iLand - the book

1 Preface

Welcome to **iLand - the book**. This book is about iLand - the invidual-based forest landscape and disturbance model. iLand is a model of forest landscape dynamics, simulating individual tree competition, growth, mortality, and regeneration. The model addresses interactions between climate (change), disturbance regimes, vegetation dynamics, and forest management. iLand was developed over the last 12+ years in Austria, the US, and Germany. It started as a humble PostDoc project of Rupert Seidl and Werner Rammer, and grew over the years into a larger-scale collaborative effort that has been further developed and successfully applied in landscapes on three continents and two biomes.

Figure 1.1: The individual-based forest landscape and disturbance model

1.1 Who this book is for

We intend this book to be a useful resource for both beginners and advanced users of iLand. It should serve as an easy-to-use introduction to get started with the model for new users, but also covers more advanced topics for experienced users that want to dive deeper into the model.

The focus of the book is **how to work** with the model - if you are interested in **how the model works** (i.e., how specific ecological processes are modeled), you might be disappointed.

Model logic - down to the individual equations - is covered in scientific publications, and in the model wiki (https://iland-model.org). The wiki comprehensively documents the inner workings of the model with a high level of detail. In addition, the wiki also acts as a reference for model settings, input data and more. While comprehensive, the information is often spread across multiple pages and can be difficult to digest. The current book complements the iLand Wiki in providing a compact and accessible way to working with iLand.

1.2 Content of the book

The book consists of two parts:

The first part (**Part I**) is about getting to know iLand and guides you through the first steps of using the model. The second part (**Part II**) covers a number of advanced topics that may or may not be immediately relevant for you. The material covers technical aspects of using the model (for example, advanced JavaScript, running of simulation experiments locally and on compute clusters), and also advanced tasks in applying the model (such as parameterization of species, calibration and evaluation, and setting up new landscapes).

1.3 Other resources

Tip

Quick links
Homepage and extensive model documentation: https://iland-model.org
List of iLand related publications: https://iland-model.org/iLand+publications
Full model source code: https://github.com/edfm-tum/iland-model
Community channel for interaction and collaboration among iLand users: https://tinyurl.com/iland-model-discord

1.3.1 Contribute to the book!

We created this book with Quarto and RStudio under a CC0 1.0 licence - and the full "source code" of the book lives on GitHub: https://github.com/edfm-tum/iland-book/

Build date: 2024-07-17 - https://github.com/edfm-tum/iland-book/releases/tag/202407

We'd also love to get your help! Help is welcome in different forms, from comments to pointing at errors to revising the text or adding new chapters - feel free to reach out to us via Discord or e-Mail!

1.4 Acknowledgements

We'd like to thank everyone who contributed time, energy, and dedication in writing this book! It was a fun process over several month from the first realization that such a book could be useful, to brain-storming what it could include, to actually working on the content. The dedicated group of people at the Ecosystem Dynamics lab at TU Munich included: Martin Baumann, Kristin Braziunas, Christina Dollinger, Jonas Kerber, Johannes Mohr, Dominik Thom, Werner Rammer and Rupert Seidl.

We'd also like to thank everyone who over the years helped to develop and apply the model and create the awesome community of iLand users that we see today. Just to name a few: Katharina Albrich, Laura Dobor, Winslow Hansen, Brian J. Harvey, Tomáš Hlásny, Tyler Hoecker, Dominik Holzer, Juha Honkaniemi, Timon Keller, Yuta Kobayashi, Akira Mori, Mariana Silva Pedro, Zak Ratajczak, Julius Sebald, Andreas Sommerfeld, Ilié Storms, Kureha Suzuki, Anthony R. Taylor, Susan Willis, Soňa Zimová.

A particular thanks to mentors and friends that helped to get the iLand project going, and contributed resources and words of wisdom particularly in the early years of the project: Kristina Blennow, Manfred J. Lexer, Robert M. Scheller, Thomas A. Spies, and Monica G. Turner.

2 Introduction

2.1 Why iLand has been developed

Forest ecosystems cover roughly 30% of the global land area, store approximately three times more carbon than the earths' atmosphere, are hotspots of biodiversity, and provide a multitude of ecosystem services to society. However, many of these crucial ecosystem functions and services are severely threatened by anthropogenic climate change. Understanding the trajectories and sensitivities of forest ecosystems is thus crucial for sustaining the planet's life-support system, and for a transformation towards a sustainable, carbon-neutral society.

Forest ecosystems are complex adaptive systems. Their dynamics emerges from nonlinear interactions between adaptive biotic agents (i.e., individual trees) and their relationship with a spatially and temporally heterogeneous abiotic environment. Processes at multiple scales, from organs of individual trees (e.g., photosynthesis) to the landscape (e.g., wildfire) interact to form diverse and resilient ecosystems. In order to assess how climate change - which affects a variety of these processes - will impact ecosystems, and to sustainably manage these complex systems, we need to consider these multiple interactions among processes and scales. To that end we have developed a simulation model of forest ecosystem dynamics at landscape scales, the individual-based forest landscape and disturbance model iLand.

iLand explicitly simulates the principal adaptive agents in forest ecosystems, i.e., individual trees, over large areas. This scalability is achieved by (i) employing a pattern-based rendering of ecological field theory to efficiently model spatially explicit resource availability at the landscape scale, and by (ii) integrating local resource competition and physiological resource use via a hierarchical multi-scale framework.

2.2 How iLand works - in a nutshell

This section gives a brief discussion of design principles and describes the main simulation entities and processes. Please find more details in iLand publications - such as (Rammer et al. 2024) and (Rupert Seidl, Rammer, et al. 2012) - the iLand wiki, and following chapters in this guide!

2.2.1 Design principles

Individual-based modeling
iLand simulates each tree within the forest. This allows the model to capture how unique trees interact with each other (e.g., competition for light) and their individual growth over time. As a result, forest structure and organization emerge inherently from the simulation. Individual-based models permit tracking detailed effects of forest disturbances and offer realistic simulations of forest resilience.

Process-based modeling
iLand tries to model the underlying processes in a forest, like how trees convert sunlight into energy and grow. This means the model can be more reliable when simulating how a forest will react to changes in its environment. However, it's important to remember that process-based models sometimes sacrifice a bit of precision in favor of a more realistic simulation.

Multi-scale modeling
iLand considers the forest at many scales, both in space and time. It can simulate events happening over minutes (tree damage in a storm), days (effects of weather), or years (tree growth). The model also looks at areas from square meters (regeneration under tree cover) up to hectares (landscape units with similar climate and resources). This multi-scale approach allows fine-scale processes to influence the larger forest and vice versa.

Modeling for application
iLand was designed to be flexible to serve different scientific research needs. Users have lots of control over outputs (by selecting the specific data required at various scales), processes (by turning model components on or off for experiments), and scenarios (using built-in scripting language for customized events or management)

Transparent and efficient software design
iLand aims to balance scientific detail with a focus on being fast and usable within the limits of real-world computers. It makes design choices to improve performance, and by being open-source, developers around the world can see how it works and help improve it.

2.3 Main entities and environmental factors modeled in iLand

Individual trees
The model simulates each tree over 4m tall as a unique individual. It keeps track of the tree's size (diameter, height), different biomass components (leaves, branches, roots), and its health based on factors like stored energy and stress levels. iLand can simulate a vast number of individual trees within a landscape.

Saplings
For computational efficiency, trees under 4m tall are grouped into cohorts that share the same

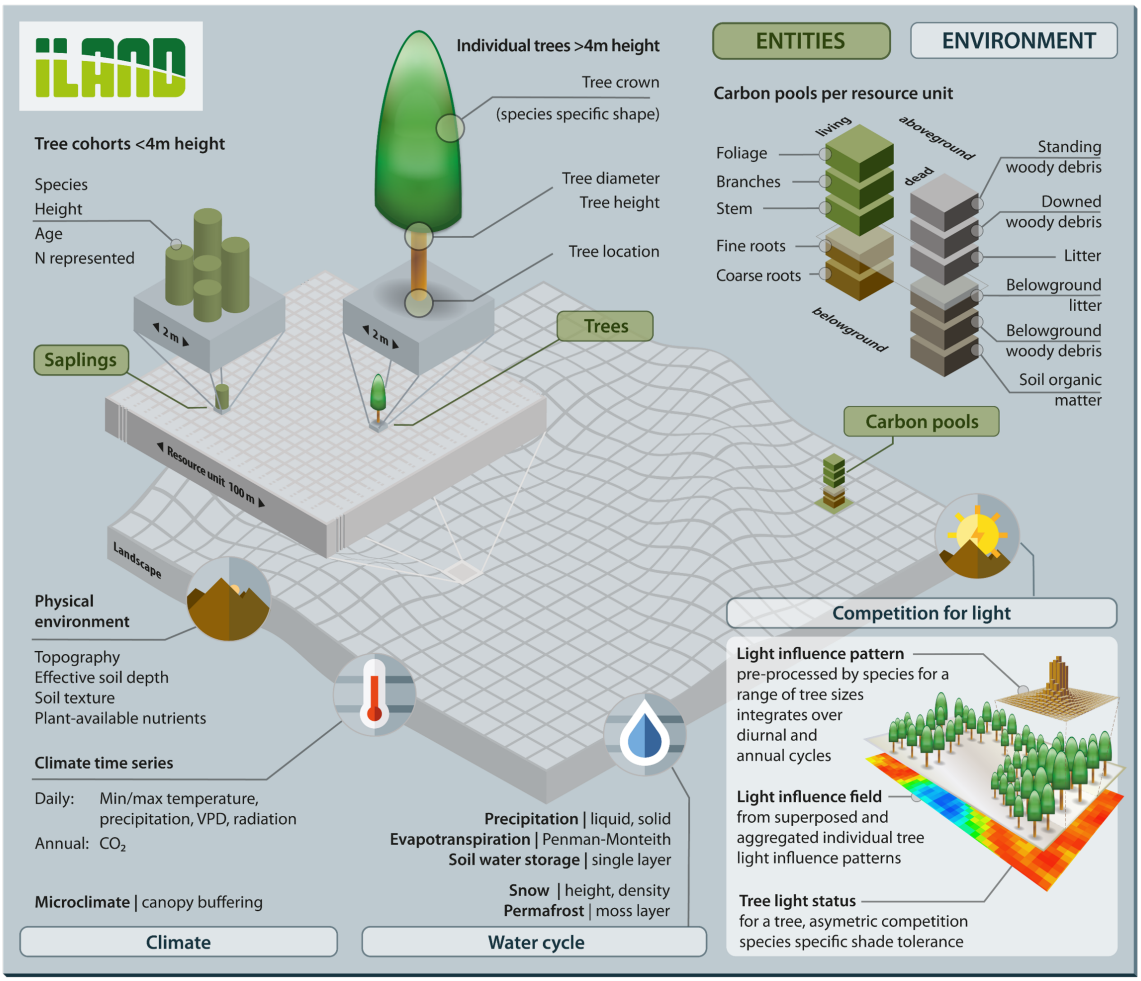

Figure 2.1: Overview of the main entitites simulated in iLand.

height and age. This allows iLand to track younger trees without the overhead of simulating each one individually.

Carbon pools
iLand tracks how carbon is stored in both living and dead parts of the forest ecosystem. This includes carbon pools in individual trees, saplings, deadwood, fallen leaves, and even the soil itself. These pools are essential to understanding a forest's response to climate change.

Environment
Forest development in iLand is heavily influenced by environmental factors (see Figure 2.1), including:

- **Climate**: The model uses daily weather data (temperature, precipitation, sunlight, etc.) provided at the local level. This means it can simulate differences in climate within a landscape, such as mountain areas with varying weather patterns.
- **Light:** A key driver of competition between trees, iLand uses a specialized approach to calculate how neighboring trees shade each other. This calculation takes into account a tree's size, shape, species, and the changing position of the sun over time.
- **Water**: iLand simulates the entire water cycle on a daily basis. This includes rainfall interception by leaves, snow accumulation and melting, soil water storage, and the water used by trees. It even has the option to include specialized modeling of permafrost (permanently frozen ground) dynamics.

2.4 Overview of demographic processes

iLand continuously simulates the core processes that shape a forest, like growth, death, and new tree establishment (regeneration).

Growth
iLand calculates how much energy each tree can gain from sunlight based on its position within the forest and environmental conditions (temperature, water, etc.). This energy is then used to grow different parts of the tree. Taller trees, older trees, and those in harsh environments grow differently, and iLand accounts for these factors.

Mortality
Trees in the model can die for various reasons. Old age increases the chance of death, but the biggest factor is stress. Stress happens when a tree can't get enough energy from sunlight to maintain itself, either due to lack of light or because its stored resources have been depleted.

Regeneration
New trees start from seeds. iLand tracks how seeds are produced, spread across the landscape, and how environmental conditions (like frost or light availability) determine if those seeds can

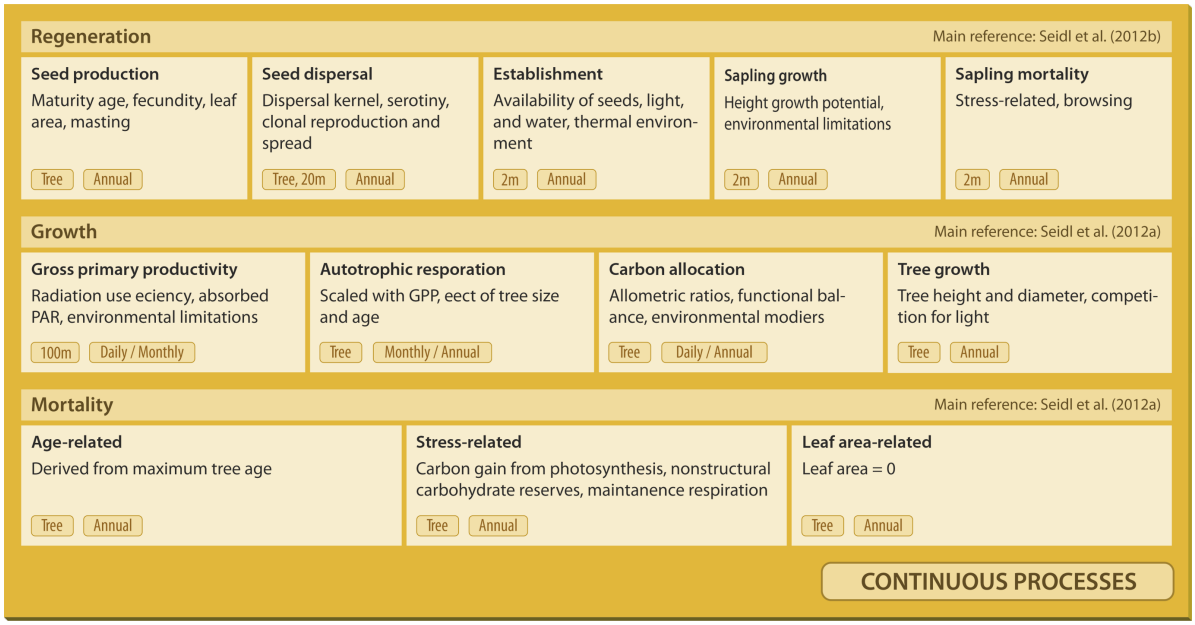

Figure 2.2: Overview of the main demographic processes modeled in iLand.

successfully grow into saplings. The model can even include special cases like seeds released from cones by fire (serotiny).

2.5 Overview of disturbances and management

Unlike the continuous processes of growth, mortality, and regeneration iLand also simulates sudden disruptive events, like natural disturbances or management interventions.

Natural Disturbances

- **Wind**
  iLand uses external data about storms occurring in the area. It then calculates how local wind speeds interact with the forest, including the effects of trees sheltering each other and the size of gaps in the canopy. Trees can be uprooted or broken, changing the landscape for other trees.
- **Wildfire**
  Wildfires in the model depend on fuel (deadwood, etc.), weather, how well fires are controlled, and the area's fire history. Once started, iLand simulates fire spread using wind, slope, and fuel conditions. Fires burn until they run out of fuel or reach a maximum size based on data.

Figure 2.3: High-level overview of disturbance agents and forest management modules implemented in iLand.

- **Spruce bark beetle**
  The European spruce bark beetle is a major threat, and iLand has a specific module for it. This module goes into high process detail, modeling how outbreaks start, how the beetles develop based on temperature, how they spread to find new trees, and how an outbreak naturally declines over time.
- **BITE framework**
  iLand uses a system called BITE to simulate many different pests and diseases. BITE allows the model to track where these agents can live, how they spread, how their populations change, and the damage they do to trees. This flexibility means users can study a wide range of forest health threats.

Management Interventions

iLand can simulate common forestry practices like planting, thinning, and harvesting trees. Using Javascript, users can precisely define when and where these happen, and which types of trees are affected.

It also includes a sophisticated management module (ABE) to model complex forest management strategies. ABE simulates different managers across the landscape who make decisions based on their goals, the current state of the forest, and how it's expected to change due to things like climate or disturbances.

2.6 iLand submodules

In addition to modules for natural and human disturbance, iLand includes a number of submodules or extensions that cover specific processes or topics. These submodules are optional and can be enabled for certain landscapes (e.g., the permafrost module is only relevant for boreal landscapes) or when addressing specific questions (e.g., the browsing module).

The following list gives an overview and links to the respective model documentation on the Wiki.

2.6.1 Browsing

The browsing module is a simple representation of the effect of browsing of ungulates on tree regeneration. The process affects growth rate of sapling and is parameterized using species specific browsing probabilities and an overall browsing pressure representing ungulate density (https://iland-model.org/browsing, (Holzer et al. 2024)).

2.6.2 Microclimate

The microclimate module uses a simple empirical model to predict near-ground (1m height) temperature as a function of macroclimate temperature (i.e., input daily climate data), topography, canopy density, and tree species composition. Microclimate temperature is simulated at 10m spatial resolution and updated annually, meaning feedback from disturbances, management, or increasing forest density on canopy temperature buffering is dynamically considered. Currently, three temperature-dependent processes (decomposition, spruce bark beetle development rates and overwintering success, and tree seedling establishment) can be optionally driven by either microclimate or macroclimate temperature at 1ha resolution, enabling exploration of the effects of temperature buffering by forest canopies across scales (Braziunas et al. In prep) .

2.6.3 Permafrost

The permafrost module simulates permafrost and soil-surface organic layer (SOL) dynamics at a fine spatial (1 ha) and temporal (daily) resolution. It efficiently models daily changes in permafrost depth, annual SOL accumulation, and their complex effects on boreal forest structure and functions (Hansen et al. 2023) (https://iland-model.org/permafrost).

2.6.4 Grass

An experimental module that focuses on the effect of forest floor vegetation on regeneration success of trees. The module has not yet been used in a study (https://iland-model.org/ground+vegetation).

2.6.5 Carbon / Nitrogen cycling

iLand tracks carbon content in live biomass, snags, downed dead wood, litter, and soil. The model also includes dynamic nitrogen cycling in the soil, based on the ICBM/2N approach. However, the feedback from soil nitrogen cycling to plant available nitrogen (and thus forest growth) is typically turned off, as, despite quite some effort, nitrogen cycling never worked sufficiently well (https://iland-model.org/soil+C+and+N+cycling).

2.7 iLand applications

iLand is a general model of forest ecosystem dynamics. It can and has been employed to tackle a wide variety of ecology- and management-related questions. Major applications of the model are:

- The resilience of ecosystems to disturbances arises from their multi-scale diversity in agents and responses, an aspect that is well represented in iLand. The model can thus contribute crucial capacities to studies aiming to understand and foster the resilience of ecosystem functions and services under intensifying disturbance regimes.
- Climate change adaptation is a major concern in forest management. Drastically changing environments necessitate new approaches in order to ensure a sustainable provisioning of crucial ecosystem services. With its process-based foundation and agent-based management engine iLand offers a robust framework for scenario analysis towards developing climate-smart management systems for the future.
- iLand keeps track of above- and belowground carbon stocks in forest ecosystems. It can thus be employed to study questions of forest C storage and exchange, e.g., in the context of the increasingly important question of climate change mitigation through forest management.
- Not only does iLand predict biological diversity in space and time explicitly, e.g., with regard to tree species richness and diversity, it also simulates many diversity measures highly indicative for other guilds of organisms (e.g., standing and downed deadwood, vertical canopy structure). iLand can thus be a powerful tool in the context of conservation of biodiversity, and for elucidating the functional roles of diversity.

Part I

Part I - Getting to know iLand

The first part of the book provides a short introduction into the iLand world. Here you'll learn the basics about operating the model. Specifically, you'll see how to install the model, run your first simulation and get an overview over the many components that the model has.

Figure 2.4: Start your journey with iLand!

3 Ingredients for an iLand project

This chapter will guide you through the basics of using the iLand simulation software. By the end, you'll understand how to set up a model project and run a simulation.

3.1 Installing iLand

iLand is free, open-source simulation software. You have a few ways to get it:

- **Stable releases**
  Visit the download page of iland-model.org for regular releases. Windows packages are available for direct download.
- **Development versions**
  For frequent bug fixes and feature updates, see https://github.com/edfm-tum/iland-model. These are also compiled for Windows.
- **Source code**
  If you need to build iLand for Linux, Mac, or want to modify the model, download the source code. Find details in the Developing iLand chapter.

If you have downloaded a release package, it contains the following:

- The executable files (`iland.exe` and `ilandc.exe`) and the required library files in multiple folders (Qt6*.dll, icu*54.dll, libEGL.dll, libGLESv2.dll, msvcp140.dll, etc) for running iLand on Microsoft Windows (XP, Vista, Windows 7/8/10/11).
- You can find a comprehensive example for running the model "out of the box" in the `example` folder.
- The full source code of iLand is packed in the `sourcecode.zip` file (or see iLand on GitHub)

Simply extract the ZIP folder to a location of your choice and start the model by double-clicking `iland.exe`. iLand doesn't have an installer and won't modify your Windows registry, so you can even run it from a USB stick! Please note that installing iLand within the Windows `Program Files` folder can cause issues due to security restrictions – user folders are recommended.

iLand is written in C++ and uses the versatile Qt toolkit (hence the QtXXX.dll library files). This means iLand is designed to work on Linux and Mac systems as well as Windows. We've successfully run it on Linux machines (like Ubuntu), and while we haven't tested it directly on a Mac yet, it should be possible there too.

