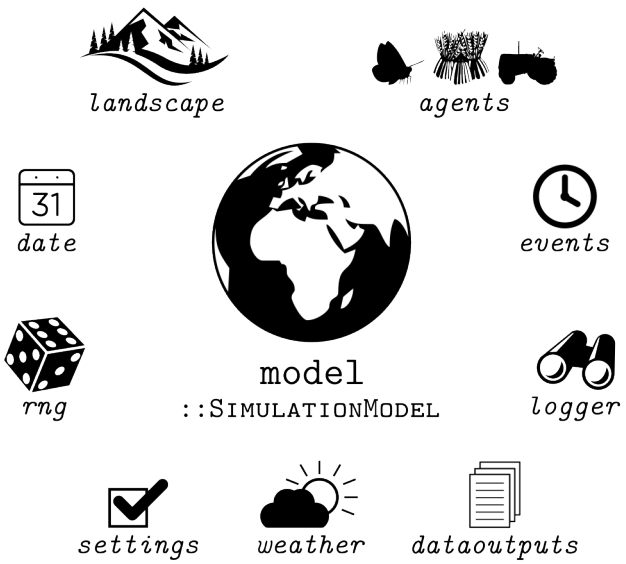

Figure 7.1: "the model object"

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

@param and other macros

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

Output data

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

Farm events

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

Random numbers and logging

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

Using the model RNG

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

Chapter 8

Adapting Persefone

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

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

Changing the parameters

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

Changing the region

To apply Persefone to a new region, you need to create new input maps of land cover, field geometries, and soil type, and download the local weather data. How to do so is described here.

Adding new animal species

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

Adding new crop species

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

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

Linking to another model

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

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

Chapter 9

Maps and weather data

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

There is a QGIS project file at `data/regions/auxiliary/persefone.qgz`, which can be used get an overview of the existing region input files and add new ones. All region data files are stored using the following convention:

```
data/regions/<regionname>/
-> <regionname>.geojson
-> landcover.tif
-> fields.tif
-> soil.tif
-> weather.csv
```

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

9.1 Land cover maps

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

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

To create a Persefone map input file, you need to crop the national Mundialis map to the extent that you want to simulate (suggestion: edge lengths between 10-20 km are a reasonable size).

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

9.2 Field ID maps

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

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

The first thing to do is to make sure that the vector layer has a numeric (!) field with a unique identifier for each field block (check the attribute table). The Thüringen data has the FBI ("Feldblockident") field, but this is a string value and therefore not usable by the rasteriser. So, we set the vector layer to edit mode, open the field calculator, enter the information for a new field (call it "FID" and set it to a 32-bit integer), and enter `@row_number` in the expression field. Then save the layer and close the calculator.

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

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

9.3 Soil data

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

To create the Persefone input file, you first need to rasterise the data. See the instructions above - choose `BODENART` as the field for the burn-in value. (Note: rastering the whole map produces a 20GB file! This can later be deleted again.) Then you need to align and crop it to the extent you require, using the dialog at `Raster -> Align Rasters...`. Select your landcover map as the reference layer and the extent layer, then choose your national soil map as the input. (Don't forget to define the output file name using `Configure Raster...`, this is a bit hidden.) The created output file can then be used for the `soilmap` parameter. Its integer values map onto the `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
```

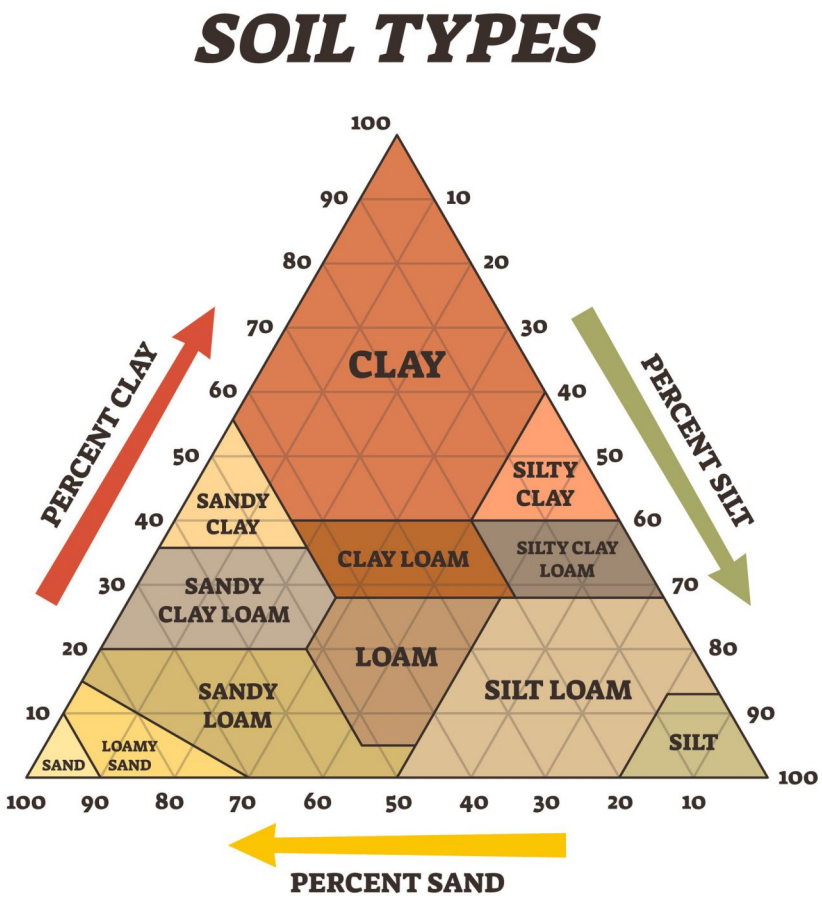

Figure 9.1: Soil types triangle

```
12: Tonlehme (tl) -> clay_loam
13: Tonschluffe (tu) -> silty_clay_loam
14: Watt -> nosoil
```

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

9.4 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 Nördlicher Oberrhein:** station number 05275 ("Waghäusel-Kirrlach")

The script `data/regions/auxiliary/extract_weather_data.R` 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.

Chapter 10

Defining new species

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

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

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

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

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

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

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

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

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

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

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

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

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

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

!!! tip Read the source The simplest way to understand how this DSL works is probably to read the source code of the existing animal models, found in the `src/nature/species` folder. If you have questions, ask the Persefone developers for help.

Chapter 11

Changelog

All notable changes to this project will be documented in this file.

The format is based on Keep a Changelog, and this project adheres to Semantic Versioning.

11.1 [1.0.0] - in planning

Aim: 4 species, 2 crop growth models, farm model, GAEC scenarios, experimental analysis

11.2 [0.8.0] - 29-8-2025

This minor release adds the first butterfly model (marbled white), and expands the farm model.

Added

- Marbled white (*Melanargia galathea*) species model with associated data outputs
- implemented management of extensive and intensive grassland in the farm model (new parameters: `farm.extensivegrassland`, `farm.mowingthreshold`, `farm.mowingperiod`)
- management scenarios can now be implemented using functions in the farm model and activated using the `farm.scenarios` parameter
- added `cropgroup()` functions
- added macros for weather functions (`@humidity()`, `@maxtemp()`, etc.)
- added `runparallel.jl` and `slurm.sh` to allow parallel processing of parameter scans on an HPC
- region-specific modified crop parameters for use with AquaCrop will now be loaded automatically from the `data/crops/aquacrop/regions/` directory, e.g. the file `data/crops/aquacrop/regions/jena/winter_wheat.toml` can be used to overload the parameters for "winter wheat" in the "jena" region. Use of the modified crop parameters can be controlled with the `crop.use_region_specific_params` configuration file parameter.

Changed

- Skylark model parameters can be set via configuration file
- added Skylark parameters `limitterritory` and `offfieldnesting`

- Setasides (fallows) are now reassigned to new fields every year
- ALMaSS grassland data expanded to allow continued growth after mowing
- crop rotation on all fields can now be set via parameter `farm.croprotation`
- the model now attempts to fix misclassifications in the land cover map if the `world.fixlandcover` parameter is `true`
- DataOutputs can now be passed a string with an AnnualDate instead of a frequency to specify on which date they should be run
- the `world.mapdirectory` parameter has been split up into `world.mapdirectory` and `world.region`
- updated and expanded documentation

Deprecated

Removed

Fixed

- fixed post-mowing grass growth in ALMaSS
- fixed `isvalidstart()` bug in `almass.jl`
- fixed a bug in `initpopulations!()`
- interpolated missing weather data
- crop cover is now given as percent by both crop models

11.3 [0.7.1] - 17-6-2025

Added

- AquaCrop can now estimate crop height from dry biomass using the rational function regression `y = a * (x/x0)^b / (c + (x/x0)^b)` with parameters fitted for maize, winter wheat, winter barley, and winter rapeseed. AquaCrop by itself does not model plant height.

11.4 [0.7.0] - 14-03-2025

This minor release adds support for the AquaCrop crop model.

Added

- Add AquaCrop crop model
- Simple linear crop height estimation for AquaCrop plants from biomass (AquaCrop does not model plant height)
- Read soil type map, controlled with the setting "world.soiltypesmap"
- Landscape `Pixels` store their soil type (`enum SoilType`)
- `FarmPlots` store the most common soil type of their landscape `Pixels` (AquaCrop needs the soil type as an input parameter).

Changed

- Allow multiple crop models to be used in one simulation. The settings "crop.cropmodel" and "crop.cropdirectory" are now comma-separated lists of crop models and their data directories.

Deprecated

Removed

Fixed

11.5 [0.6.1] - 14-03-2025

Added

- user manual: documentation is now compiled to PDF (#91)
- added soil maps to region data, but not used yet
- added `mean_cloud_cover` and `potential_evapotranspiration` fields to weather data csv files

Changed

- added soil map section to GIS docs
- Changed weather internal representation to struct-of-arrays (previously a dict-of-struct representation). The `Weather` type now stores all weather information for the whole simulation, with a function interface, e.g. `sunshine(weather, date)`.
- When reading weather data, we now throw an error when there are missing days or any missing values for the fields `min_temperature`, `max_temperature`, `mean_temperature`, `precipitation`, and `potential_evapotranspiration` In the future missing values could also be imputed.
- The script for weather data extraction at `data/regions/auxiliary/extract_weather_data.R` has been reworked to always return an output row for each day in the date range, even if the day is missing in the original data source. It now also accepts the stations to download as command-line arguments. The `renv` lockfile for the R environment used to run the script can be found at `data/regions/auxiliary/renv.lock`.

Deprecated

Removed

Fixed

11.6 [0.6.0] - 13-01-2025

This minor release re-implements the ALMaSS crop model

Added

- `crop.cropdirectory` parameter specifies folder in which all crop data files for the selected crop model can be found.

Changed

- `preprocessparameters()` checks whether the map and crop directories are reachable from the current working directory. If not, it checks whether it can be reached from the package directory. This makes running simulations easier when Persefone has been installed as a package.
- `simulate()` and `initialise()` now take a `params` keyword argument that can be used to override parameters from other input sources
- The ALMaSS crop data config file `data/crops/almass/crop_data_general.csv` now has extra columns for `is_c4_plant`, `sowingdensity`, and `phase_before_harvest`

Deprecated

Removed

- `crop.cropfile` and `crop.growthfile` parameters -> user configuration is now done via `crop.cropdirectory`, names of ALMaSS input files are specified as constants in `almass.jl`

Fixed

- The implementation of the ALMaSS vegetation model in Persefone has been completely rewritten, hopefully more faithfully reproducing the logic in ALMaSS. The resulting plant heights are now more realistic and do not produce the extreme plant heights seen previously (which was due to an erroneous interpretation of the ALMaSS growth curves).

11.7 [0.5.5] - 09-08-2024

This point release implements the first basic farm model

Added

- basic farm model that assigns a crop rotation to each field, sowing and harvesting when appropriate
- new parameters: `farm.farmmodel`, `farm.setaside`, `farm.fieldoutfreq`
- visualisation of cropped area and crop growth over time
- `farm.setaside` setting to configure what proportion of land farmers let lie fallow
- `isharvestable()` function for `FarmPlots`
- `@areaof` macro to calculate the area of a given number of landscape pixels
- `data/farm/standard_gross_margins.csv` from KTBL data

Changed

- expanded & adapted general crop data and crop growth curve tables

Fixed

- bug fixes in the ALMaSS crop model

11.8 [0.5.4] - 08-08-2024

Skylark data analysis and new internal utility functions

Added

- `AnnualDate` type and associated functions for working with recurring dates (#101)
  - can be constructed from two `Int64`, a `Date`, or a `Tuple{Int64,Int64}`
  - automatic conversion from `Date` or `Tuple{Int64,Int64}`
  - can use operators: `==`, `<`, `+`, `-`, `:`
  - `thisyear()`, `lastyear()`, `nextyear()` functions and macros
- new file `core/utils.jl` for utility functions that fit in no other file
- irregular data logging using `record!()`/`@record()` (#103)
- data outputs & visualisation for the skylark model (#97)
- `randn()` function and macro to sample from a vector using a normal distribution
- `make install` to download and install Julia and package dependencies (on Linux, #67)
- weather file for the Thüringer Becken

Changed

- moved random number functions and macros from `input.jl` to `utils.jl`
- expanded weather data for Jena to 1990-2023
- Non-breeding skylarks only search for neighbours to follow once (-> huge performance improvement!)

Fixed

- bug fixes in the skylark model

11.9 [0.5.3] - 31-07-2024

Switchable crop models

Added

- Support for switchable crop models (#70), crop models can be set with the `cropmodel` setting in the `[crop]` section of `parameters.toml`.
- New submodules `ALMaSS` for the ALMaSS crop model, and `SimpleCrop` for testing switchable crop models.

Changed

- All functionality specific to the ALMaSS crop model has been moved to the submodule `ALMaSS`.
- Due to switchable crop models, some types are now parametric: `AgricultureModel{Tcroptype,Tcropstate}` and `FarmPlot{Tcropstate}`.
- `FarmPlot{Tcropstate}` now only stores basic information about which pixels are part of the farm plot, all crop-specific information is now stored in the field `cropstate`. Many functions acting on a `FarmPlot` now mostly forward to functions of the same name acting on the `cropstate` field of a `FarmPlot`.
- The type of `height` in `ALMaSS.CropState` and `ALMaSS.CropCurveParams` is now a unitful number `::Length{Float64}`.
- Both crop models `ALMaSS` and `SimpleCrop` now also support the functions `cropcover` and `cropyield` in addition to `croptype`, `cropname`, `cropheight`.
- `cropheight` now returns a unitful number `::Length{Float64}`, and returns height `0cm` if the landscape at that position is not a `FarmPlot`.

11.10 [0.5.2] - 30-07-2024

Rewrote the skylark model

Added

- Skylark model is largely rewritten to follow a new phase structure (#9)
- animals can occupy territories (see `@occupy`, `@isoccupied`, `@vacate` macros/functions) (#94)
- `@cropcover` macro and function
- ODD documentation for Skylark

Changed

- input files that are now copied to a separate `inputs` directory within the output directory
- `EventType` renamed to `Management` for clarity
- documentation website now has a "Scientific Documentation" section

Removed

- old skylark model (has been rewritten, see above)

Fixed

- all skylarks now migrate (#90)
- `insectbiomass()` uses units

11.11 [0.5.1] - 13-06-2024

Added Unitful.jl

Added

- Unitful.jl now used to add units to quantities
- `world.mapdirectory` parameter specifies the path to the directory in which

`landcovermap`, `farmfieldsmap`, and `weatherfile` are located

- `world.mapresolution` parameter specifies the input maps' spatial resolution in meters

Changed

- spatial functions now work with explicit distances (using Unitful.jl) rather than using the number of pixels
- all species definitions and tests updated to use units

11.12 [0.5.0] - 07-06-2024

This release doesn't add much new functionality, but represents a major restructuring of the code base. Specifically, it removes the Agents.jl dependency and changes the way the species definition macros work and are used.

Added

- `SimulationModel` type, extended by `AgricultureModel` struct
- `@create` macro defines a special phase function that is called when an individual animal is created (at birth or on model initialisation)
- functions (and associated macros) to replace Agents.jl functionality:
  - `move!()` and `walk!()`
  - `nearby_ids()`, `nearby_animals()`, `countanimals()`, `neighbours()`
  - `directionto()`, `distanceto()`, `randomdirection()`
  - `nagents()`, `killallanimals!()`
- `@here` macro to return the pixel currently occupied by the active animal
- `core.logoutput` parameter to define whether logs are printed to screen, file, none, or both
- large logo and model structure diagram
- Changelog

Changed

- `SimulationModel` replaces `AgentBasedModel`
- Species definition macros revamped:
  - `@species` now only defines parameters and variables and creates a mutable struct
  - `@phase` must now be defined in the top-level code and creates a global function
  - `@initialise` renamed to `@populate`, must also be called in the top-level code
  - `initindividual()` renamed to `create!()`
- Skylark, Wolpertinger, and Wyvern updated to match the new macros
- requires Julia 1.10

Removed

- Agents.jl dependency (including `AgentBasedModel` and functions for adding/moving/removing agents)

11.13 [0.4.1] - 2023-11-14

Initial version of the skylark model

Added

- initial version of the Skylark species
- small Jena map
- animal individuals keep track of their parents' IDs
- several new functions and macros for animals
- installation instructions for Windows

Changed

- graphics output is more configurable

11.14 [0.4.0] - 2023-10-28

Functions for animal populations

Added

- initialisation functions for individuals (not just species)
- migration function / migrant pool for animals that disappear from the landscape during winter
- skylark migration

Changed

- online documentation was expanded and restructured
- nature macros moved to a separate file

Started changelog at this point, earlier versions are not included.

<!– Template

11.15 [version] - unreleased

<comments>

PLANNED

Added

Changed

Deprecated

Removed

Fixed

–>

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.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, farmplots).

source

`Persefone.SimulationModel` – Type.