Note that you have two executable files:

- `iland.exe`: This version includes the graphical user interface, making it easy to use, especially when you're getting started with the model.
- `ilandc.exe`: The console version runs in a terminal window. This is ideal for running multiple simulations in an experiment or if you're working with a high-performance computing cluster.

For more details, see the iLand Software page: https://iland-model.org/iLand+Software

3.2 Structure of a project

A "project" for iLand is everything that is needed to simulate a specific landscape (or a stand). You'd load a project in the model (actually a project file), and then "run" it for a number of simulation years. You can think of a project as a "landscape". A good starting point for getting familiar with iLand is the "demo" landscape that can be downloaded from the the website.

The main ingredients of a project are (more details below):

- project file
  This file holds all current settings of iLand and links to other types of data (e.g., climate data files). You can have multiple project files for the same landscape, typically stored in the main project folder. See comments on the Settings Editor in the iLand GUI chapter for details on how to change settings within the user interface - use this feature with caution, as it will overwrite the original settings in the `.xml` file!
- species parameters
  They define properties / traits of tree species. Tree species parameters are stored in a database, typically in the `database` folder. Related to species traits are also the light influence patters stored in the `lip` folder.
- spatial data
  Think of GIS data that defines the space of your project / landscape (note that generic landscapes and single stand projects are possible too!). This is often the extent of the simulated area and its properties (e.g., soil data, elevation, etc)

- climate data
  Time series information on climate that drives the simulations. Climate data is stored in a database (you guessed correctly, default location is the `database` folder)
- vegetation data
  Data on the initial vegetation at the start of the simulation
- additional data (for example, Javascript code files for forest management, specific spatial data for disturbance modules, etc)

Which type of data you work more with depends on the phase of your project. When you are setting up a new landscape, you need to collect all the data and provide it in a way compatible with the model. See Chapter on landscape setup for a deep dive. If you are working with an already existing landscape and previously created simulation scenarios, then your life is likely much easier. For instance, if your project folder already contains climate data for multiple climate scenarios, then you can *select* one of the climate databases by simply changing the project file (or the iLand settings).

3.2.1 The project file

The project-file is stored in the main folder (i.e., see the `projectBGD`-folder structure in Figure 3.1).

The project file is a XML-file, so its content is organized in a tree like structure. To adjust a parameter, it is probably best to use the search function of your favorite editor with the parameter name as the search term. The parameters with a description thereof can be found in the iLand wiki.

> ⚠ Working with the Settings Editor and XML-file in parallel
>
> Consider the Settings Editor as a form which is filled with the parameters set in the project file when opened and which rewrites the XML-file when closed with 'Save changes'. The Settings Editor loads data from the project file **only when the editor is opened**. That means, whatever changes you make to the project file while the Settings Editor is open, they won't be adopted by the Settings Editor. And more importantly: Those changes will be **overwritten** when the Settings Editor is closed with 'Save changes'. After saving changes, the values of the parameters either reflect the values they were before opening the Settings Editor, or new values defined in the Settings Editor.

3.2.2 Spatial layout

These files are stored in the `/gis`-folder.

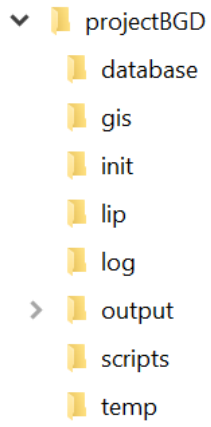

Figure 3.1: Suggested folder structure

Required

Resource unit or environment grid

- .txt- or .asc-file, resolution of 100x100 m
- contains the raster of the simulation landscape with cell values indicating the ID of the resource unit (=100x100 m cell)
- for more information see iLand wiki pages on simulation extent and subsection grid mode
- node in project-file: *model.world.environmentGrid*

Stand grid

- .txt- or .asc-file, default resolution of 10x10 m
- contains the raster of the simulation landscape plus neighboring forests with cell values indicating the stand-ID, which can be linked to information on vegetation
- for more information see the chapter on landscape setup and the wiki section on Setting up the stand grid
- node in project-file: *model.world.standGrid.fileName*

Optional

DEM (digital elevation model)

- .txt- or .asc-file
- The resolution of the input file can vary as iLand internally creates a DEM with a 10 m resolution
- iLand calculates maps with slope and aspect
- When no DEM is given, a flat topography is assumed
- for more information see wiki section on digital elevation model
- node in project-file: *model.world.DEM*

Wind topoModifier-grid

- .txt- or .asc-file, resolution of 100x100 m
- only needed if the wind module is used, scales the global wind pattern to a local value, values per resource unit
- node in project-file: *modules.wind.topoGridFile*

All grids have to align (same projection - here DHDN / 3-degree Gauss-Kruger zone 4, same extent). Shown here for the demo landscape in QGIS:

- DEM (npbg_dem50fix, 50x50 m) in green to grey palette
- Wind topoModifier-grid (wind_scale, 100x100 m) in pinks
- Resource unit grid (objectid, 100x100 m) in single value colors
- Stand grid (soid, 10x10 m) in rainbow colors

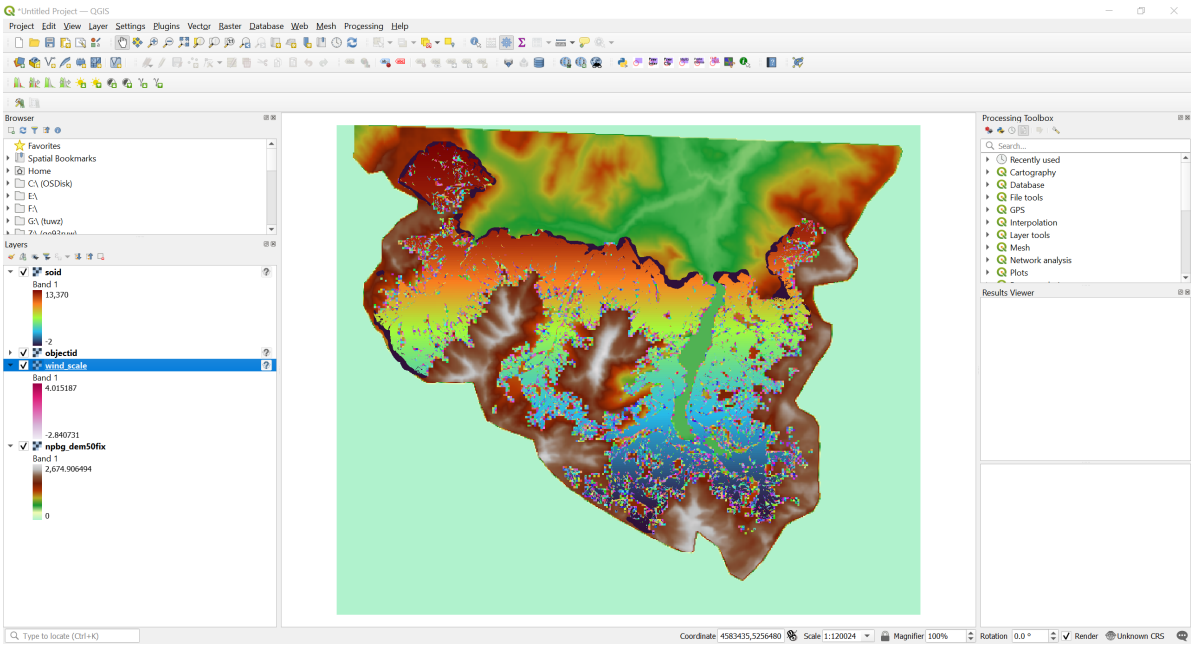

Figure 3.2: Spatial inputs in QGIS

3.2.3 Climate data

These files are stored in the `/database`-folder.

Required

Climate databases

- .sqlite-file, one per climate scenario
- contain tables with daily climate variables, table names can be linked to resource units via the environment file
- for more information on the environment file see wiki page on ClimateData
- node in project-file: *system.database.climate*

3.2.4 Species data

This file is stored in the `/database`-folder.

Required

Species databases

- .sqlite-file
- contains the species parameter for the simulated tree species, region-specific
- for more information see wiki page on species parameter
- node in project-file: *system.database.in*

3.2.5 Soil data

This file is stored in the `/gis`-folder.

Required

Environment file

- CSV or other text format
- contains for each resource unit the values for soil conditions and climate table name
- for more information see wiki section on spatially distributed parameters
- node in project-file: *model.world.environmentFile*

3.2.6 Initial vegetation

These files are stored in the `/init`-folder.

The files needed to set up the initial vegetation vary based on simulation set-up, presented here are the two most common versions:

Initialization file

- CSV or other text format
- contains information about the number of trees of a certain species that will be initialized on a specific stand-ID

- see wiki page on initialize trees and subsection about tree distribution input files
- node in project file: *model.initialization.file*
- note: *model.initialization.mode* can be `unit`, `map`, or `standgrid`

Snapshot

- .sqlite-file
- full representation of the internal vegetation and soil state of a landscape
- see section on Snapshots and chapter on spinup
- node in project file: *model.initialization.file*
- note: *model.initialization.mode* must be `snapshot`

3.2.7 Other input files

Required

Light interference pattern (LIP)

- bin-files
- LIP files contains several light influence patterns for various diameters and heights of a tree species
- LIP-files are not project-specific and are stored in the `/lip`-folder
- see wiki page on Lightroom

Optional

Files needed for management etc.
Usually these files are stored in the `/scripts`-folder

4 The iLand viewer

The "iLand viewer" is the the main application of iLand. You can start the program by running `iland.exe` (on Windows). When you just started iLand, no landscape is loaded (Figure 4.1). In order to load a project, select a project file (see previous section) either via `File->Open` or from the side panel. iLand remembers recently loaded projects (`File -> Recent files`). When you click the `Create Model` in the toolbar, iLand loads the initial state of the selected project / landscape as defined in the project file.

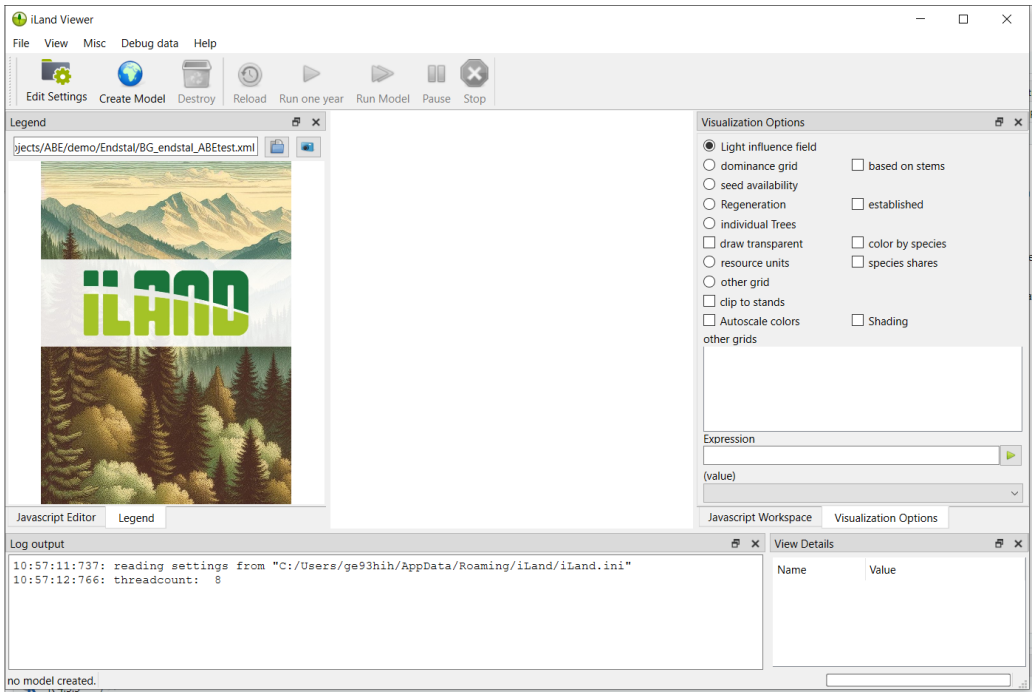

Figure 4.1: The iLand user interface after startup. The next step is to select a project file and load it into the model.

When a project is successfully loaded (indicated by a success message in the status bar at the bottom of the window), you can start a simulation! Do this by clicking the `Run one year` or `Run Model` buttons in the tool bar.

As soon as a project is loaded, iLand provides different means to visualize and interact with the model (i.e., with the simulated landscape and its vegetation) that are described in more

detail below (Figure 4.2).

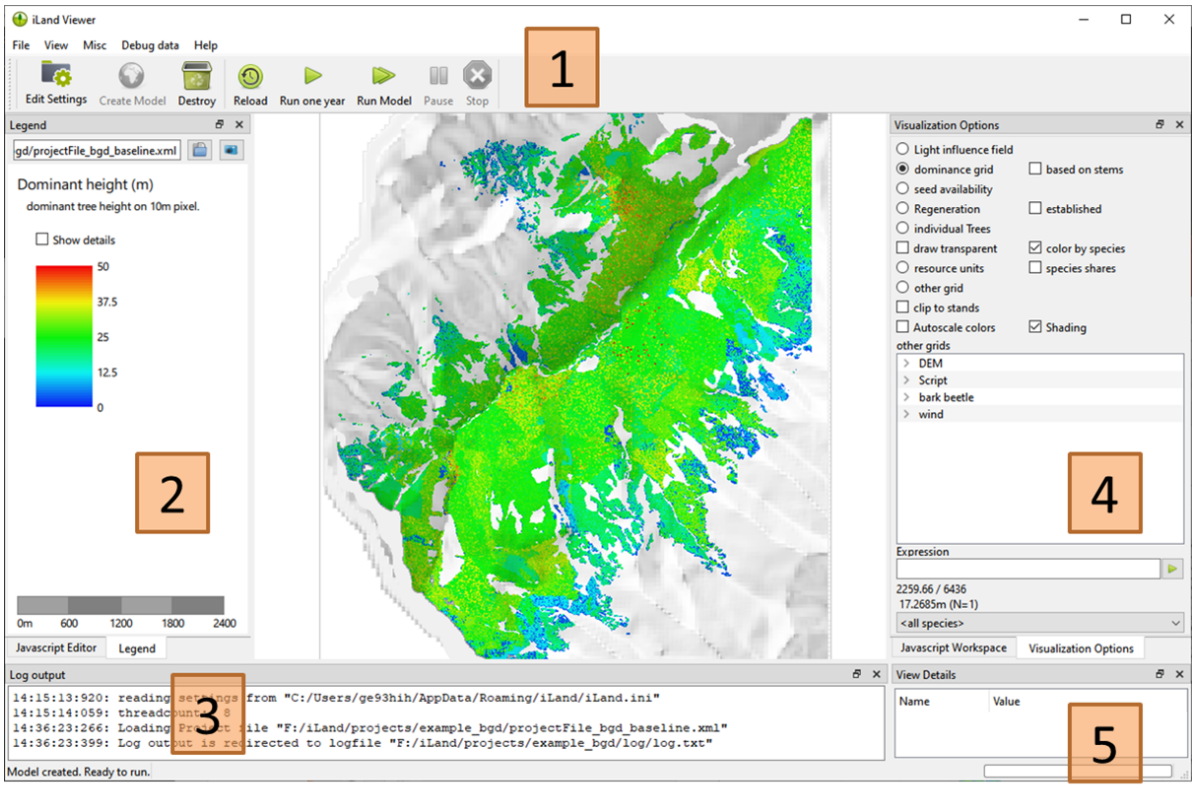

Figure 4.2: Overview of the main components of the iLand user interface. See text for more details. Note also that the positions of panels can be changed by the user.

(1) The toolbar provides access to the most important functions: you can create / destroy a landscape (destroying means deleting and freeing all resources), or reload the model (which is a quicker way to destroy and recreate the landscape). Furthermore, you can run the model for 1 year or multiple years. While running, you can pause or cancel (Stop button) the execution. You can also edit settings in the project file with the Settings Editor.
(2) The main panel shows the legend, but also provides access to the scripting engine - see Section 4.2
(3) The logging area shows all the log messages from the model (if logging is not redirected to a log file). This kind of log info is especially helpful to track configuration or model errors.
(4) There are many options to customize the main visualization in iLand (see Section 4.3)
(5) Clicking on a single tree or on a resource unit populates this detail view of tree/resource unit variables

4.1 Settings Editor

In iLand 2.0 the Settings Editor was introduced. The Settings Editor depicts a GUI-interface to work with the XML-project file in a more intuitive manner, especially for those who are new to iLand:

Figure 4.3: The start screen when the Settings Editor is opened for the first time in a session. On the left is a navigation bar with which various settings can be selected. The project description shown is contained in the project file and can be edited using the "Edit Project Description" button. There are various filter settings in the top bar to simplify the view of the parameters if necessary. If the Settings Editor is closed and reopened in a running session, the last active setting appears.

Parameters can be set both via the Settings Editor and directly via the project file. Above all, comments are also part of both representations:

Figure 4.4: Comparison of the project file and the corresponding display in the Settings Editor. Comments are indicated in xml by . If a comment is available for a parameter, the pencil symbol to the left of the corresponding parameter label is highlighted in the Settings Editor. If you hover over the symbol with the mouse, the comment appears. If you click on the pencil symbol, a dialogue opens in which the comment can be edited.

However, it is important to note that in the end it is still the XML-file that determines the simulation. The Settings Editor only provides a convenient way to edit the project file, so it is still worth getting familiar with the project file and its structure.

> ⚠ Save Changes
>
> The `Save Changes` and `Show changes` buttons will be activated when parameters have been changed. `Show changes` opens a dialog with which you can follow changes made during the active Settings Editor session. By pressing `Save Changes` all changes are accepted and saved to the project file, which is **overwritten** in the process.

4.2 Main side panel

The main side panel (typically on the left side of the window) contains two "windows" that are organized as tabs: The "legend" shows the legend for the current visualization and a ruler, while the "Javascript editor" tab lets you access the JavaScript engine.

4.2.1 Legend

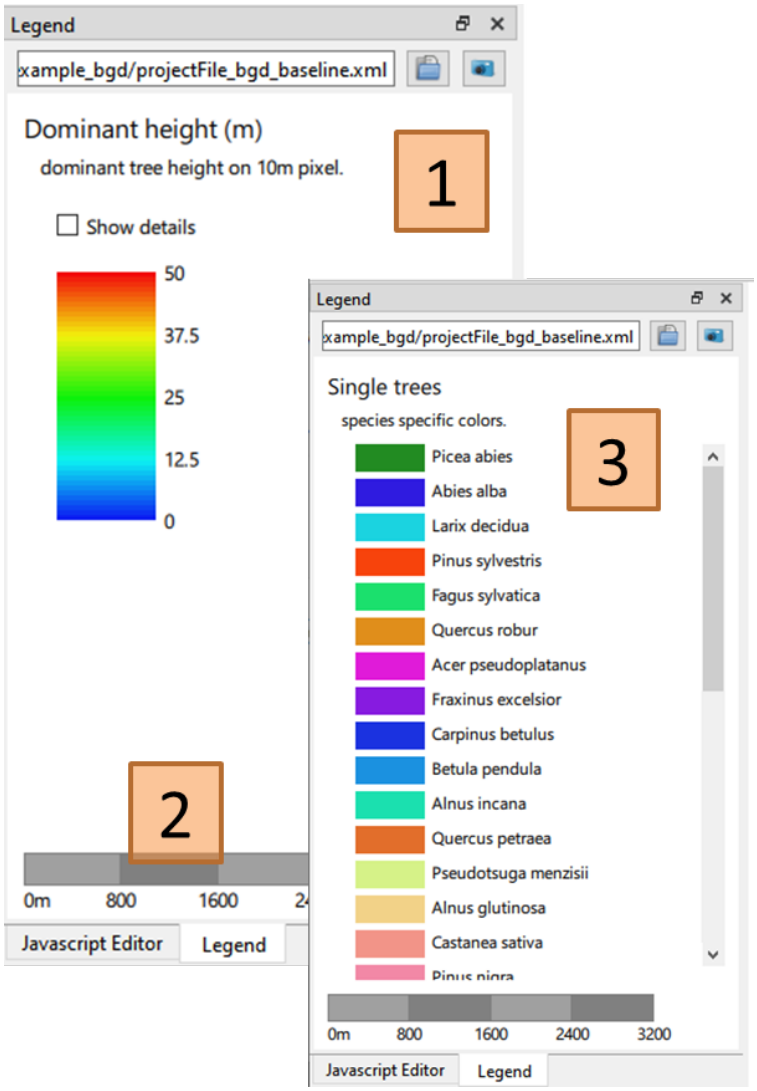

Figure 4.5: The legend shows either continuous or categorical data and a scale bar to indicate the zoom level.

(1) Click the 'open' icon to select an XML project file (or use the `File->Open` command). Note that selecting a file does not *load* the project!
(2) The legend depends on the selected visualization option; there are, broadly speaking, two types of legends: continuous numerical values (such as tree heights or standing volume) and categorical values (such as tree species or stand IDs). For continuous variables, the value range can be edited after checking "show details".

(3) The scale bar shows the scale of the current visualization in meters.

4.2.2 Javascript editor tab

Figure 4.6: The Javascript editor tab allows interactive writing and running of JavaScript code.

The **Javascript editor** tab allows the direct access to the JavaScript engine in iLand. You can use the editor window to add / edit and work with JavaScript code. This can be user-defined functions, calls to the iLand system libraries, or even full programs (Figure 4.6). Moreover, you can execute/run code using `Ctrl+Enter` key combination interactively - that is, you can execute code that retrieves data from the model (e.g., a map of currently dominant species), or that even modifies the state of the model (e.g., by harvesting / killing trees). The text output (e.g. results of `console.log()` calls) is printed to the iLand log window. See Section 6.3.3 for

more details on the code editing capabilities of iLand!

4.3 Visualization

iLand provides means to visualize what is "going on" in the simulation. This feature has many times proven to be very important and useful: It allows researchers to get a feeling of what is really happening in the model, and it is a powerful tool for spying problems, but also for finding cues for solving those issues. The visualization options are in typically on the right side of the main window (denoted with **(4)** in Figure 4.2).

The following picture zooms in on the available options.

Sections include **(1)** basic options, **(2)** show additional layers provided by submodules (e.g, disturbance modules), **(3)** apply and visualize user defined expressions, and **(4)** visualize only a selected species.