```
SimulationModel
```

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

source

`Persefone.AnnualDate` – Type.

```
AnnualDate
```

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

source

`Base.randn` – Function.

```
randn(vector)
```

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

source

`Persefone.bounds` – Method.

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

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

source

`Persefone.cycle!` – Function.

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

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

source

`Persefone.thisyear` – Method.

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

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

source

`Persefone.@areaof` – Macro.

```
@areaof()
```

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

source

`Persefone.@chance` – Macro.

```
@chance(odds)
```

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

source

`Persefone.@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.nagents` – Method.

```
nagents(model)
```

Return the total number of agents in a model object.

source

`Persefone.paramscan` – Method.

```
paramscan(settings)
```

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

source

`Persefone.simulate!` – Method.

```
simulate!(model)
```

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

source

`Persefone.simulate` – Method.

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

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

This is the default way to run a Persefone simulation.

source

`Persefone.stepagent!` – Method.

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

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

source

`Persefone.stepsimulation!` – Method.

```
stepsimulation!(model)
```

Execute one update of the model.

source

12.3 landscape.jl

This file manages the landscape maps that underlie the model.

`Persefone.df_soiltypes_bodenatlas` – Constant.

Bodenatlas soil type id, corresponding Persefone soil type, and numbers to Persefone SoilType enum and the original Bodenatlas description of the soil type

source

`Persefone.soiltype_bodenatlas_to_persefone` – Constant.

Map a Bodenatlas soil type integer to a Persefone `SoilType` enum

source

`Persefone.FarmEvent` – Type.

```
FarmEvent
```

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

source

`Persefone.LandCover` – Type.

The land cover classes encoded in the Mundialis Sentinel data.

source

`Persefone.Management` – Type.

The types of management event that can be simulated

source

`Persefone.Pixel` – Type.

```
Pixel
```

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

source

`Persefone.SoilType` – Type.

The soil type of a Pixel or FarmPlot

source

`Persefone.createevent!` – Function.

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

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

source

`Persefone.directionto` – Method.

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

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

source

`Persefone.directionto` – Method.

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

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

source

`Persefone.distanceto` – Method.

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

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

source

Persefone.distanceto – Method.

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

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

source

Persefone.distancetoedge – Method.

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

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

source

Persefone.farmplot – Method.

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

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

source

Persefone.inbounds – Method.

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

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

source

Persefone.initlandscape – Method.

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

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

source

`Persefone.landcover` – Method.

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

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

source

`Persefone.randomdirection` – Method.

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

Get a random direction coordinate tuple within the specified distance.

source

`Persefone.randompixel` – Function.

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

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

source

`Persefone.safebounds` – Method.

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

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

source

`Persefone.updateevents!` – Method.

```
updateevents!(model)
```

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

source

12.4 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

Input and Output

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

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.@param` – Macro.

```
@param(domainparam)
```

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

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

source

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 makieplots.jl

`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.marbledwhiteabundanceplot` – Method.

```
marbledwhiteabundanceplot(model)
```

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

source

`Persefone.marbledwhitelifestats` – Method.

```
marbledwhitelifestats(model)
```

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

source

`Persefone.marbledwhitetrendsplot` – Method.

```
marbledwhitetrendsplot(model)
```

... Returns a Makie figure object.

source

`Persefone.populationtrends` – Method.

```
populationtrends(model)
```

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

source

`Persefone.skylarkpopulation` – Method.

```
skylarkpopulation(model)
```

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

source

`Persefone.skylarkstats` – Method.

```
skylarkstats(model)
```

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

source

`Persefone.visualisemap` – Function.

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

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

source

Chapter 14

Nature submodel

Unused source files

There are two source files in the `src/nature` 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 nature.jl

This file is responsible for managing the animal modules.

`Persefone.Animal` – Type.

```
Animal
```

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

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

source

`Persefone.animalid` – Method.

```
animalid(animal)
```

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

source

`Persefone.create!` – Method.

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

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

source

`Persefone.initnature!` – Method.

```
initnature!(model)
```

Initialise the model with all simulated animal populations.

source

`Persefone.killallanimals!` – Method.

```
killallanimals!(model)
```

Remove all animal individuals from the simulation.

source

`Persefone.speciesof` – Method.

```
speciesof(animal)
```

Return the species name of this animal as a string.

source

`Persefone.speciestype` – Method.

```
speciestype(name)
```

Return the Type of this species.

source

`Persefone.stepagent!` – Method.

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

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

source

`Persefone.updatenature!` – Method.

```
updatenature!(model)
```

Run processes that affect all animals.

source

14.2 macros.jl

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

`Persefone.@animal` – Macro.

```
@animal(id)
```

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

source

`Persefone.@countanimals` – Macro.

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

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

source

`Persefone.@create` – Macro.

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

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

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

source

`Persefone.@cropcover` – Macro.

```
@cropcover
```

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

source

`Persefone.@cropgroup` – Macro.

```
@cropgroup
```

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

source

`Persefone.@cropheight` – Macro.

```
@cropheight
```

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

source

`Persefone.@cropname` – Macro.

```
@cropname
```

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

source

`Persefone.@destroynest` – Macro.

```
@destroynest(reason)
```

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

source

`Persefone.@directionto` – Macro.

```
@directionto
```

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

source

`Persefone.@distanceto` – Macro.

```
@distanceto
```

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

source

`Persefone.@distancetoedge` – Macro.

```
@distancetoedge
```

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

source

`Persefone.@follow` – Macro.

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

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

source

`Persefone.@habitat` – Macro.

```
@habitat
```

Specify habitat suitability for spatial ecological processes.

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

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

Two example uses of `@habitat` might look like this:

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

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

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

source

`Persefone.@here` – Macro.

```
@here()
```

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

source

`Persefone.@isalive` – Macro.

```
@isalive(id)
```

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

source

`Persefone.@isoccupied` – Macro.

```
@isoccupied(position)
```

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

source

`Persefone.@kill` – Macro.

```
@kill
```

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

source

`Persefone.@killother` – Macro.

```
@killother
```

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

source

`Persefone.@landcover` – Macro.

```
@landcover
```

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

source

Persefone.@migrate – Macro.

```
@migrate(arrival)
```

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

source

Persefone.@move – Macro.

```
@move(position)
```

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

source

Persefone.@nearby_animals – Macro.

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

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

source

Persefone.@neighbours – Macro.

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

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

source

Persefone.@occupy – Macro.

```
@occupy(position)
```

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

source

`Persefone.@phase` – Macro.

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

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

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

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

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

source

`Persefone.@populate` – Macro.

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

Set the parameters that are used to initialise this species' population. For parameter options, see `PopInitParams`. This macro has access to the `model` object.

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

source

`Persefone.@randomdirection` – Macro.

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

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

source

`Persefone.@randompixel` – Macro.

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

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

source

`Persefone.@reproduce` – Macro.

```
@reproduce
```

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

source

`Persefone.@respond` – Macro.

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

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

source

`Persefone.@setphase` – Macro.

```
@setphase(newphase)
```

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

source

`Persefone.@species` – Macro.

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

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

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

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

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

source

`Persefone.@vacate` – Macro.

```
@vacate(position)
```

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

source

`Persefone.@vacate` – Macro.

```
@vacate()
```

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

source

`Persefone.@walk` – Macro.

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

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

source

14.3 individuals.jl

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

`Persefone.followanimal!` – Function.

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

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

source

`Persefone.kill!` – Function.

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

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

source

`Persefone.migrate!` – Method.

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

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

source

`Persefone.move!` – Method.

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

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

source

`Persefone.occupy!` – Method.

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

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

source

`Persefone.reproduce!` – Function.

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

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

source

`Persefone.vacate!` – Method.

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

Remove this position from the animal's territory.

source

`Persefone.vacate!` – Method.

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

Remove the animal's complete territory.

source

`Persefone.walk!` – Function.

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

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

source

`Persefone.walk!` – Function.

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

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

source

14.4 populations.jl

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

`Persefone.PopInitParams` – Type.

```
PopInitParams
```

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

- `initphase` determines which life phase individuals will be assigned to at model initialisation (required).
- `birthphase` determines which life phase individuals will be assigned to at birth (required).

- `habitat` is a function that determines whether a given location is suitable or not (create this using `@habitat`). By default, every cell will be occupied.
- `popsize` determines the number of individuals that will be created, dispersed over the suitable locations in the landscape. If this is zero or negative, one individual will be created in every suitable location. If it is greater than the number of suitable locations, multiple individuals will be created per location. Alternately, use `indarea`.
- `indarea`: if this is greater than zero, it determines the habitat area allocated to each individual or pair. To be precise, the chance of creating an individual (or pair of individuals) at a suitable location is 1/indarea. Use this as an alternative to `popsize`.
- `sex` determines how indidivduals are assigned a sex. If this is `:pairs` (the default), a male and a female will be created in each selected location. If `:random`, a male or a female will be created. Otherwise, `male`, `female`, or `hermaphrodite` can be chosen to force all individuals to be of the same sex.

source

`Persefone.countanimals` – Method.

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

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

source

`Persefone.directionto` – Method.

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

Calculate the direction from the given position to the animal.

source

`Persefone.distanceto` – Method.

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

Calculate the distance from the given position to the animal.

source

`Persefone.initindividuals!` – Method.

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

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

source

`Persefone.initpopulation!` – Method.

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

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

source

`Persefone.initpopulation!` – Method.

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

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

source

`Persefone.isalive` – Method.

isalive(id, model)

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

source

`Persefone.isoccupied` – Method.

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

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

source

`Persefone.nearby_animals` – Method.

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

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

source

`Persefone.nearby_ids` – Method.

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

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

source

`Persefone.neighbours` – Function.

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

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

source

`Persefone.populationparameters` – Method.