These basic options **(1)** are:

- **Light influence field**
  Shows the light influence field (LIF). Red means low, blue high light levels. Spatial resolution is a 2m grid.
- **dominance grid**
  Shows maximum tree and crown heights at 10m resolution; Areas outside of the project area are either drawn grey (areas that are assumed to being forested) or white (for non-forested areas). Enabling the "based on stems" option shows the top height only based on tree stems
- **seed availability**
  Shows the seed availability of a selected species (the dropdown species filter in **(4)**) on the 20m seed dispersal grid. Colors indicate the density of seeds on a cell.
- **regeneration**
  Shows the state of the regeneration layer (2m resolution). The height of cohorts are shown for a selected species (dropdown species filter in **(4)**). If no species is selected, the value is the maximum height of all living cohorts on that pixel. Note, that the maps are generated on the fly and may be slow to create. If the "established" box is checked, only newly established cohorts are shown (with value=1 for pixels with new cohorts).
- **individual trees**
  Shows tree crowns from bird's eye perspective (i.e. as stylized circles with the radius indicating the size of the tree crown). The color indicates their competitive status as described by the light resource index (LRI, again, red means low LRIs, see competition for

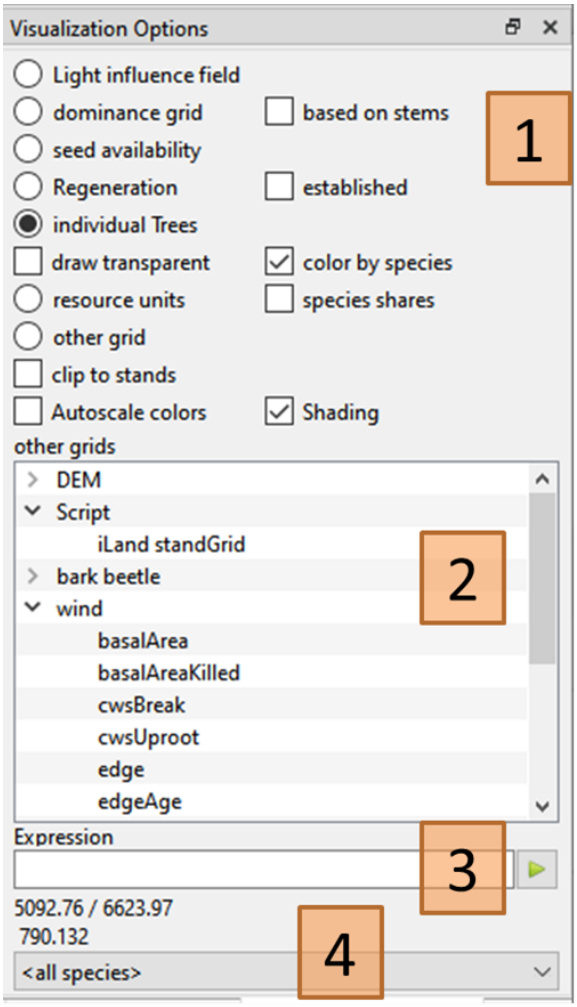

Figure 4.7: The "visualization options" tab of iLand controls what and how information is displayed in the model.

light). Check 'color by species' to draw trees in species-specific colors. The "draw transparent" option adds transparency, which may help visualize dense multi-story forests. Use the species filter **(4)** to limit drawn trees only to a single species.

- **resource unit**
  Shows the resource units, i.e., a 100m grid used for the calculation of, e.g., the water balance. Shown is either the result of an expression (see below), or species proportions (when "species shares" is checked): In this case, if no species is selected **(4)**, each resource unit is drawn with the color of the dominant species, i. e., the species with the highest basal area share. The color is dashed if the dominant species has a share $< 50\%$ (i.e., shows not a clear dominance). If a species is selected, then the basal area of that respective species is drawn (you may check the "autoscale" option).

- **other grids**
  Some modules of iLand provide additional grids that can be visualized **(2)**. Examples are a digital elevation model, details of disturbance modules, and several layers of the agent based forest management engine. Each layer has a brief description and has a legend (i.e., color ramp and value range) shown in the main panel in Figure 4.2.

- **additional checkboxes**
  "*Clip to stands*" masks out grids (especially useful for the "other" grids) with the project area (as given by the stand grid). "*Autoscale colors*" scales the visualization to the value range provided in the data. Note, that this does not work for all options. "*Shading*" works only when a DEM is available; it then overlays the DEM shading in the current visualization.

As described above, "Other grids" (section **(2)**) show different layers that are provided by modules of iLand and works much like a GIS application.

The "Expression" box (section **(3)**) can be used to visualize trees or resource units based on user-defined mathematical expressions (such as `dbh*dbh/4*3.1415`). Which variables are available from iLand depends on the type of object; for trees, you can use dimensions (`dbh`, `height`), biomass pools and other state variables (see tree variables). For resource units, you can use current climate (`meanTemp`, `annualPrecip`), stand-aggregates (`volume`, `count`) and many more (see resource unit variables). See the expression page on the wiki for a detailed description about syntax and available functions.

> **Tip**
>
> Clicking on an element in the visualization (e.g., a tree, or a grid cell) shows details of the respective object in the "View Details" window (see **(5)** in Figure 4.2). The details show (in most cases) all the available variables, which can then be used for expressions. Expressions are used in many areas in iLand. One example is management, where tree selection / filtering is based on expressions. You can test (and debug) these expressions

simply by playing around in the user interface!

4.3.1 Example visualizations

The following figure exemplifies how iLand can be used to visualized different types of data:

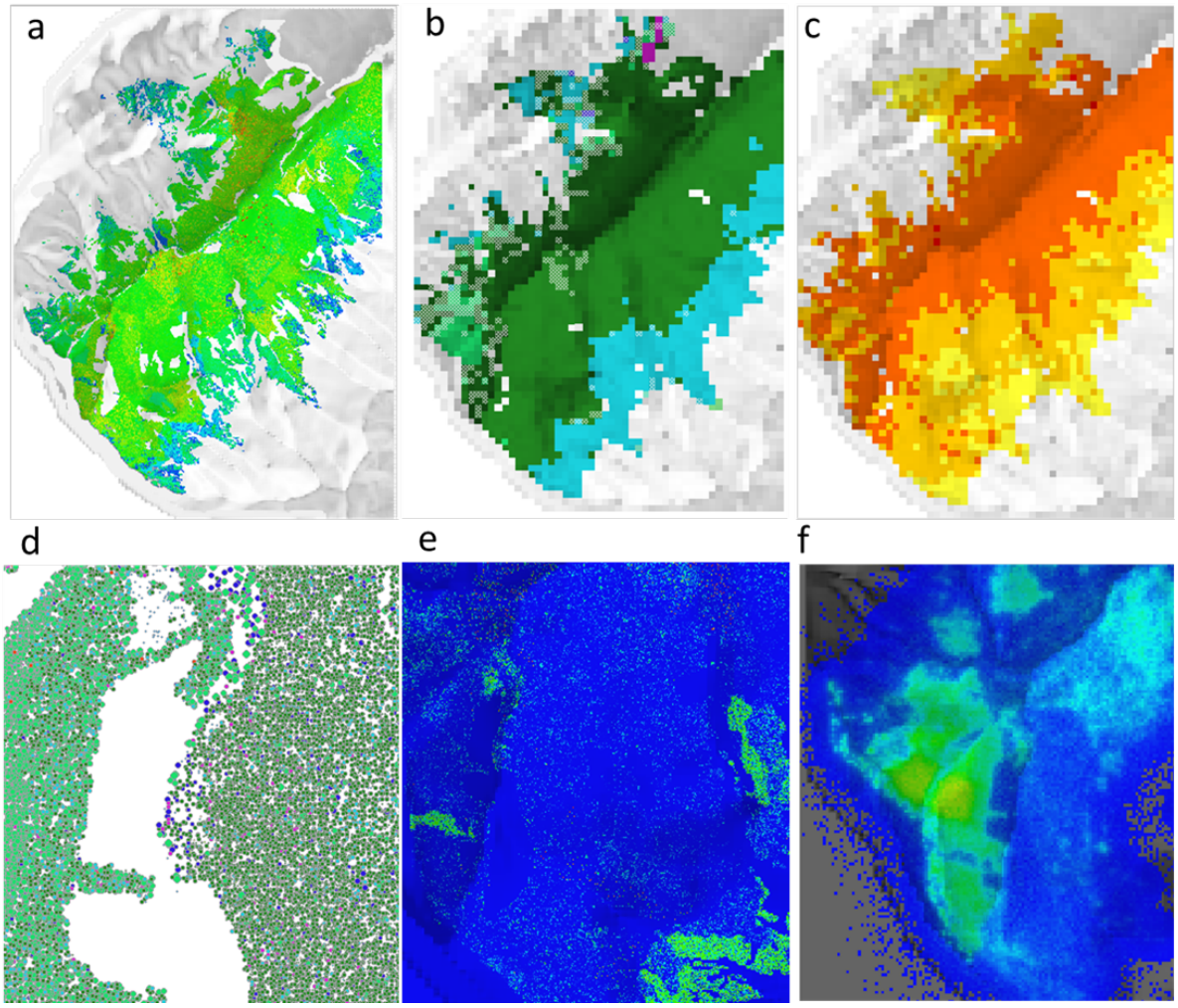

Figure 4.8: Example visualizations taken from the "demo" landscape. See the table below for detailed explanations.

Panel	Description
a	"Dominance grid", 10m top height. The background shows the digital elevation model, "shading" is enabled.
b	Dominant species on resource unit (100m) level. Colors indicate species (e.g., dark green is spruce). Dashed cells indicate that no species has more than 50% of the basal area.
c	A layer of the bark beetle module, namely the number of potential bark beetle generations per resource unit in the current year.
d	Individual trees (zoom in) - colored circles are individual trees (color = species). The size of the circle shows the crown size.
e	Regeneration layer (2m spatial resolution). The color indicates the maximum height of any cohort on each cell (blue = 0m or no regeneration)
f	Available seeds for a species (20m spatial resolution). Green / orange hues indicate more available seeds (spotty pixels on the bottom right indicate long distance dispersal of seeds)

4.3.2 Using the mouse and keyboard

You can use your mouse and keyboard to pan / zoom within the landscape:

- **Zoom in/out**
  Use the mouse wheel (while positioned over the landscape) or use the '+' or '-' keys of the keyboard. For keyboard zoom the focus must be on the landscape, i.e. a single click into the landscape might be necessary.
- **Pan**
  Click with the left mouse button and drag. After releasing the mouse button the landscape is redrawn (so there is no visual indication during the drag process except the changed mouse cursor).
- **Hover for location and value**
  The coordinates under the current mouse position are shown above the species filter (section **(4)** in Figure 4.7). The coordinates are always in meters, and are relative to the location defined in the project file. In addition to the coordinates, the "value" under the mouse is shown (details depend on what is present at the cursor location).
- **Screenshots**
  You can copy a "screenshot" of the main visualization area to the clipboard by pressing `Ctrl-P` or by selecting "copy image to clipboard" in the `View` menu. In addition, the screenshot is saved as "screenshot.png" in the project directory.

- **More useful keys**
  Press `F5` to refresh the visualization area, press the `F6` key to zoom out to the full landscape.

4.4 Miscellaneous features

4.4.1 Default project file

iLand defaults to the last loaded project file. To accomplish this, iLand saves the name of the last successfully loaded project file in a small text file named `lastxmlfile.txt` located in the same directory as the executable itself. On startup, iLand uses this information if present. This behavior is overruled, if a XML file name is provided as a command-line argument to the executable (e.g., `iland.exe e:\\iland\\project\\project1.xml`).

4.4.2 The About box

A little "About" dialog is accessible from the toolbar (`Help->About iLand`). The about box (Figure 4.9) shows details about the open source licence, and, particularly useful, detailed information about the version of iLand that you are using. Specifically, it shows the current version (here: `2.0 (rc)`), and the building environment and Qt version (`MSVC 64 bit Qt 6.5.0`- this is mostly useful for technical troubleshooting). Moreover, the second line shows the current branch in GitHub (typically `prod` for production releases or `dev` for development branch), the short name of the last "git commit" (`9d15e8b9`) and the build date of the model. The git commit is linked to GitHub, and describes the exact state of the iLand source code that was used to build the executable. See also the chapter on building iLand.

4.4.3 The Expression plotter

iLand contains a little function plotter tool that is based on the expression engine of iLand. You can plot arbitrary functions using the syntax and built-in functions of the expression engine (such as `polygon()`, or `if()`). The first unknown symbol encountered is treated as the predictor variable (you get a warning when additional variables are in the expression). The function plotter can be accessed via the menu (`Misc->Expression plotter`) or from the settings editor. The tool can be very helpful for visualizing allometric or response functions often used for species parameters.

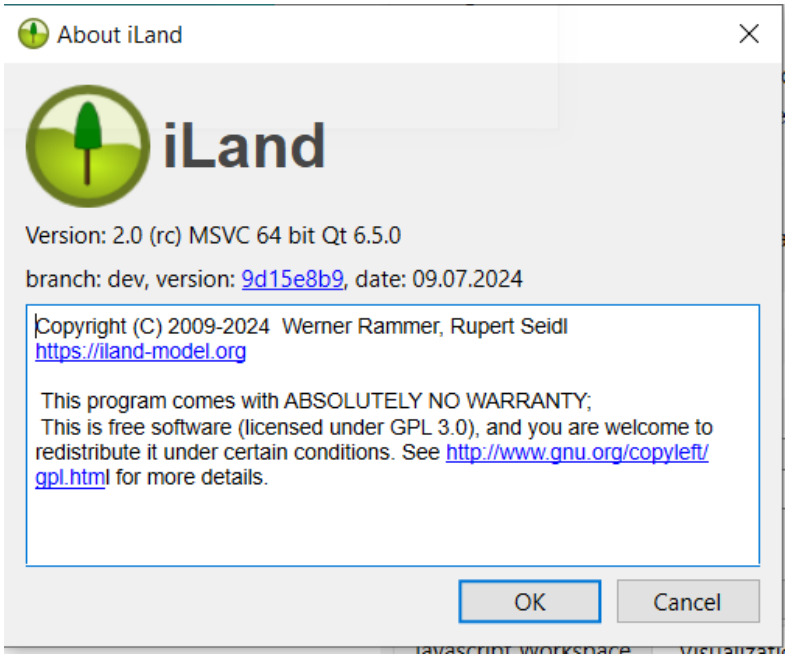

Figure 4.9: The "About iLand" box provides useful information to determine the exact version you are using.

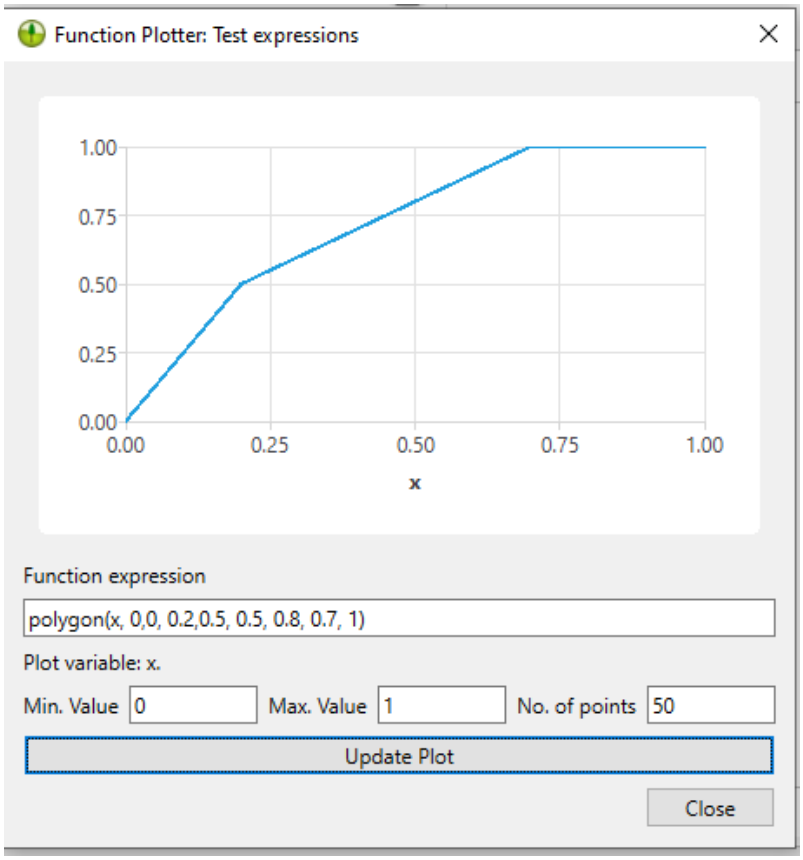

Figure 4.10: The function plotter is tool to test and visualize iLand expressions.

5 iLand outputs

iLand can provide a wealth of output data, with differing spatial and temporal resolution and on different levels of aggregation. The standard way is via "outputs", which are described now. Outputs in iLand are per default created as a SQLite database. Every type of output is represented by one table within this output database, and the user controls which outputs are active. Thus, the **results of one simulation run are contained in one single file**. Each iLand output database contains a small table named `runinfo` which stores a timestamp when the simulation was started, and iLand version information.

Since iLand is a dynamic simulation model, most outputs have a time-dimension; many outputs describe stocks (at a certain point in time) or flows (for a specific period). Data can be very fine-grained - for example, the `tree` output contains a single row for every tree and for every year of the simulation. This is not practical (particularly on landscape scale!), therefore different levels of aggregation are possible. For example, the `landscape` output describes average values for timber volume, biomass, etc. for each species but for the whole landscape. While you (technically) can enable all outputs all the time, it is usually not practical to do so. With more outputs enabled, simulation results take up moch more disk space. In addition, the time for running the simulation (more data to wrote) but also the time for analyzing the data (more data to process!) goes up. Striking the right balance here is particularly important for large simulation jobs with hundreds of individual simulation runs.

5.1 Output database

The output database is written to a location specified in the project file. Basically, there are two options:

1. reuse a database file: in this case every run will overwrite the preceding run. This is an option when still "playing around".
2. create a database for each run: this is clearly the option of choice for larger simulation jobs.

The filename of the output-database is defined by the *home.database.out* key in the project file. If this key contains a simple string, a database with that is opened (and overwritten). If you want to go with option 2, a special syntax can be used for the filename to automatically add a

time (unique) timestamp. To do this, add `$` ch (e.g. `projectX_$date$.db`), where the string between the `$date$`-signs is replaced with a time stamp value (e.g. "20091021_143059").

5.2 Controlling outputs

The iLand wiki contains a list with the detailed definition of all available outputs under https://iland-model.org/outputs. Currently, iLand provides ~25 different "outputs". Each output is configured by a corresponding section in the project file (in the section `output`). Every output can be turned on or off (using the `enabled` key), and some can be further customized, in many cases to limit the generation of output data, e.g. by limiting an output to only run every 10 years.

Here is an example in the XML project file:

```
<output>
   <tree> <!-- individual tree output -->
     <enabled>tree</enabled>
     <filter>year=10 or year=20</filter>
   </tree>
   <treeremoved> <!-- individual tree output -->
     <enabled>false</enabled>
     <filter></filter>
   </treeremoved>
   ....
```

In this example, the output `tree` is turned on, but writes data only in year 10 or 20. The output `treeremoved` is disabled.

The Settings editor provides a graphical user interface for configuring (most) aspects of outputs, see Figure 5.1.

5.3 Other types of outputs

In addition to the standard "outputs" described above, iLand provides more ways to extract / store data.

One option are "debug outputs" (https://iland-model.org/Debug+Outputs): Debug outputs are more low-level and typically highly detailed outputs. While not very useful for standard applications, there are cases where the high level of details may be important. For example, debug output allow to to track changes in the water cycle with daily resolution. Debug outputs

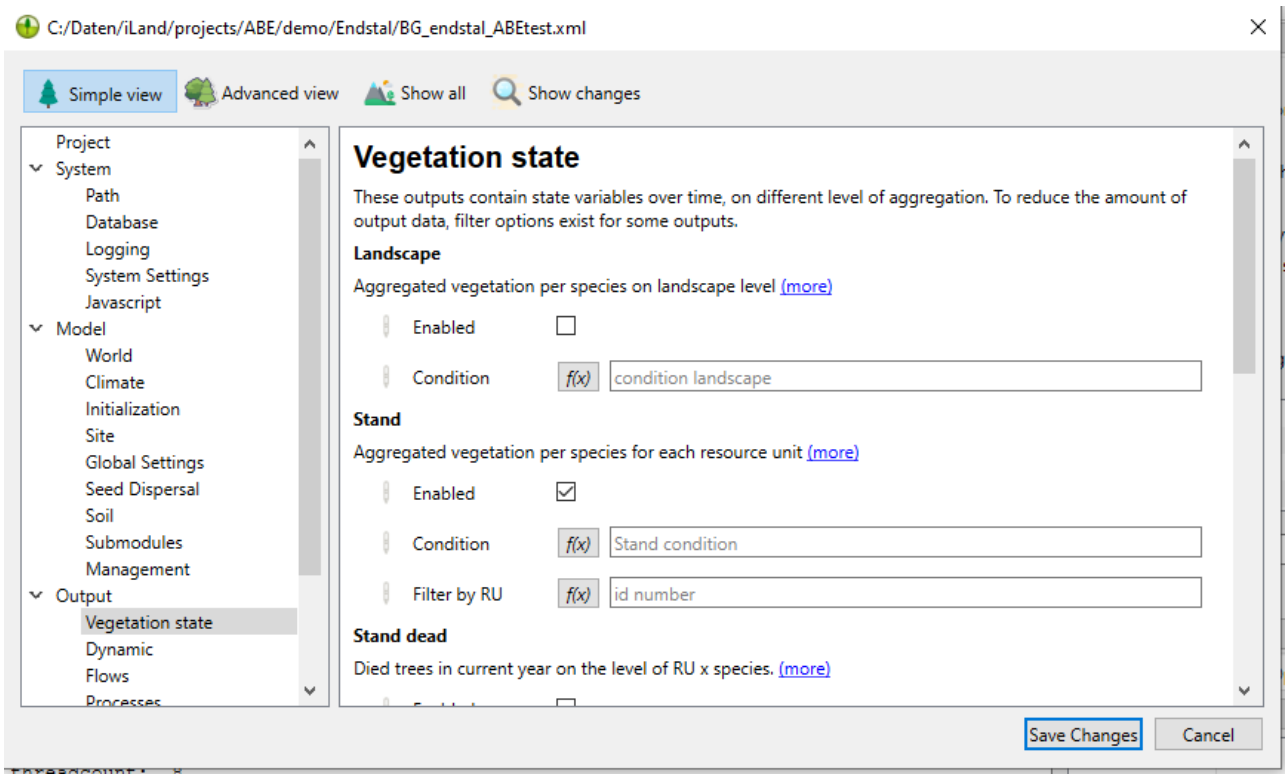

Figure 5.1: The Settings Editor provides a convenient interface for configuring outputs.

are in the form of simple CSV text files and have some quirks (e.g. the way how to enable / disable them)

Another option is gridded data: here you use JavaScript to access spatial data in iLand and write to raster files. This can be rather straightforward (write a map of basal area for a given species every 10 years), but can involve relatively complex (spatial) computation. Below is (a rather complex) example. See Scripting chapter and https://iland-model.org/apidoc/classes/Grid.html for more details!

```
var wind_grid; // global variable to hold the grid

// this event / function is called by the wind disturbance module
// the function keeps a running sum for each 10m pixel and
// counts the trees that were killed on a cell
function onAfterWind() {
   if (wind_grid == undefined)
       wind_grid = Wind.grid("treesKilled");

   var g = Wind.grid("treesKilled"); // killed this year
   // add to the total number of trees killed:
   wind_grid.combine( 'killed_sum +
   killed_now', { killed_sum: wind_grid, killed_now: g } );
}

// this function is called by iLand at the end of every simulation year
function onYearEnd() {
    if (Globals.year % 20 == 0) {
      // every 20 years run a spatial analysis to find contiguous patches
// (of cells that were affected by wind in the last 20 yrs)
        var p = wind_grid = SpatialAnalysis.patches(wind_grid, 5);
        Globals.saveTextFile(Globals.defaultDirectory('output') +
        '_stormPatches_' + Globals.year + '.txt',
        SpatialAnalysis.patchsizes + "\n");
        wind_grid = Wind.grid("treesKilled"); // +- reset the variable
    }
}
```

5.4 Analyzing iLand outputs

Once iLand has finished a simulation, you can start to look at the results of the simulation. In most cases this process requires to analyse the content of the output database (see above).