```
populationparameters(type)
```

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

source

`Persefone.territorysize` – Function.

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

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

source

14.5 ecologicaldata.jl

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

`Persefone.initecologicaldata` – Method.

```
initecologicaldata()
```

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

source

`Persefone.marbledwhiteabundance` – Method.

```
marbledwhiteabundance(model)
```

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

source

`Persefone.marbledwhitetrends` – Method.

```
marbledwhitetrends(model)
```

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

source

`Persefone.saveindividualdata` – Method.

```
saveindividualdata(model)
```

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

source

`Persefone.savepopulationdata` – Method.

```
savepopulationdata(model)
```

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

source

`Persefone.skylarkabundance` – Method.

```
skylarkabundance(model)
```

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

source

`Persefone.skylarkterritories` – Method.

```
skylarkterritories(model)
```

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

source

Chapter 15

Species models

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

15.1 Skylark

`Persefone.Skylark` – Type.

```
Skylark
```

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

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

source

`Persefone.#1371#fun` – Function.

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

source

`Persefone.allowsnesting` – Method.

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

Check whether the given position is suitable for nesting.

source

Persefone.breeding – Method.

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

source

Persefone.create! – Method.

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

source

Persefone.destroynest! – Method.

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

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

source

Persefone.findterritory – Method.

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

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

source

Persefone.foragequality – Method.

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

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

source

Persefone.matesearch – Method.

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

source

Persefone.nesting – Method.

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

source

`Persefone.nonbreeding` – Method.

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

source

`Persefone.occupation` – Method.

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

source

`Persefone.territorysearch` – Method.

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

source

15.2 Marbled White

`Persefone.MarbledWhite` – Type.

```
Marbled White
```

Melanargia galathea is a grassland specialist butterfly.

Sources: - Baguette et al. (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 - Dennis (Ed.). (1992). The ecology of butterflies in Britain. Oxford University Press. - Ebert & Rennwald (1991). Die Schmetterlinge Baden-Württembergs, Bd.2, Tagfalter: Satyridae, Libytheidae, Lycaenidae, Hesperiidae. Verlag Eugen Ulmer. - Evans et al. (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 - Gotthard et al. (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 - Hannappel & Fischer (2020). Grassland intensification strongly reduces butterfly diversity in the Westerwald mountain range, Germany. Journal of Insect Conservation, 24(2), 279–285. https://doi.org/10.1007/s10841-019-00195-1 - Kühn et al. (2024). Tagfalter-Monitoring Deutschland: Auswertung 2005-2023. Oedippus, 42, 12–45. https://www.ufz.de/export/data/6/298835*298188*Oedippus*42*klein.pdf - Lenda & Skórka (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 - Reinhardt et al. (2007): Tagfalter von Sachsen. In: Klausnitzer & Reinhardt (Hrsg.) Beiträge zur Insektenfauna Sachsens Band 6. – Entomologische Nachrichten und Berichte, Beiheft 11, 696 + 48 Seiten. Dresden. - Roy et al. (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 et al. (2007). Die Tagfalter der Pfalz—Band 2. Gesellschaft für Naturschutz und Ornithologie Rheinland-Pfalz. - Vandewoestijne et al. (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

source

`Persefone.#1521#fun` – Function.

Initialise the marbled white population with one individual on every grassland pixel.

source

`Persefone.adult` – Method.

Adult marbled whites (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 randomly scans landscape pixels within its perceptual range of 100m. If the pixel is suitable habitat, i.e. either arable or extensive grassland with plant heights within the required range (30-60cm), it moves to this pixel. In this case, the individual also lays an egg if it has not yet laid all its eggs for that day. If the pixel 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 pixel in its perceptual range.

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).

source

`Persefone.create!` – Method.

Initialise a marbled white 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.egg` – Method.

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

source

`Persefone.habitatcategory` – Method.

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

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

source

`Persefone.larva` – Method.

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

source

`Persefone.moveproximity` – Method.

```
moveproximity(self, model)
```

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 `maxindperpixel`). Returns `nothing` if no suitable spot is selected.

source

`Persefone.moverandom` – Method.

```
moverandom(self, model)
```

The butterfly randomly inspects pixels 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 `maxindperpixel`). Returns `nothing` if no suitable spot is selected.

source

`Persefone.pupa` – Method.

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

source

`Persefone.recordlifestats` – Method.

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

Save this butterfly's life stats to file.

source

`Persefone.suitablehabitat` – Method.

```
suitablehabitat(butterfly, model, pos)
```

Check whether the given position is suitable for oviposition. This means: the land cover must be either grass or fallow, and if grass, must either not be managed, or not have been fertilised and be a certain height.

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 farmplot.jl

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

`Persefone.FarmPlot` – Type.

```
FarmPlot
```

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

source

`Persefone.averagefieldsize` – Method.

```
averagefieldsize(model)
```

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

source

`Persefone.cropcover` – Method.

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

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

source

`Persefone.cropgroup` – Method.

```
cropgroup(model, position)
```

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

source

`Persefone.cropheight` – Method.

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

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

source

`Persefone.cropname` – Method.

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

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

source

`Persefone.croptype` – Method.

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

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

source

`Persefone.harvest!` – Method.

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

Harvest the crop of this farmplot.

source

`Persefone.isgrassland` – Method.

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

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

source

`Persefone.sow!` – Method.

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

Sow the specified crop on the farmplot.

source

`Persefone.stepagent!` – Method.

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

Update a farm plot by one day.

source

`Persefone.@harvest` – Macro.

```
@harvest()
```

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

source

`Persefone.@sow` – Macro.

```
@sow(cropname)
```

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

source

16.2 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)
```

Initialise the farm plots in the simulation model.

source

16.3 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. Usually part of a `FarmPlot`.

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

16.4 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/crop/aquacrop/regions/jena/winter_wheat.toml` can be used to overload the parameters of the Persefone crop "winter wheat" for the "jena" region

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 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.updatefarms!` – Method.

```
updatefarms!(model)
```

Run processes that affect all farms and fields.

source

17.2 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

17.3 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: Crop model descriptions and calibration

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)

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:

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

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 $n_{\text{plants}\,/\,\mathrm{m^2}}$ 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.