A typical approach (many different approaches exist!) is to use R to do the job. The basic pattern could look like this:

- open the data base in R. For this the `RSQLite` R package is very useful
- read the tables (i.e., the "outputs") of interest
- do some data wrangling (e.g. using the `dplyr` package)
- plot results

Here is a very simple example showing the approach:

```
library(RSQLite) # load the RSQLite package to work with SQLite databases

# Step 1 - load the data
# open the output data base of the simulation and read the "landscape" output.
# Adopt the file name!
db <- dbConnect(RSQLite::SQLite(), "output/output.sqlite")
lscp <- dbReadTable(db, "landscape")
dbDisconnect(db)
summary(lscp) ## show some stats

# Step 2 - analyse the data
library(ggplot2)
library(dplyr)
# plot time series of volume/ha for spruce and beech
ggplot(lscp %>% filter(species %in% c("piab", "fasy")),
       aes(year, volume_m3, col=species)) +
  geom_line()
```

5.5 Data analysis with R - real world example

The following shows a real-world data analysis example with R and R markdown.

Christina Dollinger

April 16th, 2024

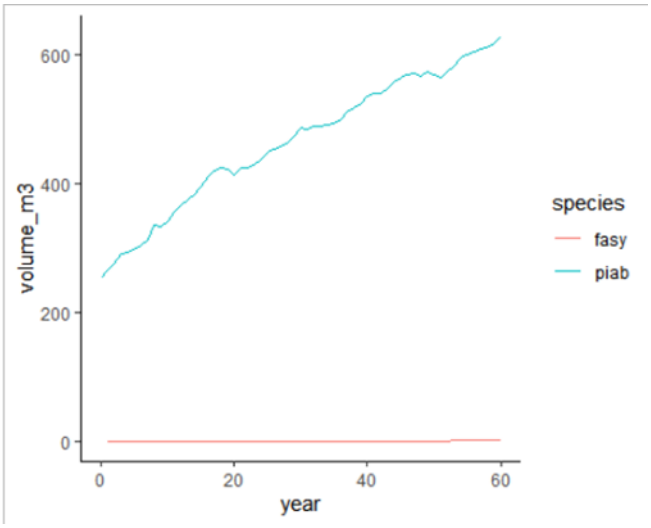

Figure 5.2: Example output of the data analysis script

Part II

Part II - Digging deeper into iLand

Part II covers more advanced topics in using the model. The chapters in this part are relatively independent and cover advanced ways of using the model (e.g., using JavaScript), a primer on how to get your own landscape ready for simulation (covering initialization, evaluation and parameterization of species), and finally how to run iLand on computer clusters.

Figure 5.3: Continue your journey into the depths of the model!

6 Scripting

6.1 Concepts of scripting in iLand

The scripting approach used in iLand relies on the scripting capabilities of the Qt-framework. Qt includes a fully ECMA compatible JavaScript engine, which means that the JavaScript capabilities of iLand are very similar to those of a modern browser. While execution speed of scripts have been heavily optimized over the years (e.g., they are compiled just-in-time by the system), there is still a performance gap between the world of JavaScript and the native C++ world of iLand.

6.2 Introduction to JavaScript

JavaScript is a versatile programming language known for its adaptability and widespread usage in various applications beyond just web development. It's like a Swiss Army knife for coding, enabling you to create interactive and dynamic content, not just for websites, but also for other tasks like data analysis, automation, and simulations.

Its user-friendly nature and extensive online resources make it relatively easy to learn, especially if you're already familiar with languages like R. Additionally, JavaScript has a large community of developers who constantly contribute to its development, ensuring that there are plenty of libraries and frameworks available to simplify complex tasks.

Since JavaScript is mostly used as the "language of the web" (e.g., for dynamic web sites) a lot of documentation and discussions on sites such as StackOverflow are web-centered. But do not despair: there is plenty information on "pure" JavaScript available. As a rule of thumb, look for content that does not mention typical web "frameworks" such as "React JS", "Angular", "jQuery". However, many sites about web development do have good information on "pure" Javascript!

> **Note**
>
> The JavaScript implementation used in iLand supports the full ECMA-262 specification - this means that all of the standard objects and functions are available (see https://doc.qt.io/qt-6/qtqml-javascript-functionlist.html for a full list). Also JavaScript libraries can

be used - as long as they do not require browser-specific functionality.
These sites (among many others) provide good documentation about pure JavaScript: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference, or https://www.w3schools.com/jsref/

6.3 JavaScript in iLand

When you run JavaScript code in iLand (we'll get to how to get code into the model in a second), then the code can access objects and functions that are provided by the model. These "links" to the model via objects and functions comprises the Application Programming Interface (API) of iLand. The documentation of the iLand API is at iland-model.org/apidoc. Code can be executed manually, or run automatically when the model runs (e.g., to execute forest management).

There is a single JavaScript "engine" in the model, which means that all JavaScript objects and functions share the same environment. As a consequence, elements can be overwritten! For instance, you could manually change a function that has been loaded previously; it also means you can only have one instance of an event handler (see Section 6.3.2.4). While a basic JavaScript engine is already available after startup of the model executable, iLand API objects are only available when a project is (successfully) loaded.

6.3.1 Ways to load JavaScript code

There are multiple options to load and execute JavaScript code in iLand. Note, that all JavaScript code shares the same execution environment!

6.3.1.1 Specify a code file in *system.javascript.filename*

iLand loads this JavaScript code file early on, and looks in the directory specified with *system.path.scripts* in the project file.

6.3.1.2 Specify a code file with a specific iLand sub module

Several sub modules of iLand allow specification of a dedicated JavaScript file. Examples are base management, ABE (agent based management engine), BITE (biotic disturbance engine), and some specialized outputs (e.g., the Stand Development Stage).

6.3.1.3 Write and run code manually in the iLand user interface

Code that is **executed** in the iLand JS code editor (see Figure 6.1) affects the JavaScript environment, i.e. you can create objects and functions, and run code. You can also run code that affects the vegetation, for example forest management scripts. Note that changes (e.g., harvested trees) may not be visible in the user interface immediately. You might need to refresh the visualization (`F5` and `F6` keys, see also the `View` menu), or even run an additional year of the simulation.

The code editor keeps the content of the editor area across iLand simulations, but does not execute any code automatically, so you'll have to do it manually (usually by selecting the lines and pressing `Ctrl+Enter` key combination).

6.3.2 When and how is JavaScript code executed?

Generally, JavaScript code is executed as soon as it is loaded. This means that public declarations and code (that is, code that is not part of a function or an object) are executed immediately.

Listing 6.1

```
// global variables are initialized immediately
var my_var = "global";

// public code is also run immediately
console.log(my_var); // this prints "global" to the log file / screen

// this can be problematic, because the model may not be fully available
// note also that the "management" interface object is only available when
// iLand management module is enabled.
var n_trees = management.loadAll();
console.log("global: total number of trees: " + n_trees);

// functions are created, but *not* executed!
function check_trees() {

  let n_trees = management.loadAll();
  console.log("function: total number of trees: " + n_trees);

}
```

In Listing 6.1 we have global variables and code, as well as functions.

However, in most cases your JavaScript code will be called (i.e., run or invoked) by the model. iLand provides several ways for your scripts to be engaged by iLand. The most important are described below.

6.3.2.1 JavaScript in forest management and BITE

The implementation of forest management in iLand relies heavily on JavaScript. There are two flavors of management available: "base" management and ABE, the agent based management approach. The "base" management module is simpler (and older) than the agent based management approach, but still versatile and allows a wide range of options for altering the forest in the model (extracting of trees, planting of trees, …).

In base management, a single javascript file is defined in the Project file and loaded during startup of the model. Every year the function `manage()` is called by iLand. The example shows a very simple and not particularly useful management routine:

```
function manage(year) {
  console.log('management called in year ' + year);

}
```

In case of ABE and BITE (a general module to simulate a wide range of biotic disturbances in forests) the integration with JavaScript is more powerful (but also more complex). Both modules rely on a declarative style (Section 6.3.4), and provide multiple ways to integrate your JavaScript code with objects of the iLand world provided via an API. See also the chapter on forest management for more details!

6.3.2.2 user interface

As mentioned previously, the user interface includes a simple "JavaScript code editor" that allows calling of JavaScript functions (Figure 6.1). In addition, the user can load JavaScript source from the iLand user interface.

6.3.2.3 time events

iLand provides a mechanism that updates the values of model settings automatically over time. Typical uses are to trigger pre-defined events such as storms, or change CO2 concentration over time. The same mechanism can also be used to run JavaScript functions at specific times.

6.3.2.4 event handler

An *event handler* in the context of iLand is a JavaScript function that is called when a specific event happens during the simulation. If no such function is found, then nothing happens. In other words: to "enable" such a handler, you just need to define a function with the correct name. The following event handlers are available in every iLand simulation:

Handler	Description
onAfterCreate	Called immediately before the first simulation year is executed.
onBeforeDestroy	Called after the simulation has ended and the model is shut down.
onYearBegin	Called at the beginning of each simulation year.
onYearEnd	Called after each simulated year.

```
// examples for event handler code
// this can be part of any javascript code

function onAfterCreate() {
  console.log("after create event handler called. Counting trees...");

  // call another function (this one was defined in the above listing  -
  // now is a good time to actually execute it,
  // as all trees are already present)
  check_trees();

}

function onYearEnd() {
  // the function is called every year, but it is simple
  // to limit actions to specific years:
  if (Globals.year == 100) {
    // do someting special in simulation year 100
    // e.g., create some very specific outputs

    // or just enable/disable existing outputs

    // or save a screenshot to a PNG file

  }
}
```

In addition, this mechanism is used in the fire module to allow some evaluation work after a fire happened.

6.3.3 Working with JavaScript in iLand

The user interface of iLand has rudimentary support for editing code and interacting with the model using JavaScript. While very useful on its own, iLand is not - and does not want to be - a full-fledged Integrated Development Environment (IDE) such as R Studio, or Visual Studio. If you are working on longer scripts, it can be a good approach to edit code using a good editor (Notepad++) or an IDE, and use iLand to inspect data and develop code snippets interactively.

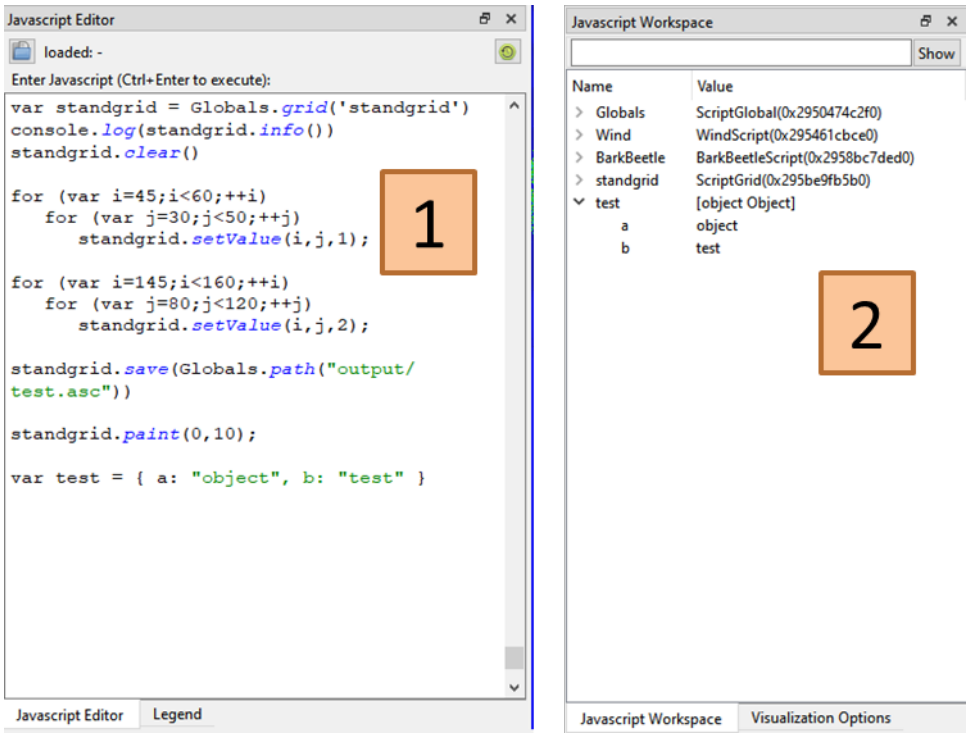

Figure 6.1: Code editor and JavaScript workspace window

Using the code editor is straightforward. You can enter / copy&paste code, and run code by selecting the code and pressing `Ctrl+Enter`. To run code in a single line, it suffices to have the cursor in the line and press `Ctrl+Enter`. iLand "remembers" the content of the code editor between restarts, which makes it easy to keep some useful code snippets around. The executed code as well as any log messages from the code are shown in the "log output window", even when iLand log output is written to a log file. If you are interested in the return value of executed code, you need to explicitly log (e.g., using `console.log()` or `Globals.alert()`),

or store to a variable for later use. Note that it is possible to "overwrite" already existing functions / objects and run them interactively (see example below) - this is a great way to quickly improve or debug code without having to restart the model multiple times.

The "workspace" window allows inspection of variables and objects that are currently available in the JavaScript engine as pairs of "name" and "value". The "name" is typically used to access the respective value. The following types of data are shown:

- The "value" of simple types (such as numbers or strings) is shown directly
- JavaScript objects (also nested objects) are shown as a tree of properties/functions, which can be expanded. Arrays of values or arrays of objects are also expandable. The "value" of objects is the generic label `object Objects`.
- Objects provided by iLand API are shown as well. In this case the value is the internal object type name (sometimes slightly cryptic) and the (hexadecimal) memory address of said object (the memory address can be useful to decide if multiple Javascript object refer to the *same* underlying object in iLand memory space). An example is `ABE::SchedulerObj(0x25d432a0)`.
- Javascript functions are shown as `function <functionname>() {[ native code ]}`.

Note that iLand hides some standard objects to avoid cluttering (Math, JSON, Atomics, Reflect, console).

The following example shows how you access (and modify) elements shown in the workspace window:

```
// simple values
console.log(filename);

// javascript objects

// iLand API
Globals.year = 23; // yields an error, as Globals.year is read-only
```

6.3.3.1 Output

Text output from JavaScript code is typically written to the log window (or the log file during simulations). A useful pattern that allows many details in the log during development, but avoids millions of log lines during large simulations, is the idea of "debug levels". A simple implementation in JavaScript with just two levels (on or off) could look like this:

```
// global switch that turns on or off detailed logging
var debug_logging = true;
// log function
function log(msg){
  if (debug_logging)
    console.log(msg);
}

// in user code:
var n = trees.count();
log(`there are ${n} trees in the list`);
```

You could have `debug_logging` set to `true` during development, or during debugging (just run `debug_logging=true;` in the editor to turn logging on), and switched off again later.

6.3.3.2 interactive development example

For example, consider the case when you want to create custom raster output files at certain years and have the code in a file that is automatically loaded (`system.javascript.filename`).

The code in Listing 6.2 shows:

- how you can create unique filenames by using the current year of a simulation and values from the project file (the section `user` is dedicated for that use, as it is not used by the model itself).
- how you can get gridded information from iLand, and can use that for simple "raster algebra" operations (see iLand API)

To test / improve your code, you could either run iLand multiple times (and wait until 100 years are simulated each time), or you do it interactively. You could:

- copy the code snippet (the `writeExtraOutput()` function in Listing 6.2) to the JavaScript editor window
- change code, and run the full function (select and `Ctrl+Enter`)
- in addition, you could run individual statements of the function (basically line by line). For example, if you run `var vol_grid = Globals.resourceUnitGrid("volume");`, you'll have also access to the `Grid` object, as `vol_grid` would the be a global variable. In that case, you can use the Javascript workspace window and inspect the properties of the grid (such as resolution, size, or name). Note, that in case of grids you can also just plot them (`grid.paint(min_value, max_value)`)!

Listing 6.2

```
// content of "extra_output.js"
function onYearEnd() {
  if (Globals.year == 100) {
    writeExtraOutput();
  }
}

function writeExtraOutput() {
  // construct a unique name for the output file by using a user-defined
  // scenario and the current year
  var filename = "output/extra_grid_" +
    Globals.setting("user.code") + "_" +  Globals.year + ".asc";
  var vol_grid = Globals.resourceUnitGrid("basalArea");
  // basal area m2 of Norway spruce
  var species_grid = Globals.speciesShareGrid("piab");
  // calculate relative basal area for norway spruce
  vol_grid.combine('bP/bT', {bP: species_grid, bT: vol_grid} );
  vol_grid.save(filename); // and save to file

}
```

- copy back working code to your JavaScript source file and run final tests with a full iLand simulation.

6.3.4 Declarative style and JavaScript objects

In JavaScript, a declarative style is an approach to programming where you focus on describing "what" you want to achieve rather than explicitly specifying "how" to achieve it. Declarative programming is often contrasted with imperative programming, where you explicitly outline step-by-step instructions for the computer to follow. Major key points of declarative programming are:

Descriptive code style
Declarative code is more descriptive and expresses the desired outcome in a clear and concise manner. Instead of detailing the steps to perform a task, you declare what you want to accomplish. For example, you describe a minimum DBH threshold for a thinning, but not the rules how to enforce that.

Level of abstraction
Declarative programming involves the use of abstractions that hide the low-level implementation details. This allows you to work at a higher level of abstraction, focusing on the overall structure and logic.

iLand uses such a declarative style mainly for ABE and BITE as it is a very flexible mechanism that allows mixing functionality provided by the model with user logic written in JavaScript (that may use again the iLand API to interact with the model). For instance, you define a management activity in ABE and describe certain properties of that activity:

- When should the activity be executed?
- Are there any conditions (e.g., minimum diameter) that need to be met?
- What should happen, when the activity is run?

Particularly for the "what should happen" part it can be helpful to be able to control programmatically in a very detailed and descriptive way what exactly should happen. iLand heavily uses "declarative objects" for this task.

Declarative objects can contain different types of data, including simple values, arrays, and functions. JavaScript object literal syntax is a concise way to create such declarative objects. Objects in JavaScript are collections of key-value pairs, where each key is a string (or a Symbol) and each value can be of any data type, including other objects. The object literal syntax provides a clean and readable way to define objects directly in your code.

Here's the basic structure of an object using object literal syntax:

```
const myObject = {
  key1: value1,
  key2: value2,
  // ...
};
```

- **myObject**: The name of the variable holding the object.
- **key1, key2**: The keys of the object, usually strings, but they can also be Symbols (introduced in ES6).
- **value1, value2**: The corresponding values associated with the keys. Values can be of any data type, including numbers, strings, arrays, functions, and even other objects.

Example

```
const person = {
  name: "John Doe",
  age: 30,
  city: "Example City",
```

```
  hobbies: ["reading", "coding", "traveling"],
  greet: function() {
    console.log(`Hello, my name is ${this.name}!`);
  },
};

console.log(person.name);           // Output: John Doe
console.log(person.hobbies[1]);     // Output: coding
person.greet();                     // Output: Hello, my name is John Doe!
```

6.3.5 Related JavaScript topics

6.3.5.1 Concatenating strings

Building complex strings is a very common requirement in iLand scripts. Typical use cases are to build filenames, e.g., to include some scenario label and simulation year to a raster output, or to build "expressions". Expressions are used in many places in iLand, e.g., to create a filter expression to select trees with specific properties in a tree list.

Javascript supports a variety of ways to concatenate strings (the web is full of useful guides); However, below are some simple examples:

```
// classic use of the + operator
var filename = Globals.path("output/species_grid_" + Globals.year + ".asc");

// if substrings are in an array, you can use the "join()" method
var filename = ["output/species_grid", Globals.year, ".asc"].join('_')

// string literals are a newish but very nice way to do it
// (https://developer.mozilla.org/en-US/docs/Web/
//  JavaScript/Reference/Template_literals)
// filter trees in a circle around a point p with a radius 10:

const p={x: 10, y=44}; const maxr=10;
const filter_str = `(x-${p.x})^2 + (y-${p.y})^2 <= ${maxr}`;
// produces: -> "(x-10)^2 + (y-44)^2 <= 10"
stand.trees.filter(filter_str);
```

6.3.5.2 Exceptions and error handling

Handling errors is an important aspect of developing any type of code - this is also relevant for JavaScript in iLand. There are various approaches how to handle error conditions. The most important are:

Creating log messages
This is clearly useful, particularly during development. Informative log messages point you to specific error conditions and thus help to improve the code. The downside, however, is that errors can easily get buried in the log file, and it can get hard to (a) notice that something went wrong, and (b) to find out what went wrong. A good strategy in such a case is to scan the log file particularly for strings used in your log messages (strings including "error" are particularly easy to find).