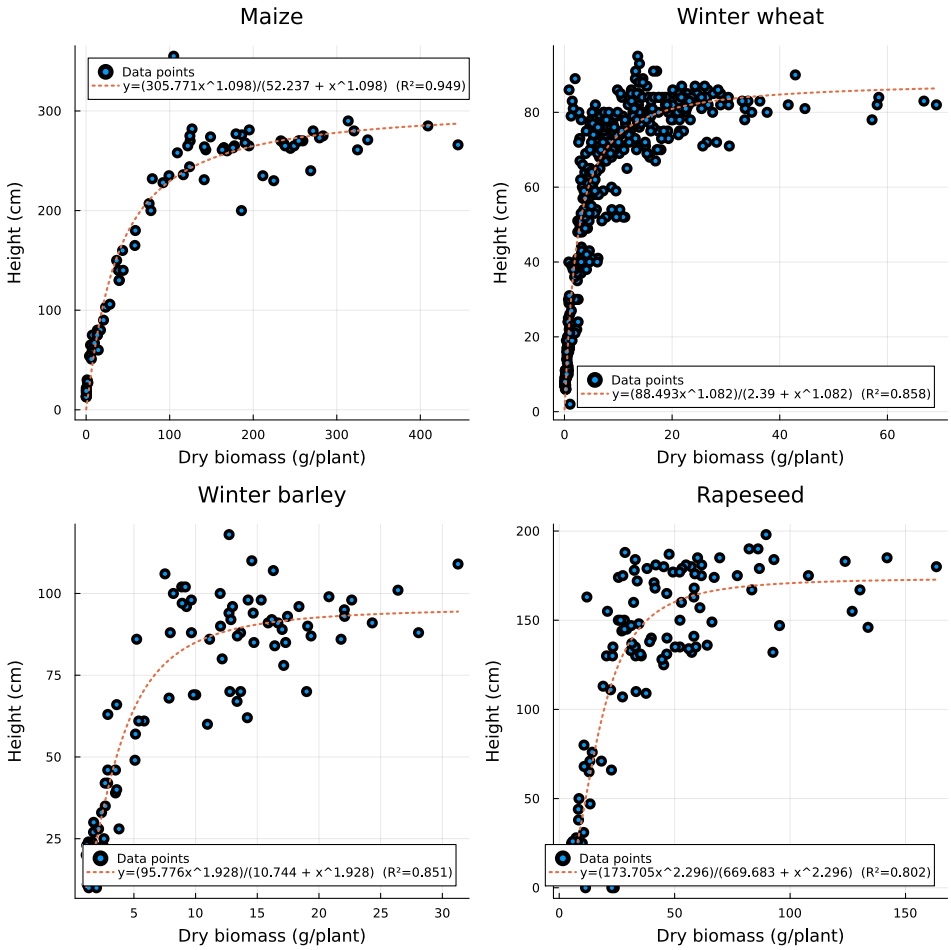

Figure 1: Data and rational function fits for predicting plant height from dry biomass

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 at `TODO:addURL`.

3 AquaCrop model calibration

Crop parameters were estimated from empirical data with `AquaCrop.jl` (Díaz Iturry et al., 2025) for silage maize, winter wheat, winter barley, and winter

rapeseed. After parameter fitting, we compared the simulated and actual results for the following variables: plant emergence, beginning of flowering, harvest date, and yield. We compared these variables at each of the three sites of Jena, Eichsfeld, and Thüringer Becken. The results of the comparison are shown in the following figures.

Silage maize.

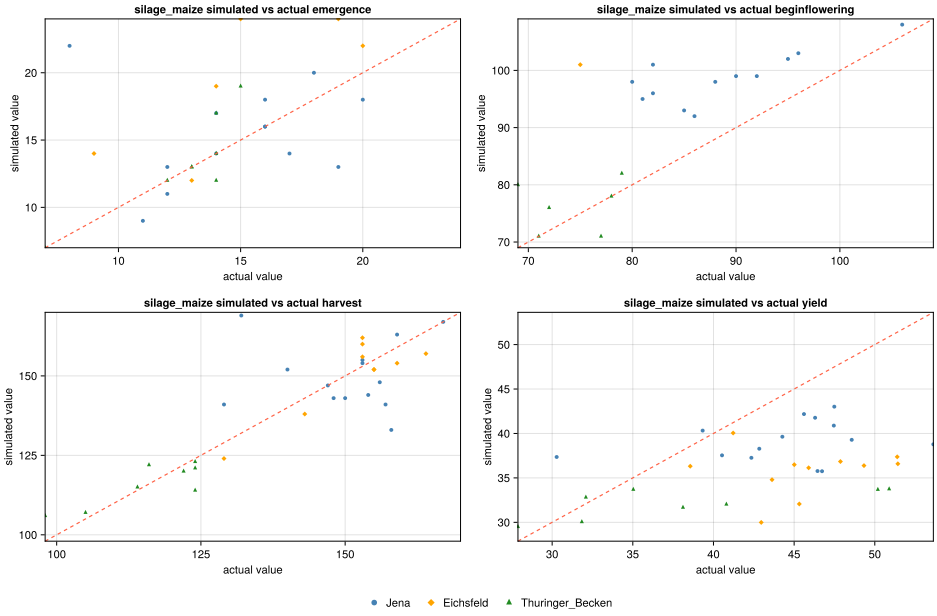

Figure 2: Silage maize: comparing simulated and actual results at three different locations.

Winter wheat.

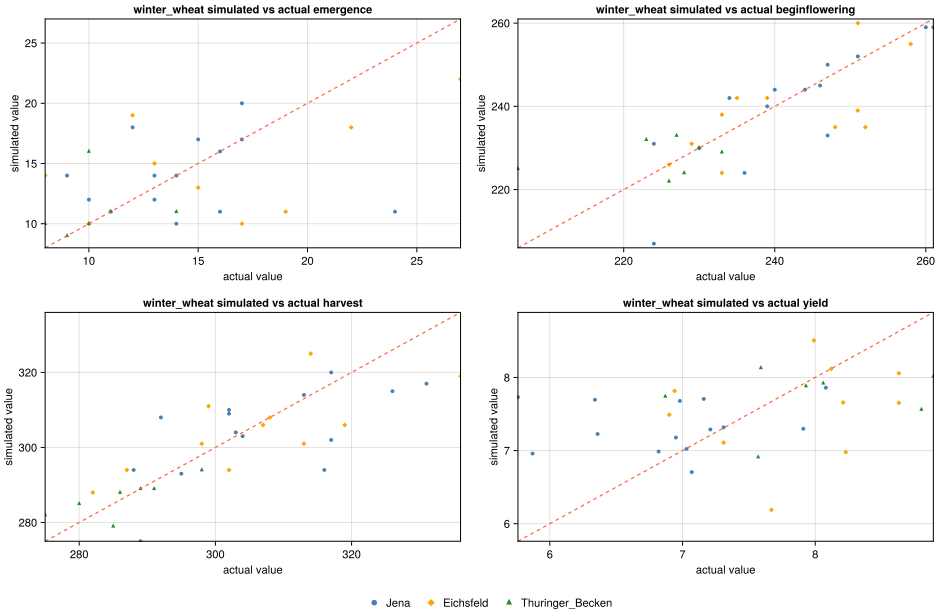

Figure 3: Winter wheat: comparing simulated and actual results at three different locations.

Winter barley.

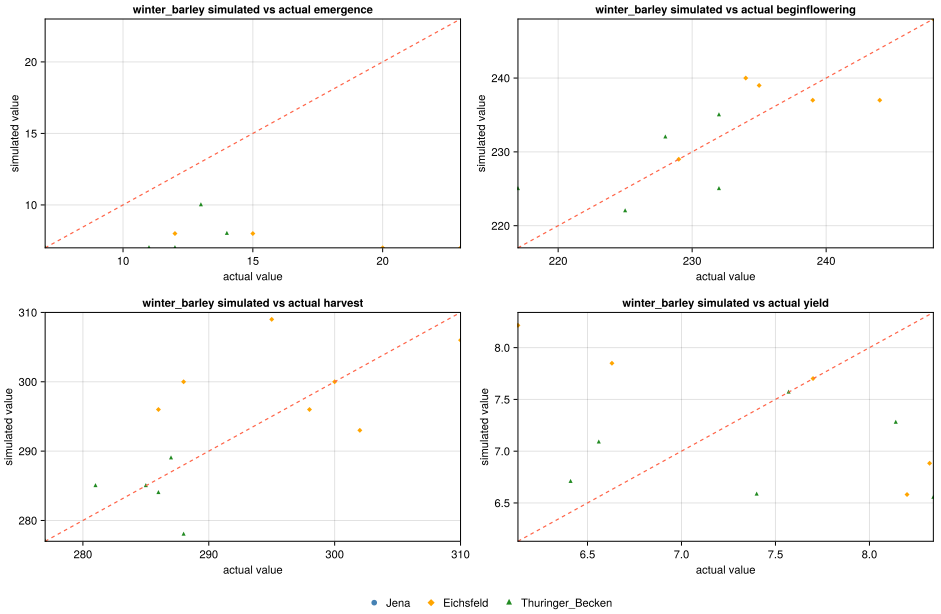

Figure 4: Winter barley: comparing simulated and actual results at three different locations.

Winter rapeseed.

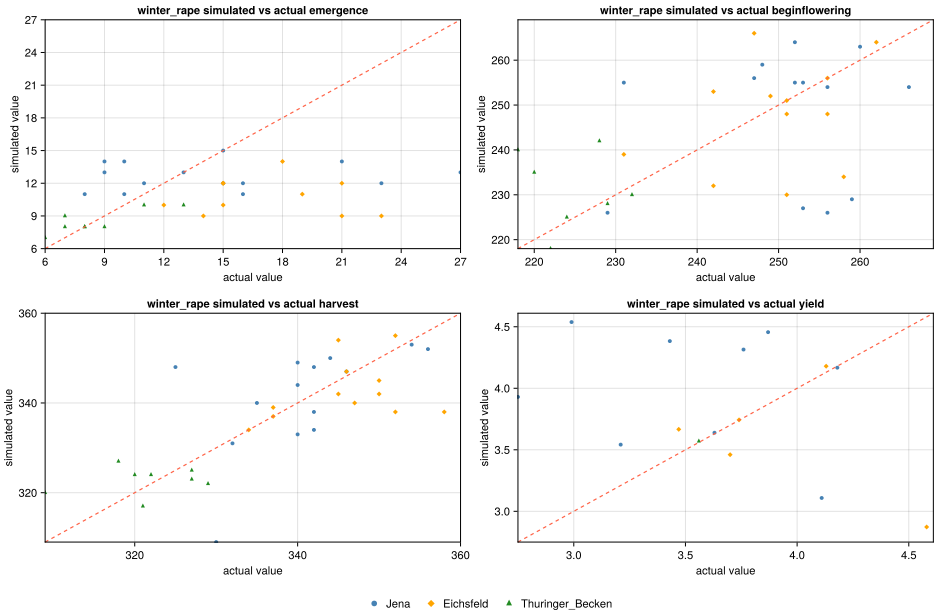

Figure 5: Winter rapeseed: comparing simulated and actual results at three different locations.

References

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.

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.

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

Vedder et al.: "Persefone.jl: Modelling Biodiversity in Dynamic Agricultural Landscapes"

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 landscapes consist of a grid of pixels with a resolution of 10m and have an extent of 270km^2-370km^2 (approximately; depending on the chosen input map). Each pixel is assigned a land cover class. It may also be associated with a farm plot, in which case it will contain information about the type and growth stage of the crop planted

here. Farm management determines which crops are grown when, and when disturbance (e.g. mowing, harvesting, tillage) takes place.

2.2. Animals

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

- `ID` A unique identifier for this individual, which can be used to link it to its parents and its offspring.
- `sex` Male or female.
- `phase` The individual's current stage in the annual 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 list of coordinates of the positions in the landscape that this individual claims as its nesting and feeding territory.
- `nest` A coordinate giving the location of the currently active nest.
- `clutch` The number of juvenile (i.e. not yet independent) skylarks that this individual is currently raising.

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

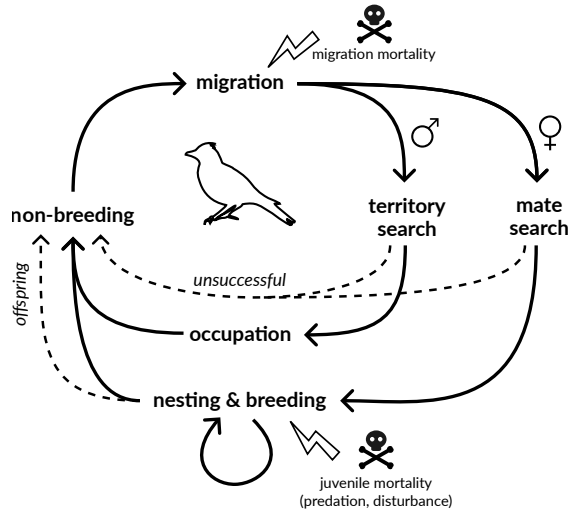

Figure 1: Phase diagram of the skylark model.

with this partner. Otherwise, they move randomly through the landscape, looking for a male with a territory and without a partner. Once the female has a partner, it changes its phase to nesting.

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

- `occupation`: The male moves at random about its territory until the breeding season is over. Then it changes its phase to nonbreeding. (Note: 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 mi-

gration 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 (see literature below). The factors and processes affecting these are therefore given the most attention in the model, while other factors and processes are only included superficially, indirectly, or not at all. Specifically, this means that the phases territorysearch, nesting, and breeding are the most relevant and detailed parts of the model, as these determine the selection of habitat and the survival of offspring.

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

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

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

4.2. Emergence

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

- **Territory size and population density:** The model assumes that skylarks occupy only as much area as they need to satisfy their nesting and foraging requirements, and that population size is limited by the amount of available habitat. This means that territories in high-quality habitat are smaller than in low-quality habitat. Scaling up, this leads to a pattern whereby population densities are highest in open landscapes with a diversity of crops, grassland, semi-natural habitat, and lower in landscapes with low habitat diversity or many woody features (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, as it is assumed that skylarks are not significantly impeded in their long-range movement by habitats that are unsuitable for foraging or nesting. Foraging movement by the male and by non-breeding individuals is also random, as it is desirable to show movement (to help model analysis) but unimportant to model this exactly.