Raise exceptions to stop a simulation
Exceptions are errors that (usually) halt the execution of the simulation and show a popup-window informing about the error. (Note that behavior is different in `ilandc`). (TODO: check JS exceptions in ilandc!!!). Exceptions are raised either by iLand (e.g., when you access a setting in the XML project file that does not exist), or you can also create your own exceptions/ error messages in JavaScript user code. Moreover, you can *catch* exceptions in JavaScript code and handle the error yourself - in that case the exception does not leave JavaScript world and thus the simulation does not stop.

Below are two examples of using exceptions. More on exceptions in Javascript can be found in this this general introduction.

6.3.5.2.1 Example 1 - raise an exception from user code

This example is taken from an ABE management script:

```
# Example 1:
# Raise an explicit error

function getNextPoint(stand_id) {
    // search for a point in a pre-loaded table
    const index = pfile.findIndex(element => element.stand_id === stand_id &&
    element.id >= 0);
    // It should not happen that no point is left, so raise an error:
    if (index == -1)
        throw new Error('no valid point left!');
    ....
}
```

When the error occurs, a message pops up:

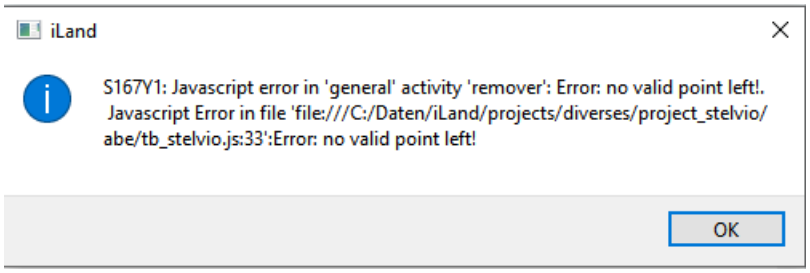

Figure 6.2: Example error message for an exception thrown from JavaScript

6.3.5.2.2 Example 2 - Handling errors raised by iLand

The following examples uses JavaScript to catch exceptions from iLand.

```
// helper function to check if a given setting
// exists in the XML project file
function settingExists(key) {
    try {
    let x = Globals.setting(key);

    } catch (error) {
        console.log(error);
        return false;
    }

    // the setting exists
    return true
}

// use of the function

if (settingExists("user.my_user_setting.a")) {
   // ....
   // execute this code only when a specific
   // setting can be found in the project file
}
```

6.3.5.3 Reading and writing files

A re-occuring requirement is the need to get auxiliary data into the model to be used with JavaScript code. In many cases you need to read input data from text files, either as tables (think CSV-delimited data files) or just pure text. Similarly, writing text files is often needed, e.g., to create and store extra output data or extra log files. iLand provides a couple of mechanisms that can be use for file input/output and some are described in the following.

Reading and writing simple text files

The `Globals` object provides the function `loadTextFile()` and `saveTextFile()` and other useful functions when working with files (e.g., `path()`, or `fileExists()`. See the Globals object API documentation.

Reading (and writing) tabular data

To read tables from text files, you can use the CSVFile object. In a nutshell, a `CSVFile` object reads and parses a text table (auto-detecting the delimiters) and provides function to access the values of particular rows and column. Note that `CSVFile` lets you also *change* the values of cells and save the changed table to file!

You can work either directly with the API from `CSVFile`, or convert the content of the data file into a more convenient form, e.g., as a JavaScript object. The following code shows a way to do it:

```
// load a text-file based data table and convert to an JS object
function loadFromFile(filename) {
    var res = [];
    // load CSV File
    var csv = new CSVFile(Globals.path(filename));
    // convert each row to an object with column names as property names
    for (var r=0;r<csv.rowCount;++r) {
        var obj = {};
        for (var c=0;c<csv.colCount;++c) {
            obj[csv.columnName(c)] = csv.jsValue(r,c);
        }
        res.push(obj);
    }
    return res;
}
```

The resulting data structure is an array with an entry for each line of the data table; each entry is a JavaScript object with column-names as key-names of the objects. Consider the following table `data.csv`:

Year	standId	species	proportion	...
5	1	abal	0.5	...
5	3	piab	0.2	...
9	3	piab	0.4	...

After reading (e.g., `data = loadFromFile('data.csv')`), you can access the content of the table, e.g., `data[0].standId` would be 1. Note that such a structure is very convenient for manipulation (e.g., filtering) with JavaScript. For example:

```
// filter all elements of the current year (here: 5)

const current = data.filter(item => item.year === 5);

// loop over the elements
for (const elem of current) {
  processStand(elem);
}

// the 'obj' has the columns of the CSV file as properties:
function processStand(obj) {
  console.log(`Process stand ${obj.standId} and species ${obj.species}!`);
}
```

Creating a log-file

Sometimes you want to create a "log-file" on your own to save some information related to your processing. For example, the stand level processing of the previous example could log information on how it all went. A simple logger-class for such a use case could look like this:

```
// a simple logger class
// to log information for each year and stand
function Logger(logfile) {
    this.path=logfile;
    this.text = "year;stand;text"; // the lines
    this.save = function() {  Globals.saveTextFile(Globals.path(this.path),
    this.text);  }
    this.log = function(id, txt) { var line=Globals.year + ";" +
    id + ";" + txt;
                                this.text = this.text + "\n" + line; }
}
```

The `Logger` uses some internal formatting logic and saves the log file using the `saveTextFile()` function discussed above. Here is how to use it:

```
// create an instance of the logger, e.g. as global variable
var plog = new Logger(Globals.path('temp/my_processing.log'));

function processStand(obj) {
 // ...
 plog.log(obj.standId, `Processing started.
 Try to remove ${obj.proportion}.` );
 // ...
}

// Note that you need to call the save() function to actually store on disk.
// this could be done when all processing is done for a year
....
plog.save();
```

7 Forest management

Forest management is a key component of a forest model, particularly when used in areas where forests are strongly affected by human interventions.

iLand includes two sub-systems that handle forest management:

- the "base" management is a rather simple, but still versatile approach which allows a wide range of altering the forest in the model (extracting of trees, planting of trees, …).
- The Agent Based management Engine (ABE) is a newer variant. ABE is a powerful module that allows to simulate multiple management agents on a landscape. Agents can even react to changes in the environment by changing management plans and can apply a variety of different management activities in a dynamically scheduled manner. The setup and application of ABE, however, is somewhat complex.

The ABE model provides a framework to dynamically simulate adaptive forest management in multi-agent landscapes under changing environmental conditions. While the Rammer, W., Seidl, R., 2015 paper desribes ABE scientifically, this chapter provides more detail on both the concepts and the technical implementation.

From a technical point of view, forest management in iLand uses the integration of JavaScript with the model. Specifically, ABE uses a declarative programming paradigm. The section Section 6.3.4 provides more details.

7.1 Components of ABE

Agent and agent types
In ABE, all forest managers are represented by Agents. *Agents* are responsible for the management of a specific part of the landscape. Each Agent is of a certain *Agent Type*. An *Agent Type* corresponds to a manager archetype such as "farmer" or "forest company", while an *Agent* represents an individual manager with specific properties, e.g., the agents' age. The AgentType is created implicitly, when just an agent is created.

Stand treatment program (STP)
A *stand treatment program* consists basically of a collection of activity objects, which typically cover the silvicultural treatments to regenerate, tend and thin, and harvest a stand. In

addition, the *STP* includes a definition of a time window for each activity, from which ABE autonomously derives the sequence and preliminary time of execution for each activity.

Management activities
Activities are the core element of forest management in ABE, as they are the elements that actually change the forest state in the ecosystem model by removing or planting trees. ABE provides a library of pre-defined activities that cover the most important aspects of forest management. Yet, the activity library can be easily extended by providing user-defined activities. Activities are fetched from the library by specifying the name of the activity type and by further defining the properties of the activity.

7.1.1 Defining a minimum ABE script

```
var stp = {   U: 100,
              thinning1: { /* defined below */ },
              thinning2: {/* defined below */ },
              clearcut: { /* defined below */ }
}
fmengine.addManagement(stp, "program");
```

Note that if you do not define a `unit` and an `agent`, a default unit (with the name `_unit` and default agent (`_agent`) are automatically created by iLand.

7.1.2 Accessing elements of ABE

There are two central objects for accessing ABE, namely `fmengine` and `stand`. On the one hand, `fmengine` provides access to global properties of ABE, such as a list of all stands in the system. `stand`, on the other hand, allows to access properties of the stand that is *currently* processed. Linked to a `stand` are then the management `unit` , the stand treatment program (`stp`) or the current activity that is executed. For "normal" management activities you can just use `stand`, as ABE takes care of everything. You can, however, also manually override the current stand that ABE processes - this can be useful for debugging or advanced scripting tasks. To set the "focus" of ABE to a specific stand, you just have to set `fmengine.standId` to the numeric Id of this stand.

7.2 Inspection

7.2.1 Loop over all stands in a simulation

The property `fmengine.standIds` lets you access all stands that are set up with ABE (i.e., stands for which at least basic information is available in the file referenced by `abe.agentDataFile`) . In order to access objects related to this stand, you need to set `fmengine.standId` to the respective stand:

```
// print id and name of STP for each stand:
for (const id of fmengine.standIds) {
  // set the focus of ABE to this stand:
  fmengine.standId = id;
  // access data for the stand
  console.log('Stand ' + id + ' managed by ' + stand.stp.name );
}
```

7.2.2 Check all available STPs

A list of all STPs currently available can be queried via `fmengine.stpNames`. Note that since the names of STPs are unique, you can easily loop over all STPs:

```
// Show details for each STP.
for (var i=0; i<fmengine.stpNames.length; ++i) {
  let s = fmengine.stpNames[i];
  // Show a pop-up window with a detailed report for each STP
  Globals.alert(fmengine.stpByName(s).info);
}
```

7.2.3 Assign a new STP to a stand

It's always possible to assign a new STP to a stand or overwrite an already existing STP:

```
// create STPs, details omitted
fmengine.addManagement({ ... }, 'bau'); // default
fmengine.addManagement({ ... }, 'am1');

fmengine.standId = 1;

console.log(stand.stp.name);  // -> "bau"
```

```
// set to a new STP identified by the name
stand.setSTP('am1');
console.log(stand.stp.name);  // -> "am1"
```

7.2.4 Dynamic update of a STP program

This is the case, when a STP changes during a simulation, i.e., when the definition of a STP changes *dynamically*. Note, that usually dynamic behavior is implemented within a *static* STP using conditional statements or similar. The spinup of iLand is an example that is changing STPs iteratively to improve the match of simulated with target forest state within a longer spinup simulation.

Caution

Updating a STP while it is currently executed is not possible! A workaround solution could look like this:

- the STP that realizes that a new STP should be generated stores that information to a list (e.g. `stand_to_process.push_back(stand.id)`
- Using iLand events you can trigger the creation of new STPs; for example, use a `onYearEnd` event:

```
function onYearEnd() {
   while (stands_to_process.length>0) {
        var stand_id = stands_to_process.pop();
        console.log("*** processing stand" + stand_id );
        // create a STP programmatically. Remember the definition
        // of a STP is just a collection of objects (representing
        // activities)...
        const stp = createNewStp( stand_id );
        // update the STP for the stand (use unique STP names here)
        fmengine.updateManagement(stp, "mgmt_"+stand_id);
    }
    console.log('**** update stands finished *****');
}
```

7.3 Working with activities

Activities are the main building blocks of forest management in ABE. A detailed description is on the wiki: https://iland-model.org/ABE+activities

Here we cover some practical aspects of working with activities.

7.3.1 Defining activities

Activities are JavaScript objects, that (can) include also code. Therefore there are several ways how activities can be created; a straightforward is to create an explicit object:

```
// (1): explicit object

// a simple activity that runs when the stand is 10 years old
// and does not do much
const my_activity = {
   type: 'general', schedule: 10,
   action: function() { fmengine.log('Activity executed!')}

}
```

You can later use `my_activity` as part of the definition of a STP. A more advanced approach facilitates the fact that functions can *generate* activities and uses JavaScript *closures*:

```
// (2) An activity generator
function createActivity(max_dbh, age) {
   // the function is "hidden" here
   // and can encapsulate complex behavior
   // note that for instance 'max_dbh'
   // is still visbile in do_x() - this is
   // a closure.
   function do_x() {
      //
      fmengine.log(max_dbh);
   }
   return {
     type: 'general', schedule: age,
     action: function() { do_x(); }
   }
}
```

```
const my_act = createActivity(20, 50);
```

Note that in many instances the simple approach is sufficing. Note further, that usually an activity can react to the specifics of a stand (see next section) without the need to create *new* activities for every stand.

7.3.2 Stand variables and stand flags

Activities are executed for a stand, and stand-specific information is available typically via the `stand` object. In this section, we introduce some patterns how this can be done in practice. Look at the following example that shows the `action` function of a (general) activity.

```
...
action: function() {
  // access properties of the stand, for instance volume
  // see also: https://iland-model.org/apidoc/classes/Stand.html
  if (stand.volume > 300 ) {
     // do something only if standing volume / ha is high enough
     const target_volume = stand.flag('target_volume');
     // ... use target volume
  }
}
...
```

Some statistics on the current state of the stand are available via the `stand` object, for example `stand.volume` is the standing volume (m3/ha) (see https://iland-model.org/apidoc/classes/Stand.html for more variables). More interesting is the `stand.flag()` in the example. Flags are stored as key-value pairs per stand; different stands can have different keys (or also the same keys) and can hold any (JavaScript) value, including JavaScript options. In the following, we discuss some example use of flags.

Example 1: Memory

Flags can be used as "memory". For instance, a counter (per stand) for a repeating activity can be implemented via a flag, or flags can indicate that certain events happened already in the past (for example, a flag could indicate that a planting happened)

```
// counter example
{ type: 'general', schedule: {repeat: true, repeatInterval: 5 },
  action: function() {
            let event_happened = ....; // logic that evaluates event
            if (event_happened) {
```

```
            let cnt = stand.flag('my_counter');
            if (cnt > 3) {
                // the "event" happened already three times: time to
                // do something specific!
            }
            stand.setFlag('my_counter', cnt + 1);
        }


    },
  onSetup: function() {
      // the onSetup event is a good place to initialize the counter
      stand.setFlag('my_counter', 1);
      }
  }
```

Example 2: Pass information

Flags can pass information between different activities of a STP. Think of a situation where an (earlier) activity makes a decision e.g. which species to promote, and a later activity needs to know the species to further protect.

```
//  activity A (runs earlier)
...
action: function() {
    // decide which species to promote, for example
    const promoted = ['acps', 'frex']; // an array!
    stand.setFlag('promoted', promoted);
} ...

//  activity B (runs later)
...
action: function() {
    // get list of promoted species
    let p = stand.flag('promoted');
    // convert to syntax used by expression
    let save_string = `in(species, ${p.join(', ')})=false`;
    // will be: "in(species, acps, frex)=false"
    // do *not* load the promoted trees
    trees.load(save_string);
}
```

Example 3: External properties

Flags can be used to provide extra information to ABE; for example, you could provide a flag providing the distance to the next forest road - which could influence the actual management! While you can always programmatically set flags, e.g., read a table, set flags for each record (see Section 6.3.5.3), iLand provides a much simpler way, namely the ABE data file (`agentDataFile`). iLand reads data from some columns (e.g., `id`, `stp`, see https://iland-model.org/ABE+spatial+setup). If the file contains additional columns, then the values are automatically assigned to each stand as stand flags! See the following part of the `agentDataFile`, that defines "target species" for each stand (Note that you could also define two different STPs here!).

id	stp	species	...
1	bau	abal, fasy	...
2	bau	acps, frex	...
...	...	...	...

Now it is straight-forward to use this information:

```
...
action: function() {
    // read the value from the 'species' column
    let target_species = stand.flag('species');
    // let's assume we want to run over something
    // for each species in the list of species
    for (const s of target_species.split(',')) {
       fmengine.log(`processing species ${s}...`);
       // do something...
    }

}
```

7.4 Advanced topics

7.4.1 Working with patches

From a spatial perspective, ABE typically operates on stands, i.e., (pixelated) polygons which represent forest stands. Such stands are usually relatively homogeneous in composition and structure and are usually also the typical spatial entity of real-world forest management. As a user, you provide the logic how each single stand should be treated, and that logic is then applied to (potentially) multiple stands, one at a time. In many cases the management on a stand applies for the whole area of the stand - a clear-cut removes all trees from a stand

just as a wall-to-wall planting affects the full area. The `TreeList` and `SaplingList` object of iLand are perfectly suited for such tasks. Not all management activities work that way - there is increasing interest and real-world application of more spatially explicit forest management. Such activities create gaps of different sizes to foster regeneration, harvest stands as sequence of strip cuts, or respond to disturbance by planting target species in crown openings created by disturbance. The `patches` feature of iLand is designed to aid the implementation of such spatially explicit activities.

A `patch` describes an area within a stand, and is basically a list of pixels on the stand grid (i.e., with 10m resolution). The pixels of a patch are usually, but not necessarily, adjacent, e.g., to form a gap in the forest. Each `patch` is identified by a numeric integer ID and an `area`, and a `score` (see below).

The `patches` property of a `stand` holds a collection of 0 or more `patches`, (the `list`-property) and provides a number of functions to create or modify `patches`. Internally, the object maintains a map of the stand and each 10m cell is either linked to none or exactly one `patch`. However, 10x10m pixels can be part of multiple patches at the same time!

7.4.1.1 Using patches to manipulate trees and saplings

Both trees and saplings provide a variable `patch` which can be used in every expression for filtering, loading etc. of tree- and sapling-lists. The value of `patch` is -1 if the entity is not located on a patch (or if there are no patches on a stand).

Listing 7.1 Code example for a clear cut on certain patches defined in ABE

```
// within ABE code:
// fill the trees list with trees standing on patches 11,12,...
stand.trees.load('patch > 10');

function clearPatch(patch_id) {
  stand.trees.load('patch=' + patch_id);
  stand.trees.kill();
  // use a SaplingList defined elsewere
  // (e.g. using var saplist = new SaplingList)
  saplist.load('patch=' + patch_id);
  saplist.kill();
}
 clearPatch(4); // kill all trees and saps on that patch
```

Note, that you can use the `patch` property also outside of ABE, e.g., to visualize patches in the iLand user interface!

Patches can also be used to specify locations the as target area for planting activities in ABE. To do this, set the `patches` property of the planting activity. The property is either an expression that is evaluated for every 10m pixel of the stand (use the `patch` variable to indicate the patch), or a Javascript array of patch-Ids.

Listing 7.2 Example how to implement a planting activity on certain patches with ABE

```
// run a planting manually, i.e. from within some other ABE code:
// plant on all defined patches (i.e, with values > -1)
fmengine.runPlanting(stand.id, {
   species: "abal", fraction: 0.4, height: 0.3, clear: true,
   patches: 'patch>-1' });

// Example for a regular planting activity within the definition
// use a Javascript array to specify the patches
**** not implemented :( *****
```

7.4.1.2 Creating patches

You can create patches using the functions provided by the `patches` object. It provides functions to create (a large number of) regularly spaced patches or to split a stand into several strips, e.g., for strip-cutting. Moreover, there are functions that allow to copy patches from another grid - this could be used for pre-defined patterns, but also to react to disturbances. Typically, those functions return multiple patches. Note that you need to explicitly update the `list` variable in order to effectively use the patches!

See the section on Patches in the iLand documentation and the examples below for details.

7.4.1.3 Manipulating lists of patches

The list of individual `patch` objects (each representing a single patch) can be accessed and manipulated from JavaScript via the `stand.patches` property in the context of ABE. Thus, you can apply advanced JavaScript logic to select / remove patches! For an overview of available options see here.

Note that it is not possible to change individual elements of the `list`, instead you have to set the `list` to a new set of `patch` objects! After changing the list, the internal map is updated and thus the new list is effectively used for subsequent use in expressions.

The following example is more complex and outlines the steps required to find "optimal" patches for specific use. The steps are:

Listing 7.3 Simple example: how to work with the patches list

```
// Example 1: just create patches.
// Note that stand.patches.list is set explictly.
stand.patches.list = stand.patches.createStrips(20, /*horizontal=*/ false);

// now use JavaScript to loop over the list, here use the forEach() function
stand.patches.list.forEach( (p) => console.log(p.id + " area: " + p.area) );

// the same thing, with another looping construct
for (p of stand.patches.list) {
  console.log(p.id + " area: " + p.area) );
}

// ....
// now use the patches, e.g. for harvesting in an activity
//
stand.trees.load('patch=' + current_strip);
stand.trees.harvest();
current_strip = current_strip + 1;
```

- create a large number of candidate patches
- evaluate each candidate patch (based on vegetation on the patch) and calculate a score based on some user-defined metric
- select the “best” patches

Listing 7.4 Advanced example on how to work with patches

```
// Example 2: complex manipulation of patches

// Let us start with the top level view:
function findPatches() {
    // (1) create patches, here just strips
    stand.patches.list = stand.patches.createStrips(10,/*horiziontal=*/false);

    // (2) run the evaluation function for every patch
    stand.patches.list.forEach((p) => patchEvaluation(p));

    // (3) select patches based on the score provided in the evaluation
    // function and *update* the patch list by setting the list property.
    stand.patches.list = topN(5);
}

// Ok, let us look at the evaluation function that is called for each patch
// evaluate trees and saplings and calculate a score for the patch
// The score is high on patches with only few trees, but abundant regeneration
function patchEvaluation(patch) {
    var score = 0;
    stand.trees.load('patch = ' + patch.id);
    let n_adult = stand.trees.count / patch.area;

    // calc number of (represented) saplings
    saplist.loadFromStand(stand.id, 'patch = ' + patch.id);
    let n_target = saplist.sum('min(nrep,10)', 'in(species, fasy, abal)') /
    patch.area;

    console.log(`N-Adult: ${n_adult}, n-saps: ${saplist.count} n-good-saps:
    ${n_target}`);

    // insert fancy scoring logic here :)
    score = Math.max(0, n_target - n_adult); //
    patch.score = score;
}

// The last part is selecting a subset and actually *modifying* the patches
// of the stand select the N patches with top scores
function topN(n) {
    // Note that sorting *directly* within stand.patches.list does not work.
    // instead use an extra list:
    var slist = stand.patches.list;

    // sort score (descending) - this uses the standard sort() function
    // of JavaScript arrays
    slist.sort( (a,b) => b.score - a.score);

    // reduce to the N patches with top score (again, slice() is a)
    slist = slist.slice(0, n);
    return slist;
}
```

8 Setting up landscapes

Initialization of a simulation landscape in iLand entails four core components:

- Landscape and forest extent (Section 8.1)
- Geospatial data on biophysical drivers (Section 8.2)
- Initial vegetation conditions (Section 8.3)
- Information about disturbance regimes and/or management (Section 8.4)

Although initialization typically refers to simulating landscapes that represent real locations, iLand is also used to simulate individual stands (Hansen et al. 2020; Kobayashi et al. 2023) or generic, hypothetical landscapes (Braziunas et al. 2018) depending on the research questions.