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 all input files used for a simulation run alongside the run's output data, so that all run's 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 60m to vertical structures and allowing each pair approximately 3ha of suitable habitat (an average territory size in agricultural landscapes).

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

6. 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 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

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

Parameter	Default value	Explanation	References	Values tested
`eggtime`	11	Days spent in each	Glutz	
`nestlingtime`	9	juvenile life stage.	von Blotzheim	
`fledglingtime`	21		and Bauer (1985)	
`egg-`	0.03	Daily probability of dying	Delius (1965)	
`predation-`	0.03	in each juvenile life stage.	and Jenny (1990)	
`mortality`	0.01			
`nestling-`				
`predation-`				
`mortality`				
`fledgling-`				
`predation-`				
`mortality`				
`firstyear-`	0.38	Probability of dying in	Delius (1965)	
`mortality`		the first year of life after fledging.		
`migration-`	0.33	Probability of dying	Delius (1965)	
`mortality`		during on each winter migration.		
`migration--`	15 Sep –	Time period in which	Glutz	
`departure`	1 Nov	adults leave for / arrive	von Blotzheim	
`migration--`	15 Feb –	back from winter	and Bauer (1985)	
`arrival`	1 Mar	migration.		
`migrationdelay-`	15	Number of days which	Glutz	
`females`		females return later than males from winter migration.	von Blotzheim and Bauer (1985)	
`minimum-`	5000 m^2	Minimum required size of	Delius (1965)	
`territory`		a territory in ideal habitat.		
`mindistance-`	60 m	Minimum required	Jenny (1990)	
`toedge`		distance of suitable habitat to vertical structures.		

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

Parameter	Default value	Explanation	References	Values tested
`maxforageheight`	50 cm	Maximum preferred plant height and canopy cover for foraging.	Püttmanns et al. (2022)	
`maxforagecover`	70 %			
`nestingheight`	15–45 cm	Range of preferred plant height and canopy cover for nesting.	Jenny (1990)	15–20 / 15-60 cm
`nestingcover`	20–100 %			20–50 %
`matefaithfulness`	50 %	Probability that a female retains her partner from the previous year.	Jenny (1990)	
`nestingbegin`	10–20 Apr	Time in which nesting takes place.	Glutz von Blotzheim and Bauer (1985)	
`nestingend`	15 Aug			
`nestbuildingtime`	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.		
`visionrange`	200 m	Perception range while moving (used in the `matesearch` and `nonbreeding` phases).		
`offfield-nesting`	false	Allow skylarks to nest in unmanaged areas of the landscape?		true
`limit-territory`	false	Limit skylark territory sizes to a diameter of `movementrange`.		true

the landscape in concentric circles around their location, adding landscape pixels to their putative territory until they have accumulated a sufficiently large effective habitat area. The area that each pixel 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 block of habitat, 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.)

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. We enabled the latter to curtail the creation of very large territories (>40 ha) in a few cases; 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 landscapes consist of a grid of pixels with a resolution of 10m and have an extent of 270km^2-370km^2 (approximately; depending on the chosen input map). Each pixel is assigned a land cover class. It may also be associated with a farm plot, in which case it will contain information about the type and growth stage of the crop planted here. Farm management determines which crops are grown when, and when disturbance (e.g. mowing, harvesting, tillage) takes place.

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).

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

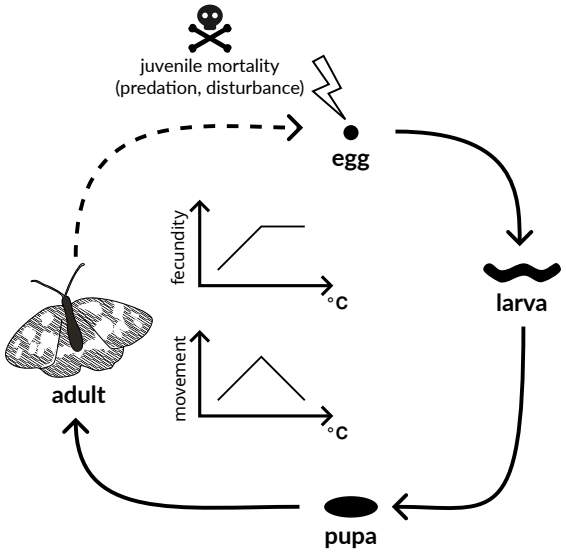

Figure 2: Phase diagram of the marbled model.

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. unmanaged grassy 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 kilometers) 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 local 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 landscape pixel 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 all input files used for a simulation run alongside the run's output data, so that all run's 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 suitable habitat, defined as unmanaged grassy areas, extensive grassland, or fallow land. 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 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). In addition, the model can be configured so that no activity takes place on days with rainfall (i.e. precipitation > 0, configured with rainactive).

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

Parameter	Default value	Explanation	References	Values tested
`minheight` `maxheight`	30cm 60cm	Range of vegetation heights suitable for oviposition.	Vandewoestijne et al. (2004)	
`mintemp` `maxtemp`	16°C 32°C	Temperature range suitable for activity (oviposition and movement).	Ebert and Rennwald (1991), Evans et al. (2019) and Gotthard et al. (2007)	18°C, 20°C 30°C, 28°C
`rainactive`	true	Allow activity on rainy days?		false
`movement`	"random"	Movement algorithm to use (see sections 7.2, 8.2).		"proximity"
`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`	100m	Maximum distance for a single movement step.	Cant et al. (2005)	
`self-avoidance`	0.9	Probability of declining to move to a location previously visited that day (with movement = "proximity").		
`maxind-perpixel`	Inf	Maximum local population density; probability of avoiding a location increases linearly up to this value.		10, 100
`maxeggs-perday`		Number of eggs laid per day under optimal weather conditions.	Reinhardt et al. (2007)	3–12

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

Parameter	Default value	Explanation	Justification	Values tested
`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`		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
`earliest-eclosure`	15th June	Range of dates in which eclosure takes place.	Ebert and Rennwald (1991)	
`latest-eclosure`	15th August			
`enforce-flyingperiod`	true	Ensure all individuals eclose in the empirically observed flying period.		false

7.2. Movement

Adults move a given number of steps each day, depending on the temperature (see above). Each step, an individual randomly scans landscape pixels within its perceptual range of 100m. If the pixel is suitable habitat, i.e. either arable or extensive grassland with plant heights within the required range (30-60cm), it moves to this pixel. In this case, the individual also lays an egg if it has not yet laid all its eggs for that day. If the pixel 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 pixel in its perceptual range.

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 18-30°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 maxindperpixel). 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).

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* (B. Klausnitzer & R. Reinhardt, **typeredactors**). 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

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