8.1 Landscape extent

Landscape extents often align with jurisdictional boundaries, natural barriers to seed dispersal, or disturbances such as ridge tops or large water bodies, or the edges of an entire forest patch. Once boundaries are defined, forested or potentially forested (i.e., stockable) area must be further delineated within the landscape, because iLand is specifically designed to simulate forest dynamics and doesn't explicitly represent vegetation types that would prevent forest expansion into non-forested areas. Stockable area in iLand landscapes typically ranges in the 1,000s to 10,000s of ha, although the landscape extent may be much larger if there are extensive non-forested areas.

This chapter explains how to set up a "real" landscape. Information on other types of simulation set-ups (e.g., for growth evaluation tests) can be found here.

8.1.1 Stand grid

The stand grid is the definition of the project area as a GIS grid (10m resolution). It is often derived from polygonal data, e. g., a map of stand polygons. The stand grid defines the forested area of the project landscape and the spatial distribution of forest stands within it at the beginning of the simulation. Stand delineation is often based on the forest management

history (stands as management units) or forest type maps (stands as forest patches with similar composition and structure). A *stand* is defined by an integer ID greater than zero. Cells with the same grid value belong to the same "stand". In addition to the forested area within the project area, the grid contains information about the area surrounding the forest:

- Pixel values of *-1* (or the NO_DATA_VALUE of the grid) = "non-forested outside pixel": non-project-area, no regeneration establishment possible
- Pixel values of *-2* = "forested outside pixel": continuous "shading" from the assumed forest outside of the project area is applied (LIF calculation), can act as source of external seeds, included in fetch calculations of the wind module

The stand grid can also be used to initialize the tree vegetation of each stand (see Initializing with forest inventory data).

8.2 Biophysical landscape

The next step after defining the size and the boundaries of the project area is to define the actual content, i.e. the distribution of biophysical drivers like topography, daily climate data, and soil properties atop of the landscape. Their distribution is initialized in iLand via the environment file and grid, which define site and climate conditions:

- The **environment grid** contains regions with homogeneous soil and climate conditions with a resolution of 100m (these cells are henceforth referred to as **resource units**). The grid itself holds an (integer) "id" for these units
- The **environment file** is a table (CSV or other text format) containing for each "id" (i.e., resource unit) the values for soil and climate conditions (and also things like carbon pools for soil and deadwood)

When iLand initializes a resource unit, it uses the values provided in the environment file. The environment file can also be used to establish a link to climate data (see Climate). Each row contains the information about one resource unit:

Key (column name in the environment file)	**Description**
id	resource unit id corresponding to the environment grid (grid mode)
x	0-based index of the resource unit in x-direction (matrix mode)
y	0-based index of the resource unit in y-direction (matrix mode)
model.site.availableNitrogen	available nitrogen of the unit (kg/ha/yr)

Key (column name in the environment file)	Description
model.site.soilDepth	depth of soil (cm) of the unit
model.site.pctSand, model.site.pctSilt, model.site.pctClay (one column each)	soil texture [%], cell-wise sum = 100
model.climate.tableName	climate data for that unit (table name in the .sqlite-file)

If the environment file **does not** contain a specific key, then the value provided in the project file is used throughout the landscape. The keys refer to nodes in the project file. More keys can be added if needed (e.g., keys for Dynamic carbon cycling or Fire).

8.2.1 Topography

Topographic drivers used in iLand are derived from digital elevation models (DEM), which are often available at high resolution across the globe (e.g., the US National Elevation Dataset, Germany Digital Terrain Models). The resolution of the input file may vary, and iLand internally creates a DEM with a resolution of 10 m and calculates maps of slope and aspect. The DEM is optional and doesn't affect simulation outcomes. When no DEM is given, a flat topography is assumed.

8.2.2 Climate

Daily climate data for the time period of interest (e.g., historical, future) and required variables (global radiation, minimum and maximum temperature, precipitation, vapor pressure deficit) is often available at regional, continental, or global scales, although spatial resolution and availability of specific future climate models may vary. Additional steps can be taken to downscale daily climate to the resolution of resource units in iLand (Thom et al. 2022) or generate daily time series from coarser temporal resolutions. Annual atmospheric CO_2 concentration can also be incorporated as a climate driver.

Data about daily climate is stored in SQLite-databases which contain one table per climate cluster. Climate clusters are regions of the landscape with similar climatic conditions and are independent of the stand grid. Each resource unit is assigned to one climate cluster (table in the climate-SQLite). The link between resource unit and climate table is established in the environment file (key "model.climate.tableName", see Biophysical landscape). Climate scenarios ("historic", "RCP85" etc.) are stored in separate .sqlite-files.

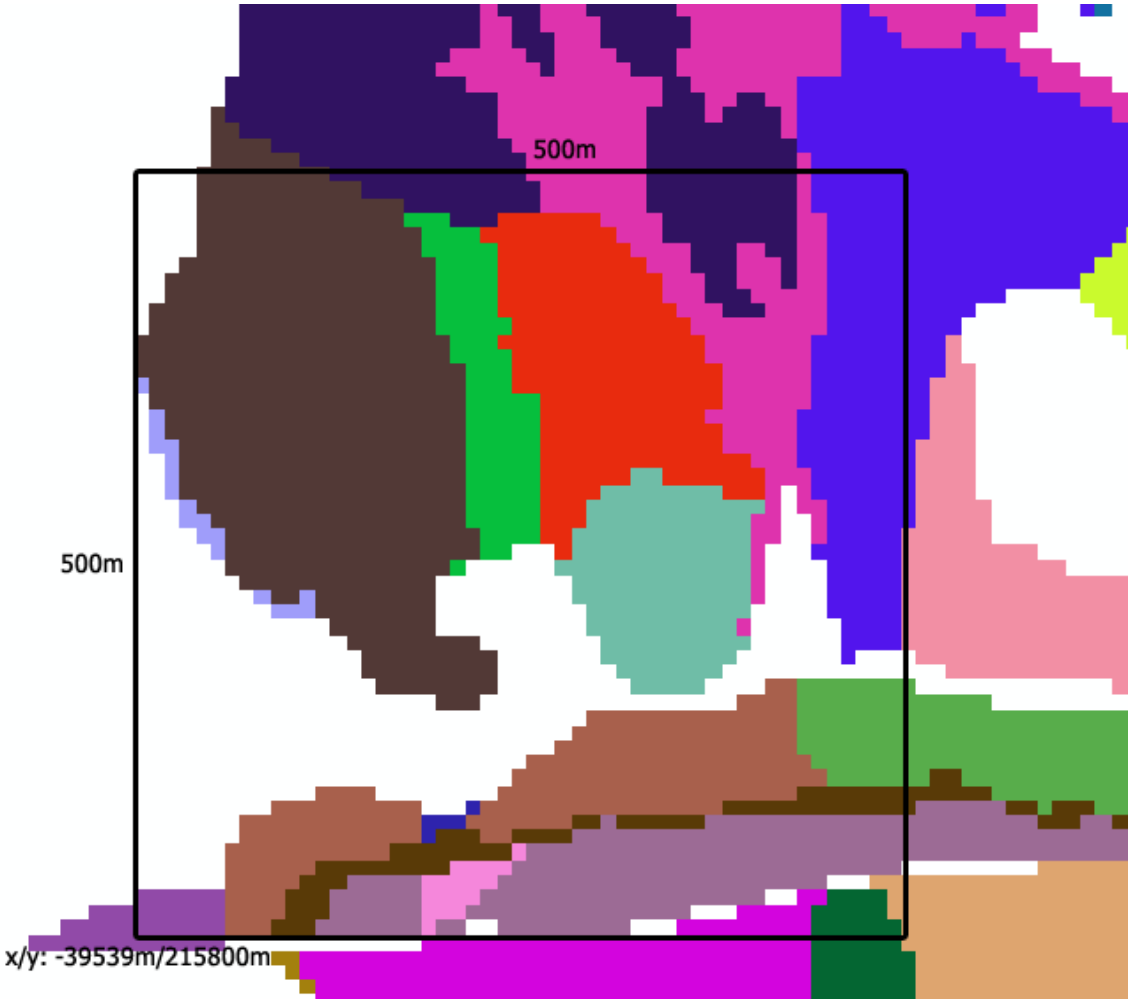

Figure 8.1: The screenshot taken from a GIS application shows the target landscape. The shown grid is derived from stand polygons and has a cell size of 10m. Cells belonging to the same stand have the same color. Blank pixels are not within the project area.

8.2.2.1 Downscaling climate data

Since future climate models are most commonly available only at resolution coarser than 100 m, methods for downscaling climate data have been developed. This is especially important in topographically complex landscapes. The method leverages the relationship between climatic variables (especially temperature and precipitation) and elevation. The lapse rate (the rate at which a climate variable changes with elevation) is calculated for each climate variable and then used for downscaling each 100x100 m-cell's climate parameters based on the cell's elevation. The method has been successfully applied in Thom et al. (2022) . The pipeline for the method is available on request from Christina Dollinger.

8.2.3 Soil properties

Soil properties including soil depth, texture, plant available nitrogen, and, if the carbon cycle is enabled, carbon pools tend to be more challenging to obtain. Spatially explicit maps of soil properties may be available only at local to regional scales or may not exist. A categorical soil type map is often used in combination with ancillary data such as soil samples, forest type, or forest productivity to estimate and assign average soil properties. Global soil databases are becoming increasingly available (Poggio et al. 2021), but may not include all variables and vary in accuracy across regions. If soil carbon or nitrogen cycling is enabled, a spin-up process can be used to derive initial pools (Hansen et al. 2020).

The minimal setup of soil properties includes spatially explicit data on soil texture (the percentage of sand/silt/clay), effective soil depth (cm), and available nitrogen (kg/ha/year) for the landscape. The soil properties of each resource unit are specified in the environment file (keys "model.site.pctSand" / "model.site.pctSilt" / "model.site.pctClay" / "model.site.soilDepth" / "model.site.availableNitrogen", see Biophysical landscape).

8.2.3.1 Dynamic soil carbon cycling

iLand uses the ICBM/2N model (Kätterer and Andrén 2001) to model the C in dead organic matter and soil pools. The technical implementation follows these steps:

1. Obtain initial values for the carbon pools: spatially explicit data about coarse woody debris carbon pools ("model.site.youngRefractoryC" in the environment file), litter carbon pools ("model.site.youngLabileC") and soil organic matter carbon pools ("model.site.somC")
2. Dynamically calculate the influx rates of dead organic matter:
    1. Simulate the landscape for 200 years, with carbon pools initialized with the values stated above

2. Extract the annual fluxes of dead organic matter simulated by iLand for each 100x100 m-cell in the last 50 years of simulation from the output "soilinput" (input_lab and input_ref and the aboveground fractions input_lab_ag, input_ref_ag)
3. Calculate mean decomposition rate of litter and coarse woody debris over the whole landscape (species parameter "snagKyl" and "snagKyr" in the species table)
4. Calculate the decomposition rate of soil organic material for each resource unit using the humification rate (either based on literature or estimated by running a sensitivity analysis)

More information on the description of the C model can be found here and on the parametrization process here.

8.3 Vegetation

Initial vegetation consists of species- and size-specific data on live individual trees and regeneration cohorts. Dead vegetation (e.g., snags, downed wood) can also be initialized. Forest inventory plots in combination with spatially continuous data such as canopy height from LiDAR or interpolation or imputation methods (e.g., FIA plot imputation by Riley, Grenfell, and Finney (2016)) can enable detailed wall-to-wall representations of current forest structure and composition. Alternatively, approximate forest conditions can be generated using a customizable spin-up process. The spin-up process ranges from potential natural vegetation, in which simulations start from bare ground with seed inputs from the landscape edge and run for 100s to 1,000s of years (Albrich, Rammer, and Seidl 2020), to a directed spin-up, in which target values such as stand age, tree size distributions, and composition are set and simulations are run iteratively until the realized conditions are close to expected values (Dobor et al. 2020).

8.3.1 Initializing with forest inventory data

Forest inventory data or data from independent field work can be used to initialize vegetation. For each stand (see Stand grid) information about stem density, tree species and DBH distributions is needed. This data is compiled in an "initialization file" (node *model.initialization.file* in the project file). Each line in the input file describes a number of trees (*count*) of a certain *species*. The diameter is randomly chosen from the range *dbh_from* and *dbh_to*. The height is calculated using the defined height/diameter ratio (*hd*). All trees of this class have the same *age*:

Column	Description
count	number of trees per hectare
species	the alphanumeric code of the tree species (e.g. piab, fasy)
dbh_from, dbh_to	the range of diameters (cm) for the trees. For each tree, the actual diameter is chosen from a uniform random distribution.
hd	height-diameter relation (m/m). The height of each tree is calculated as dbh*hd
age	(optional) age of the trees (years). If blank or zero the age is estimated.
density	(optional) density value ranging from -1..1 (see description below). Default=0.
stand_id	indicates the *stand* in *standgrid* mode.

Within a stand the positions of the individual trees are derived by applying an pseudo-random approach (more information here) but can also be set using the single tree initialization approach (more information here).

The same approach can be used to initialize saplings (i.e. trees < 4m, see here).

8.3.2 Spin-up

After initializing vegetation a spin-up is run to reach a vegetation state as close as possible to the reference data while also ensuring that the initial state is consistent with model logic (e.g. tree placement, competition effects). The model is run under an approximation of past climate and disturbances (and past management if applicable) for multiple centuries. Spin-ups vary in length and should consider the target age of the oldest forest stands and the timescales needed to represent forest processes, such as decomposition and soil carbon cycling if these processes are enabled. In landscapes where disturbance or management play a pivotal role in shaping forest conditions, historical disturbance or management regimes can be simulated during the spin-up. Furthermore, actual disturbance events or management activities can be spatially and temporally explicitly prescribed (e.g., real fire perimeters are used in the spin-up in Turner et al. (2022)).

A more elaborate spin-up procedure is the directed spin-up. Here the end state of the spin-up is not determined by a simulation length or simultaneously for the whole landscape but rather based on how well each stand corresponds with reference conditions at the given age (Thom et al. 2018).

A spin-up can also be run without previous vegetation initialization and instead starting from bare ground with external seed inputs ("potential natural vegetation" run). There are multiple options for simulating external seed input (see External seed input).

The final vegetation state of the spin-up is then saved as a "snapshot". A snapshot is a full representation of the internal vegetation and soil state of a landscape and is stored in a database. This snapshot can later be used to restore a previous vegetation state. The snapshot includes all state variables of trees, saplings, snags and soil-pools. A snapshot is created by using the JavaScript function `saveModelSnapshot()`. This function creates a SQLite database file (which can be quite large) and an additional resource-unit map as a ESRI raster-file with the same name (and .asc as file extension).

To load a snapshot, one can use the mode "snapshot" in the project-file for the tree initialization (nodes *world.initialization.mode* and *world.initialization.file* in the project file). More information on snapshots can be found here.

8.3.3 External seed input

Especially for "bare ground"/PNV runs but also for other purposes it can be useful to provide the landscape with a continuous input of seeds generated outside of the landscape. Two distinct approaches exist: the "seed belt" approach simulates seed input from forests outside of the landscape, while the "seed rain" approach provides the whole area of the landscape with a continuous supply of seeds.

The next sections give an overview of the two approaches, more detail on their implementation can be found here.

8.3.3.1 Seed belt

The seed belt is a belt shaped area surrounding the project area. This belt acts as a seed source and seeds reach the project area from the belt facilitating the iLand seed dispersal algorithm. The total area is split into a number of *sectors*, whereas each *sector* is characterized by an user-defined species composition. The seed belt is limited to those areas outside of the project area that are flagged as "forested" (pixel values of *-2*, see Stand grid).

The number of sectors, as well as each sector's unique mix of seeds differing in species and seed share (0-1) can be defined in the project file.

The following snippet is an example for defining the seed belt in the project file:

```
<seedBelt>
  <enabled>true</enabled>
    <width>9</width>
    <sizeX>3</sizeX>
    <sizeY>2</sizeY>
    <species_0_0>Abam 0.7 Psme 0.1</species_0_0>
    <species_0_1>Psme 0.1</species_0_1>
```

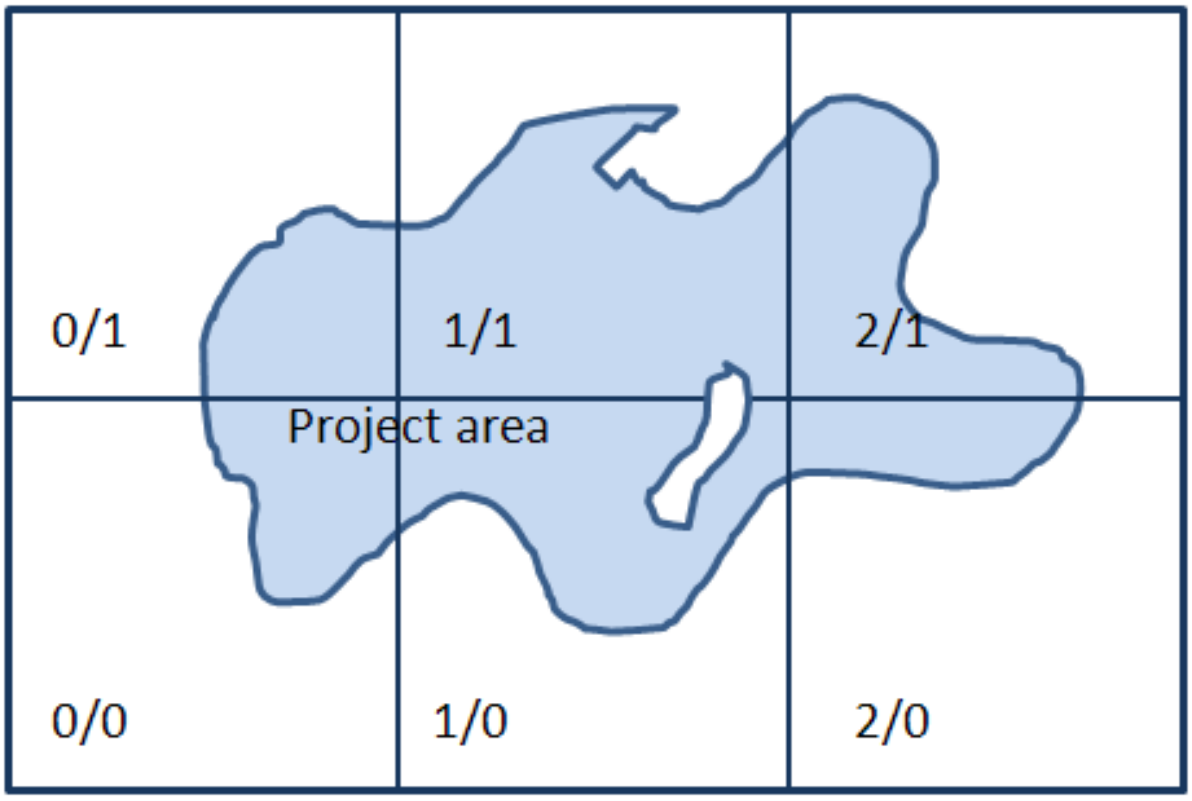

Figure 8.2: Sketch of a project area divided in 3x2 sectors

```
    <species_1_1>Psme 0.1</species_1_1>
    <species_2_1>Psme 0.1</species_2_1>
</seedBelt>
```

Example how a seed belt can be defined in the project file

8.3.3.2 Seed rain

In some cases, it could be useful to have external seed input on the full area, i.e., on all pixels regardless of their location. In the project file a list of species and their seed availability probability can be specified, for example: "`piab, 0.01, fasy, 0.02, bepe, 0.03`". This sets the seed probability of Norway spruce (piab, the species shortName in the species parameter table) to 0.01, of beech to 0.02 and of silver birch to 0.03. The probability is interpreted as the chance of having no seed limitation on each 2x2m cell (e.g., a value of 0.01 would mean that - in the absence of other limitations (environment, light, cell already occupied by saplings of the species) - cohorts would establish every year on roughly 1% of the 2x2m cells).

8.4 Additional steps

Finally, initialization requires information on disturbance and management regimes. Built-in abiotic and biotic disturbance modules must be parameterized to ensure appropriate representation of disturbance size, severity, and frequency (Hansen et al. 2020). Sequences of disturbance events can also be generated separately and prescribed (Albrich et al. 2022), enabling greater control in exploring variation in timing, frequency, size, or intensity among disturbance scenarios. Forest management is extremely flexible: for example, one can define thresholds that initiate actions, develop new treatment methods, include spatially explicit prioritization of management zones, or employ agent-based managers that react to their environment in dynamic and complex ways.

8.4.1 Disturbances

While some disturbance modules (e.g., bark beetle *Ips typographus*) are implemented in iLand fully dynamically, and thus do not need any parameterization before applying the sub-module to new situations, other disturbance modules (e.g., wind, fire) require landscape-specific information for their implementation.

8.4.1.1 Wind

The wind module in iLand (see here for more details) requires two types of landscape-specific inputs: a time series of wind events (with accompanying properties like wind direction and storm duration, see here) and a raster file at a resolution of 100x100 m which scales the global wind pattern to a local value ("topoModifier"-grid).

Data on wind speeds [m/s, measured at 10 m above ground] can for example be obtained from weather stations. Storm events are identified as the 90th percentile of yearly top wind speeds and their main wind direction is also determined. Time series of wind events can then be generated by sampling with a Gumbel distribution (Thom et al. 2022). To account for how wind speeds are modified by topography, spatial data on mean annual wind speeds is used to create the *topoModifier-grid*, which for each 100x100 m cell contains a value with which to modify (additive or multiplicative) the wind speed given in the time series.

Finally, simulated wind disturbance patterns need to be evaluated against data on historical wind regimes (e.g., simulated versus observed wind damage, see Thom et al. (2022)).

8.4.1.2 Fire

Parameterization and evaluation of the iLand fire module should include comparisons of simulated versus observed data to assess multiple aspects of the disturbance regime, such as fire sizes, shapes, annual area burned, and severity. The base fire module is described in detail on the wildfire module overview, fire module parameters and fire module evaluation wiki pages.

The fire module is easily customizable using additional scripts and inputs. For example, Hansen et al. (2020) included a value for KBDI (Keetch Byram Drought Index, the drought indicator used in the iLand fire module) that separated cooler-wetter from hotter-drier years. Fire sizes in iLand were assigned different minimum and maximum sizes and drawn from different fire size distributions depending on whether KBDI was above or below this threshold value. This functionality was implemented in a separate javascript. Or, Turner et al. (2022) combined iLand simulations with fire sizes generated from a separate statistical modeling process that estimated ignition locations and fire sizes under different projected future climate models. Consistent with fire sizes in the western United States, which can be 10,000s to 100,000s of ha, estimated fire sizes could be larger than an iLand landscape. In their paper, Turner et al. (2022) developed a process to first estimate how much of a potential future fire would be expected to "burn in" to a given landscape and then prescribed a distinct sequence of fire events (fire year, ignition location, and maximum size) for each future simulation replicate. This is conceptually similar to the way wind time events are implemented (described above), although in this case it was implemented via a customized JavaScript.

8.4.1.3 Bark beetle

The bark beetle disturbance module (see here for more details) does not need to be parameterized (but note that it is specific to the European spruce bark beetle). It does however need to be adapted to a new landscape using the following nodes in the project file:

- *modules.barkbeetle.referenceClimate.tableName*
  name of the table in the climate-SQLite that represent the reference climate for the landscape. iLand compares the climate values given by that table with the baseline climate values given by the project file nodes *seasonalPrecipSum* and *seasonalTemperatureAverage*
- *modules.barkbeetle.referenceClimate.seasonalPrecipSum*
  comma-delimited list of historic seasonal precipitation sums (spring, summer, autumn, winter) extracted from the climate table given in *tableName*. Example: `212, 234, 190, 179`
- *modules.barkbeetle.referenceClimate.seasonalTemperatureAverage*
  comma-delimited list of historic seasonal mean temperatures extracted from the climate table given in *tableName*. Seasons are: spring: March, April, May, summer: June, July, August, autumn: Sept., Oct., Nov., winter: Dec., Jan., Feb. Example: `7.6, 13.2, 6.9, 3.3`

For more information on the parameterization of the bark beetle module see R. Seidl and Rammer (2017) .

8.4.1.4 BITE

The Biotic Disturbance Engine (or BITE, see here) can simulate biotic disturbances in forest ecosystems. Its framework is also simple, modular and general enough to simulate a wide range of biotic disturbance agents, from fungi to large mammals. More information can be found in Honkaniemi, Rammer, and Seidl (2021) .

8.4.2 Management

If the landscape of interest had been shaped by management in the past, it is recommend to mimic these interventions in the spin-up to generate representative forest states. Two modules for management are integrated in iLand: the base management module (see here) and the Agent Based management Engine (ABE, see here), with the latter one being able to simulate adaptive forest management in multi-agent landscapes. More details on the management modules can be found in the Forest management chapter.

9 Parameterization of new tree species

Species parameterization in forest simulation models is a critical process that entails defining and assigning specific characteristics and behaviors to tree species. In iLand, this meticulous procedure is indispensable for accurately modeling the growth, mortality, and regeneration of individual tree species and their complex interactions.

iLand currently simulates over 100 tree species across Europe, North America, and Japan. Each species is characterized by 66 species-specific parameters. A comprehensive collection of these parameter values can be found in Thom et al. (2024).

9.1 Data requirements and sources

Collecting comprehensive data for parameterization, calibration, and model evaluation is a considerable challenge. In iLand, we have leveraged various data sources, including species trait databases (e.g., TRY Plant Trait database (Kattge et al. 2020)), field data, national forest inventories, peer-reviewed scientific literature (e.g., (Niinemets, Valladares, and Medioambientales 2006)), grey literature (e.g., Silvics of North America (Burns, and Honkala, 1990)), and other ecological models (e.g., (Purves 2007; Nitschke and Innes 2008)). For additional data source ideas, explore the iLand wiki, especially the species parameter page, and review previous iLand publications in your geographic region, especially their supplementary material. For example, Rupert Seidl, Rammer, et al. (2012) includes detailed data sources for species parameters in Central Europe and the US Pacific Northwest in their supplementary material, as does Braziunas et al. (2018) for species parameter in the US Northern Rocky Mountains in their supplementary material.

Environmental and forest data, encompassing climate variables, soil conditions, and long-term empirical observations, are essential for effective parameter calibration (see also Ingredients for an iLand project, Setting up landscapes, and Calibration and evaluation chapters).

9.2 Species parameterization process

Parameterizing a new species in iLand follows an iterative, step-wise process that is closely intertwined with model evaluation (Figure 9.1). Some parameters can be readily measured in

the field or extracted from existing data sources such as those described above, whereas others pose greater challenges and require iterative estimation.

Remember to check whether a tree species has already been parameterized. Initial parameter sets are already available for over 100 tree species (Thom et al. 2024), and subsequent studies may have already added new species (see the iLand publications page).

Once parameters have been compiled, the next steps are calibration and evaluation (next book chapter). Calibration involves iterative adjustments of species parameters based on continuous analysis of model outputs across all relevant spatiotemporal scales: individual tree, stand, and landscape levels. Model evaluation is conducted using a pattern-oriented approach (Grimm et al. 2005). This entails comparing iLand's outputs against observed patterns, such as individual tree dimensions, stand-level productivity and mortality, tree responses to disturbance, and simulated potential natural vegetation at landscape scales.

> **Tip**
>
> iLand's reliance on first principles in ecology means that site-specific parameter adjustments are not recommended unless they enhance simulation performance across different regions simultaneously, avoiding local overfitting and ensuring robustness under global change conditions. Parameters should broadly represent species across diverse conditions, occasionally sacrificing precision for accuracy in simulated outcomes. For species with wide-ranging conditions or specific applications, distinguishing tree species provenances (e.g., boreal vs. temperate *Pinus sylvestris*) in parameterization is meaningful.

9.2.1 A tiered approach

Parameters can be sorted into three tiers, with different associated calibration and evaluation steps. The first tier describes tree allometry, growth, and environmental responses, typically obtained from literature sources. These parameters are then evaluated through stand-level simulations focusing solely on growth, temporarily disregarding regeneration and mortality processes (see Productivity and mortality section in next chapter).

The focus then shifts to the second tier, parameters describing tree mortality and aging. These include readily available life history parameters (e.g., maximum age and height) as well as parameters that are difficult to determine empirically and require iterative estimation (e.g., the relationship between mortality probability and carbon starvation). Stand-level evaluations at this stage concentrate on reproducing stand density patterns over time (self-thinning) and simulating maximum tree properties (see Productivity and mortality section in next chapter).

The final tier of parameters characterizes seed production and dispersal, tree establishment, and sapling growth. These include empirically derived estimates (e.g., seed dispersal kernels, sapling growth potential) and more challenging parameters like fecundity (number of seeds produced per unit canopy area). If regeneration data is available, specific evaluation tests can

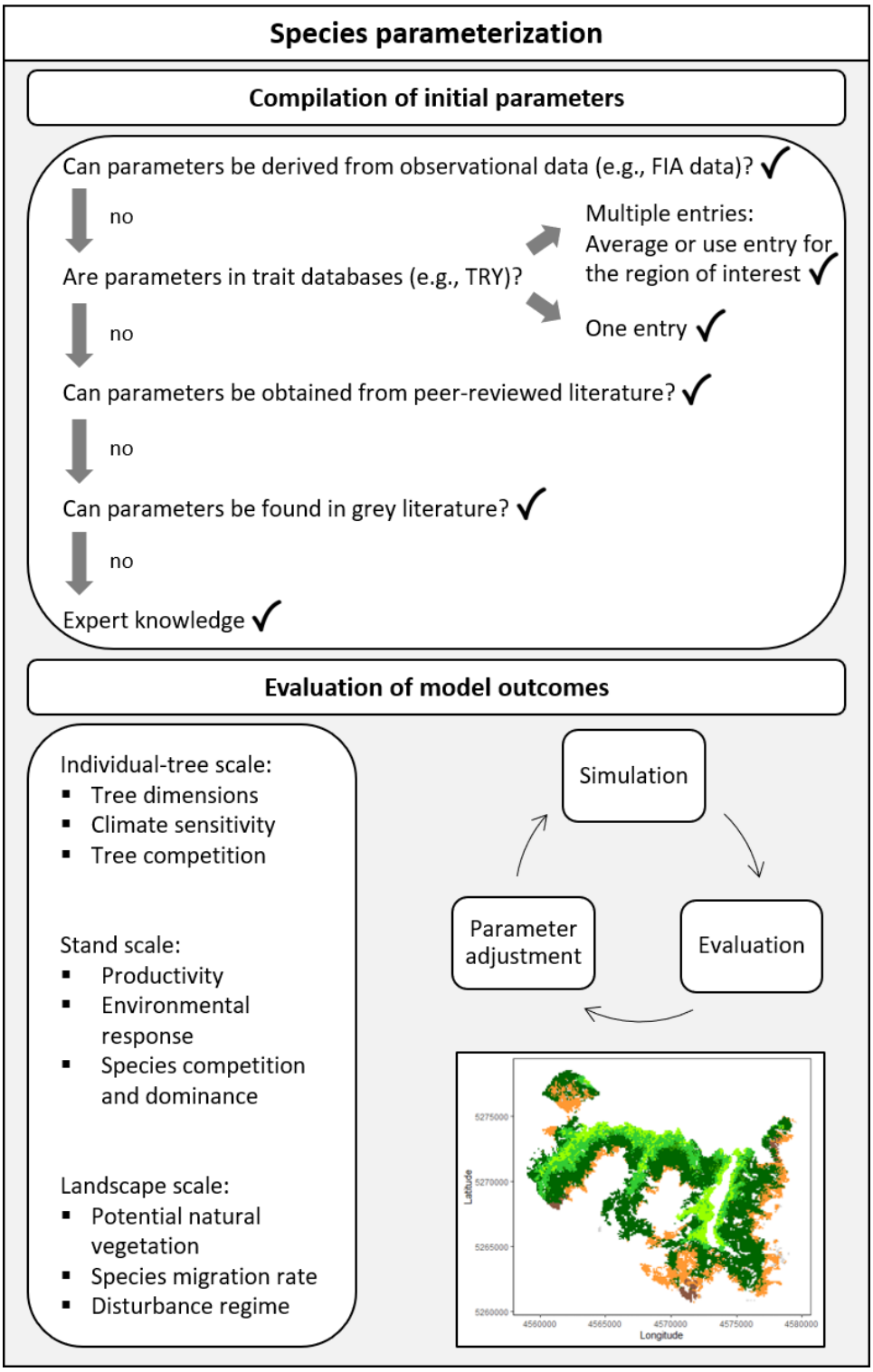

Figure 9.1: To derive a robust species parameter set for process-based modeling, start by compiling an initial set of parameters from multiple sources. Next, simulate different ecosystem patterns and evaluate these simulations against independent observations. Parameters may need to be iteratively adjusted, ensuring that each value remains within an ecologically plausible range. However, it's important to avoid local overfitting to maintain realistic responses to novel environmental conditions.

be set up to assess these parameters. In the absence of such data, a coarse filter approach is employed, involving landscape-scale simulations of potential natural vegetation development over long time frames. These simulations are evaluated based on the realism of stem densities and diameter distributions, the proximity of simulated basal area and biomass pools to observed values, and the dominance of expected early- and late-seral species under appropriate conditions. The successful emergence of these expected patterns from the simulation indicates that the parameters related to all three demographic processes (growth, mortality, and regeneration) have been well-specified (see PNV, regeneration, and competition section in next chapter).

9.2.2 Specific parameters with high uncertainty that often require manual testing and calibration

- *Aging*: Will affect both growth and mortality. Plot the curve and to check if early seral species have a more drastic decline in carbon use efficiency than late seral species. This parameter has a great impact, but little is known about it. It can thus be modified with little concern, although always take care that the relative differences with other species makes sense.
- *Fecundity_m2*: The "standard procedure" to derive fecundity as used in the past by iLand users yields very high values. Values beyond 100 are usually unrealistic. Most values should be between 1 and 10. A change here might have a great impact on the dominance of species (e.g., in PNV tests).
- *HDlow and HDhigh*: Can be easily derived from inventory plots, but the result might be quite extreme as trunks might be broken etc. Thus it is important to check whether the range is plausible. If trees grow too thick or too high, this is a good parameter to change.
- *estMinTemp* and *estGDDMin*: This will change both the spatial range of persistence and also the dominance at the trailing zone. While *estMinTemp* can be used to rather restrict a species in oceanically influenced ecosystems, *estGDDMin* might be more useful to restrict the spread into the continental cold zones. However, both likely require adaptation in parallel. I do not recommend using *estGDDMax* to restrict the growth of species as the parameter is not ecologically meaningful and could thus cause a bias in climate change simulations.
- *sapHeightGrowthPotential*: The parameter is quite uncertain, but can have a large impact on what species will outgrow another one.
- *lightResponseClass*: If a species gets lost too quickly in multi-species tests, it might be because of shade from other tree species. There are multiple studies on shade-tolerance, you might want to check if there are different values or you might have to consider the

change in shade tolerance over age (e.g., oak is more shade tolerant in its youth than later).

- C allocation, especially *bmRoot*: If growth doesn't make sense, these should be checked, although these values are hard to estimate. However, there is quite some uncertainty, especially for root C allocation. It can be useful to check for other publications or within a trait database and try another published value.
- *respVpdExponent*, *maxCanopyConductance*, *psiMin*: If you observe drought-induced mortality waves that you wouldn't expect, these are parameters which should be checked.
- *respNitrogenClass*: If a species doesn't grow well or dies under specific conditions, this is a parameter that could make a difference. Always think this one relative to the demand of other tree species.
- *respTempMin* and respTempMax: Can improve your growth curve in response to temperature. This makes sense to control relative to other species. In particular, if a species has a wide ecological niche and covers a large spatial area, it is possible that different provenances have a different minimum or optimum to conduct photosynthesis.

Well-known parameters, such as *maximumHeight* or *maximumAge*, should not be changed to improve model results.

9.3 Light influence patterns

Every tree species needs to have a light influence pattern (LIP) file in order to run in iLand (more info: competition for light). This LIP file is a lookup table with pre-calculated shading patterns based on a range of tree diameters and heights for the given species; this is created with the standalone Lightroom software. Prior to running Lightroom, parameters should be compiled for *HDlow* and *HDhigh*, for crown shape and size (Purves 2007), and for the range of tree sizes that are expected to be simulated in iLand (consider *maximumHeight* parameter). Then follow the step-by-step instructions on the iLand wiki.

9.4 Biome and landscape parameters

Biome and landscape parameters apply to all trees in the simulation and are set in the project file. These parameters should generally start with default values but can be adjusted if all species are consistently underperforming in evaluation experiments. Always check latitude in the project file to ensure that day length calculations are correct.

10 Calibration and evaluation of species and simulation results

Once species are parameterized, the next steps are calibration and evaluation. This is an iterative process. When evaluations highlight discrepancies between simulations and comparison data, parameters can be changed, or calibrated, to improve evaluation outputs and model fit (see examples of parameters to focus on here and in sections below). It is important to think critically, and ecologically, about the evaluation experiments during this process. Model fits should not be expected to be perfect; data inputs such as climate and soils have inherent uncertainty or simplification, some processes such as disturbances are not represented in the basic evaluations, and comparison datasets may have their own biases (e.g., site indexes are idealized curves, forests may have unknown management history or be influenced by unmeasured neighboring stands). The evaluations should help you build confidence that the model is representing processes of productivity and growth, mortality, regeneration, and competition in a way that aligns with available data and your ecologically-informed expectations.

There are several options for evaluating iLand's ability to simulate realistic forest growth and composition patterns. Evaluation tests encompass different hierarchical levels (individual, stand, and landscape; see Table 10.1) following a pattern-oriented approach (Grimm et al. 2005) and are usually run under historical climate conditions. The suite of evaluation tests has to be designed carefully for each project as there is no one-size-fits-all approach (e.g., due to limited data availability). This section gives an overview of the types of tests used in previous studies, their data requirements, simulation set-up, calibration recommendations, and evaluation strategy.

Included in this process are the iterative calibration of some species parameters that are derived from model simulations, rather than the literature (see also Species parameter chapter).

Table 10.1: Examples of evaluations at different hierarchical levels following a pattern-oriented approach.

Hierarchical level	Evaluation target
Individual-tree level	• Tree dimensions (e.g., average and distribution of diameter at breast height (dbh) and tree height) for each species, typically documented from historical observations or old-growth forests. • Climate sensitivity (e.g., annual growth anomalies), obtainable from regular tree growth measurements. • Tree competition (e.g., growth response to tree neighborhood), evaluable against data from silvicultural trials.
Stand level	• Stand productivity (e.g., volume, basal area, dbh, and height increment), testable for single-species and mixed-species stands using local forest inventories and yield tables. • Environmental responses (e.g., growth, mortality, and regeneration changes due to water stress), with data from permanent forest monitoring plots, eddy covariance flux towers, or environmental gradients. • Species competition and dominance (e.g., growth, mortality, and regeneration in species mixtures), comparable with periodic inventories and species mixture trials.
Landscape level	• Potential natural vegetation (e.g., natural succession towards a stable species composition in equilibrium with climatic conditions), comparable with local floristic assessments and unmanaged forest observations. • Species migration rates (e.g., movement of species across landscapes), comparable with paleo records or observations of current climatic changes. • Disturbance regimes (e.g., rates, sizes, frequencies, and interactions), comparable with remote sensing data, terrestrial inventories, or field data.

The examples below are grouped by ecological processes and parameter tier, which include evaluations at different hierarchical levels.

10.1 Productivity and mortality

Productivity tests assess how well iLand characterizes structural trajectories and variability, e.g., tree densities, basal areas, and mean heights (Braziunas et al. 2018); basal area increment (Kobayashi et al. 2023); site index (Rupert Seidl, Rammer, et al. 2012); or dominant tree heights (Thom et al. 2022). The data available for comparison - e.g., yield tables (Albrich et al. 2018), Forest Inventory Data or independent field observations (Braziunas et al. 2018) - should be considered during the simulation setup. Most commonly monospecific stands are simulated under climate conditions representing the comparison data's climate period and

without additional disturbances. If the comparison data was measured in managed stands these management interventions should be mimicked in the simulations (Albrich et al. 2018). The simulation length varies depending on the comparison data (e.g., stand age, inventory frequency, etc.). If single-tree data is available from inventory plots, observed and predicted annual growth rates can also be compared. Success is gauged based on how well ranges or values of simulated data fit with the observed data and whether natural dynamics of stand development are reproduced (e.g., self-thinning).

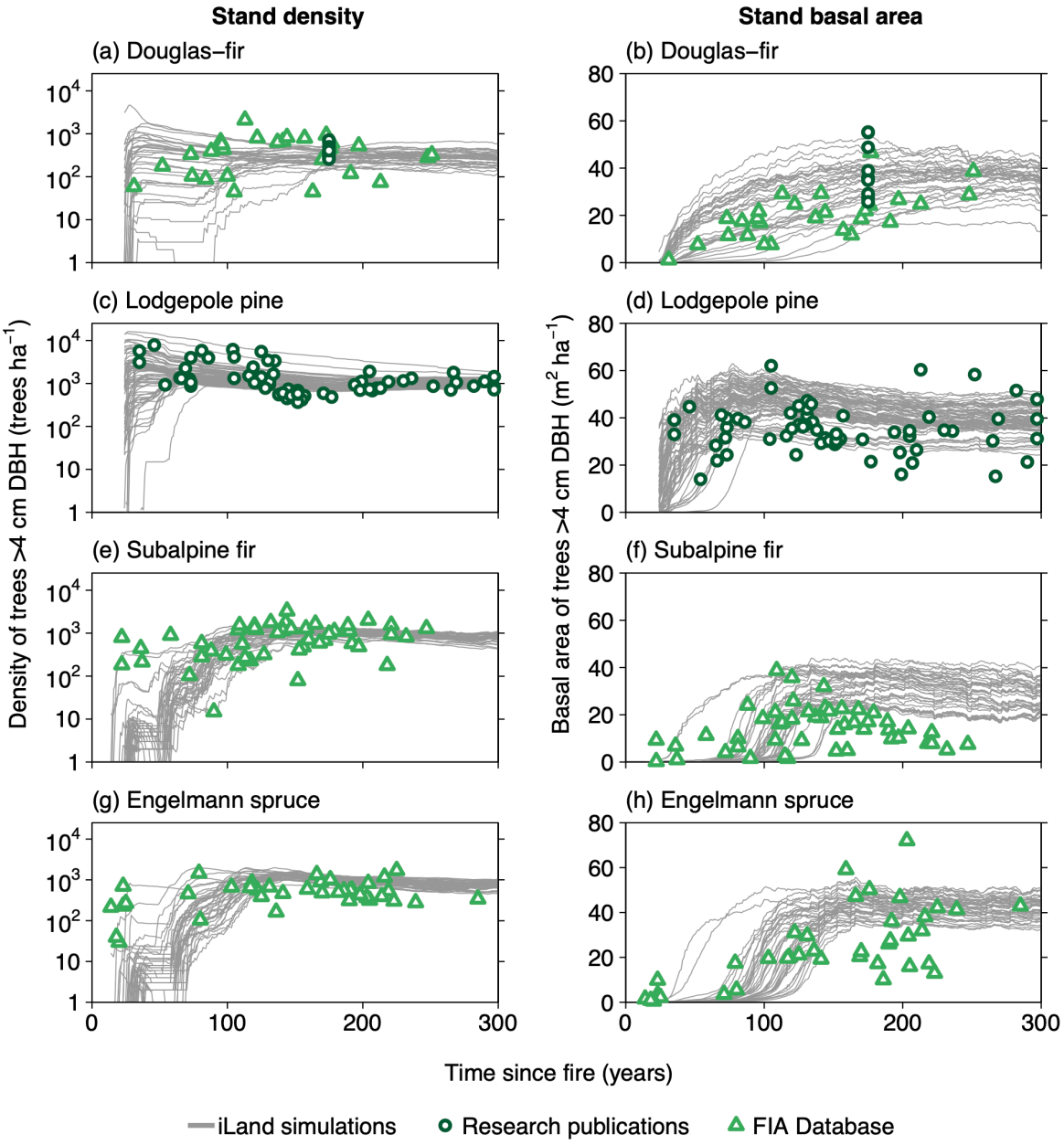

Figure 10.1: *Evaluation of monospecific stand density and basal area trajectories simulated in iLand (lines) compared with field observations (points) from Braziunas et al. (2018), Supplementary material.*

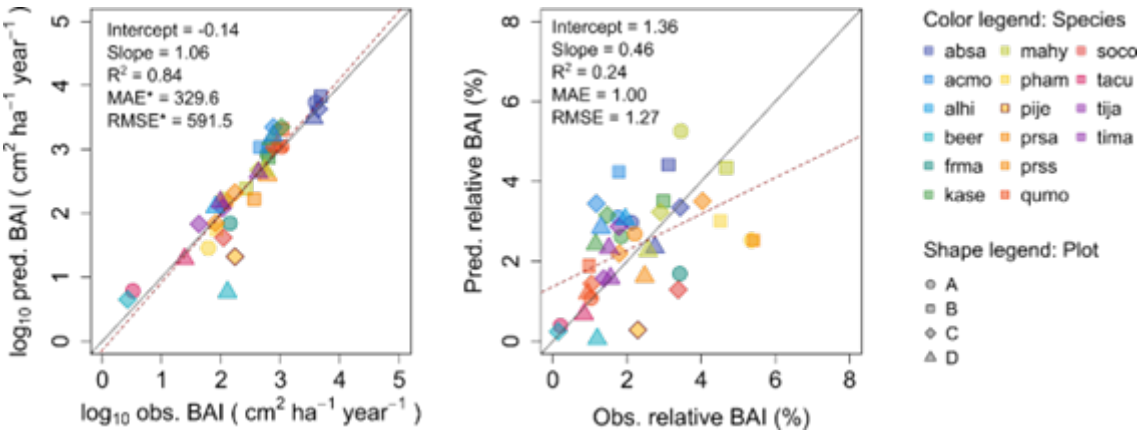

Figure 10.2: *Evaluation of basal area increment for multispecies stands simulated in iLand (predicted) compared with field observations (observed) from Kobayashi et al. (2023), Supplementary material.*

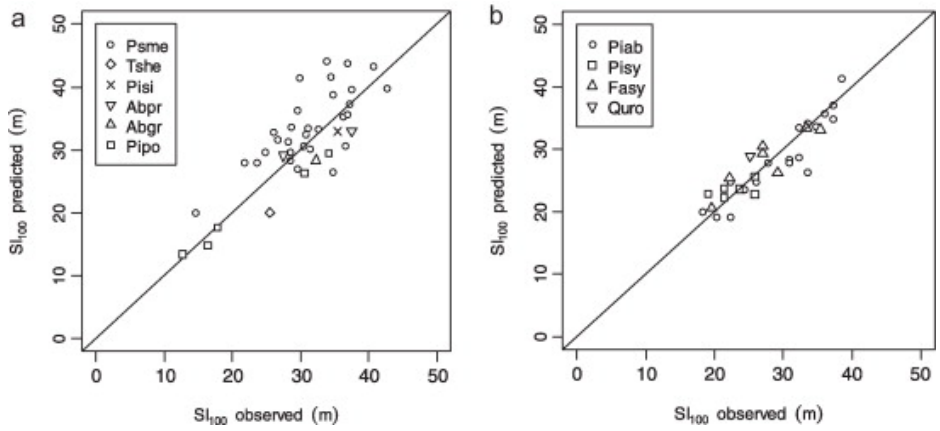

Figure 10.3: *Evaluation of site index for monospecific stands simulated in iLand (predicted) compared with field observations (observed) from Rupert Seidl, Rammer, et al. (2012).*

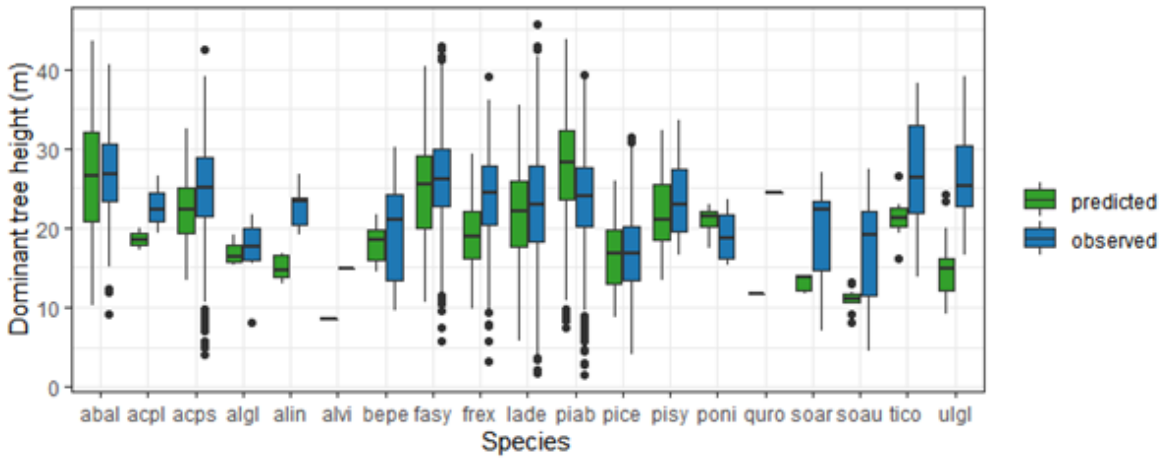

Figure 10.4: *Evaluation of dominant (i.e., 95th percentile) tree height for monospecific stands simulated in iLand (predicted) compared with forest inventory data (observed) from Thom et al. (2022), Supplementary material.*

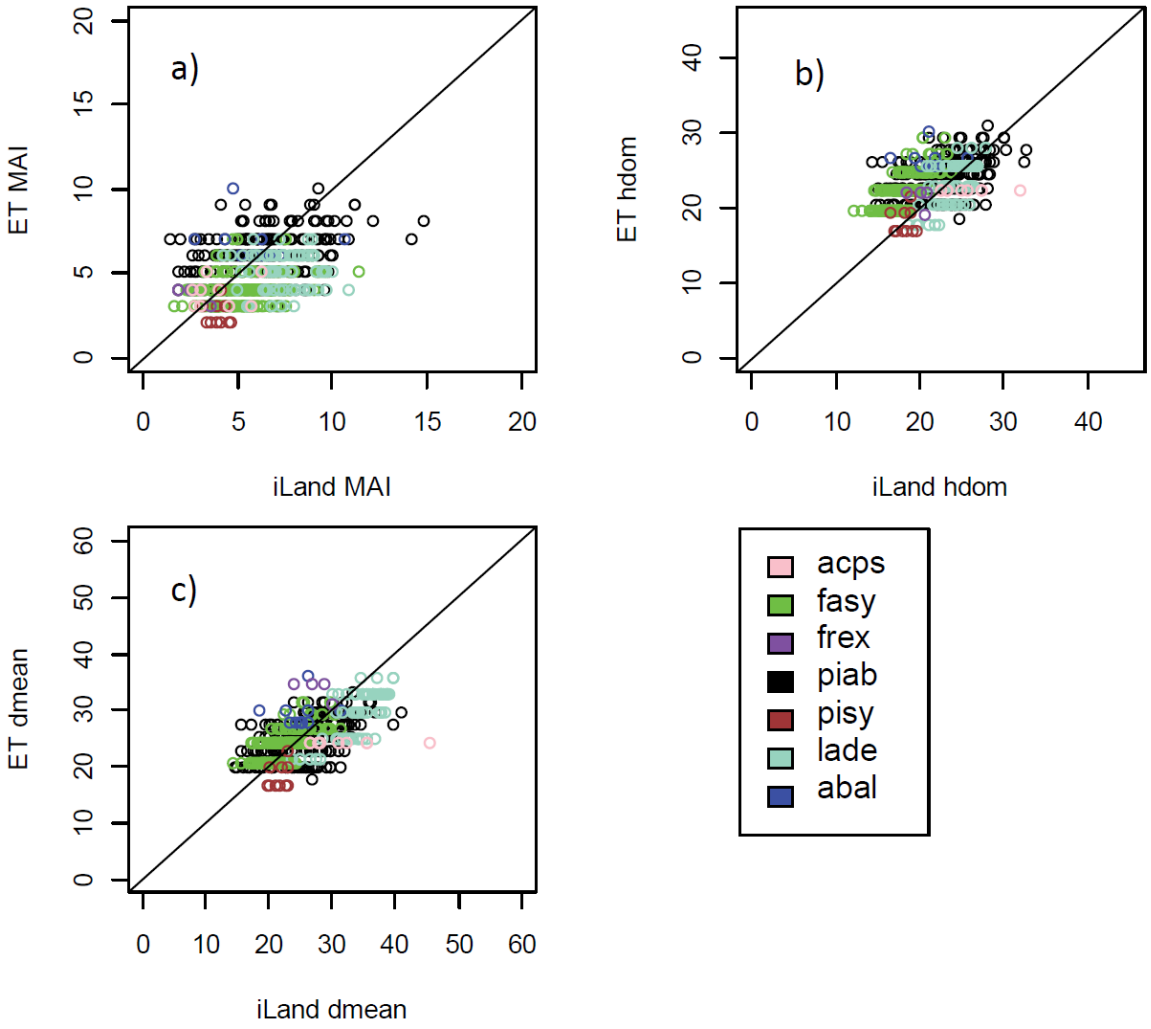

Figure 10.5: *Evaluation of tree productivity for managed monospecific stands simulated in iLand (iLand) compared with yield tables (ET) from Albrich et al. (2018), Supplementary material.*

Suggested setup

- Use field observations and/or spatial iLand inputs to generate a set of stands that cover a low to high productivity gradient for the species of interest (i.e., an environmental gradient of less favorable to more favorable site conditions in which that species occurs).

- Initialize trees in monospecific stands along this gradient and simulate stand development without allowing for interactions between stands (i.e., torus mode enabled).
- Initial tree densities and sizes can be assigned based on field data or approximated using available data on stocking levels, stand density index, or stem density from yield tables for a given species.
- Depending on the comparison data used, implement management prescriptions to control for stand density (e.g., when comparing to yield tables).
- Do not simulate any disturbances.
- Productivity experiments can be run with mortality and regeneration off, or with one or both turned on, at different stages of the process to focus on calibration or evaluation of different processes.

This same initial set of stands can typically be used for all subsequent stand-scale evaluations, including both monospecific and multispecies runs.

Calibration recommendations

For productivity experiments, tweak species parameters to better match comparison data. Remember, these are not expected to match perfectly and over-calibration of species to match one specific data set should be avoided. Review Rupert Seidl, Rammer, et al. (2012), Supplementary material for ideas of parameters to which productivity is most sensitive, and keep in mind which parameters are most uncertain versus ones that are fairly well defined. Here are some examples:

- Simulated site index (SI) and diameter at breast height (DBH) are especially sensitive to: Response to plant-available nitrogen (`respNitrogenClass`), minimum and optimum growth temperature (`respTempMin` and `respTempMax`), shade tolerance (`lightResponseClass`), and height to diameter ratio (`HDlow` and `HDhigh`).
- Simulated SI and DBH are also sensitive to many biomass allometrics (`bmWoody`, `bmBranch`, etc.), wood density, and form factor. These are generally well established parameters based on the literature or fit to empirical data, so avoid changing them unless nothing else works.
- Other parameters to consider tweaking, which are somewhere in between uncertain and well defined: Specific leaf area (`specificLeafArea`, SLA, highly variable in the literature) and soil water response (`psiMin`, although note per the Rupert Seidl, Rammer, et al. (2012) sensitivity analysis, growth is not especially sensitive to this parameter).
- There are also some default, landscape, or biome-specific parameters or inputs to consider, but only if you have a consistent problem with productivity across all species. For example, potential light use efficiency (epsilon) has been adjusted in iLand applications in different regions. Another key driver of productivity is the environmental input data

(e.g., soil fertility). If all stands are consistently underperforming, consider adjusting these values (e.g., bump up soil fertility across all stands/region).

- Note that iLand also has an extensive set of debug outputs that you can turn on or off to help determine what is limiting tree growth or survival. For example, `Daily responses` helps you determine which factors among temperature, soil water availability, and vapor pressure deficit are most limiting growth for a given species at a daily time step. If you turn on daily responses, only run simulations for 1 or a few years, as these outputs can be quite large in size.

The `aging` (i.e., the decline in production efficiency with age) parameter must be iteratively calibrated. Mortality should be turned on, and stands should not be managed or subject to disturbance during aging calibration. Simulations should be run for longer than the maximum age of a given species. Then, iteratively adjust the coefficients of the aging equation so that:

- Productivity declines with age, meaning that DBH and height curves are initially steep but shallow/flatten over time. Early seral species should be expected to have a more drastic decline in carbon use efficiency than late seral species.
- Trees die as they approach max age and do not greatly overshoot max age.
- Trees achieve reasonable maximum height and DBH. Keep in mind that maximum age, height, and DBH are theoretical maximum values. Whether and how close simulated trees are to these values should be based on your ecologically- and regionally-informed expectations.

For mortality experiments (after calibrating aging, keep motality on), consider the following:

- If trees are dying too young or too quickly, consider adjusting `probIntrinsic` or `probStress`.
- If it seems like early mortality is due to low productivity and resulting carbon starvation, consider adjusting growth-related parameters described above. The debug outputs can help with this assessment, especially `Daily responses` (e.g., if this output indicates a strong limitation of soil water, consider adjusting `psiMin`).

Finally, the `sapReferenceRatio` parameter must also be iteratively calibrated. This can be done with the same set of stands (i.e., stands covering a wide environmental and productivity gradient where you expect to find the focal species). To do this make sure the species parameter `sapReferenceRatio` is set to `1` and regeneration is turned on in the project file. Then, start simulations from bare ground (no trees, saplings, or seedlings) with external seeds turned on for the species of interest and the establishment debug output (`64`) turned on. Run for a few years (e.g., 5-10 years). Update the species parameter `sapReferenceRatio` to the highest value for `fEnvYr` from the debug output.

Evaluation strategy

Depending on the evaluation, compare iLand simulations over time (e.g., how well do simulations overlap with comparison data) or at a specific point in time (e.g., year 100) with field observations or other empirically derived data (e.g., site index or yield estimates for year 100). "Over time" comparisons evaluate how well simulations cover the range of comparison data, qualitatively and quantitatively. "Point in time" comparisons evaluate how well simulations correspond with comparison data via scatterplots, boxplots, and 1:1 lines. Model performance can be evaluated using quantitative comparisons such as root mean squared error (RMSE), slope, bias, correlation (r), goodness of fit (R^2), or other metrics.

Mortality evaluations can also include comparisons with general empirical self-thinning coefficients (Rupert Seidl, Rammer, et al. 2012).

10.2 PNV, regeneration, and competition

Potential natural vegetation (PNV) evaluations assess how well iLand characterizes successional trajectories and species composition in late-seral stages. This can be done at stand or landscape scales. Field observations for a given forest type (Braziunas et al. 2018), inventory data (Kobayashi et al. 2023), or descriptive accounts of the potential natural vegetation (Albrich et al. 2018; Thom et al. 2022) can be used for comparison. Simulations are usually started from bare ground with external seeds supplied (e.g., via seed belts) and run for 100s-1000s of years under historical climate conditions with no management. Longer time periods are necessary for landscape PNVs, because species will usually only enter from the edges of landscape. PNV simulations can be configured to include or exclude disturbances. For example, disturbances may be excluded from stand-scale PNVs to view stand development and successional trajectories. Or, for landscapes where forest structure and species composition is strongly shaped by disturbances, PNV simulations should consider including the relevant disturbance regimes so that outcomes will better align with expectations. Success is gauged based on how well iLand reproduces typical successional patterns and anticipated species composition (e.g., stratified by forest types or elevation zones).

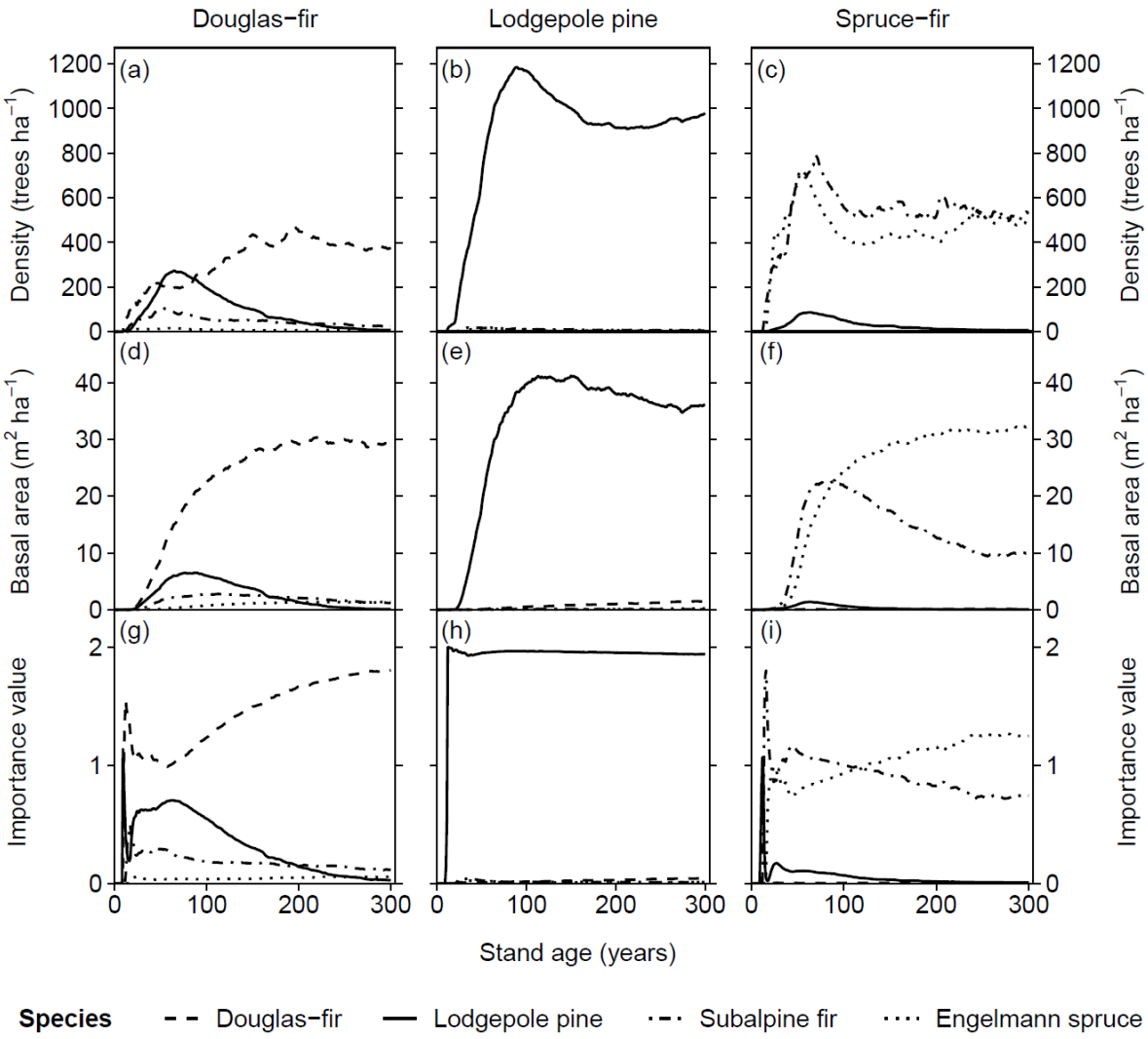

Figure 10.6: *Stand-scale evaluation of multispecies stand density, basal area, and importance value trajectories simulated in iLand for different forest types from Braziunas et al. (2018), Supplementary material.*

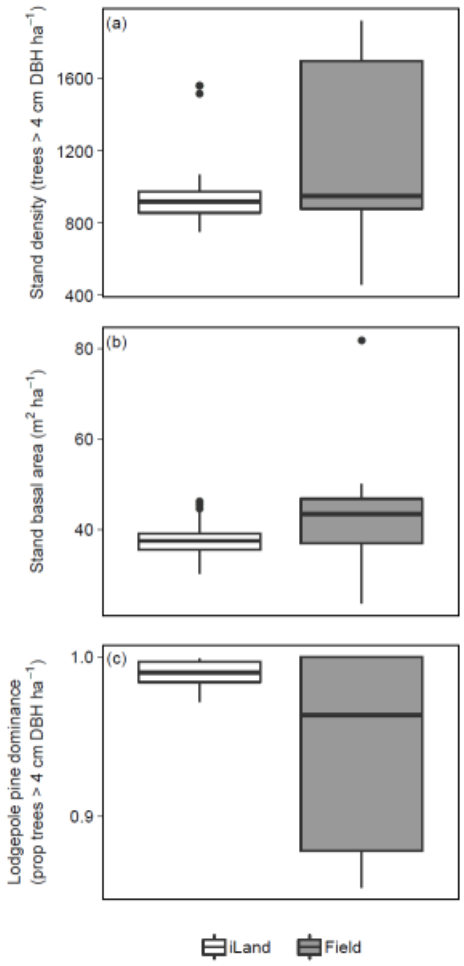

Figure 10.7: *Stand-scale evaluation of multispecies stand density, basal area, and importance value distributions for mature, 300-year-old lodgepole pine forests simulated in iLand compared to field observations representative of that forest type, from Braziunas et al. (2018), Supplementary material.*

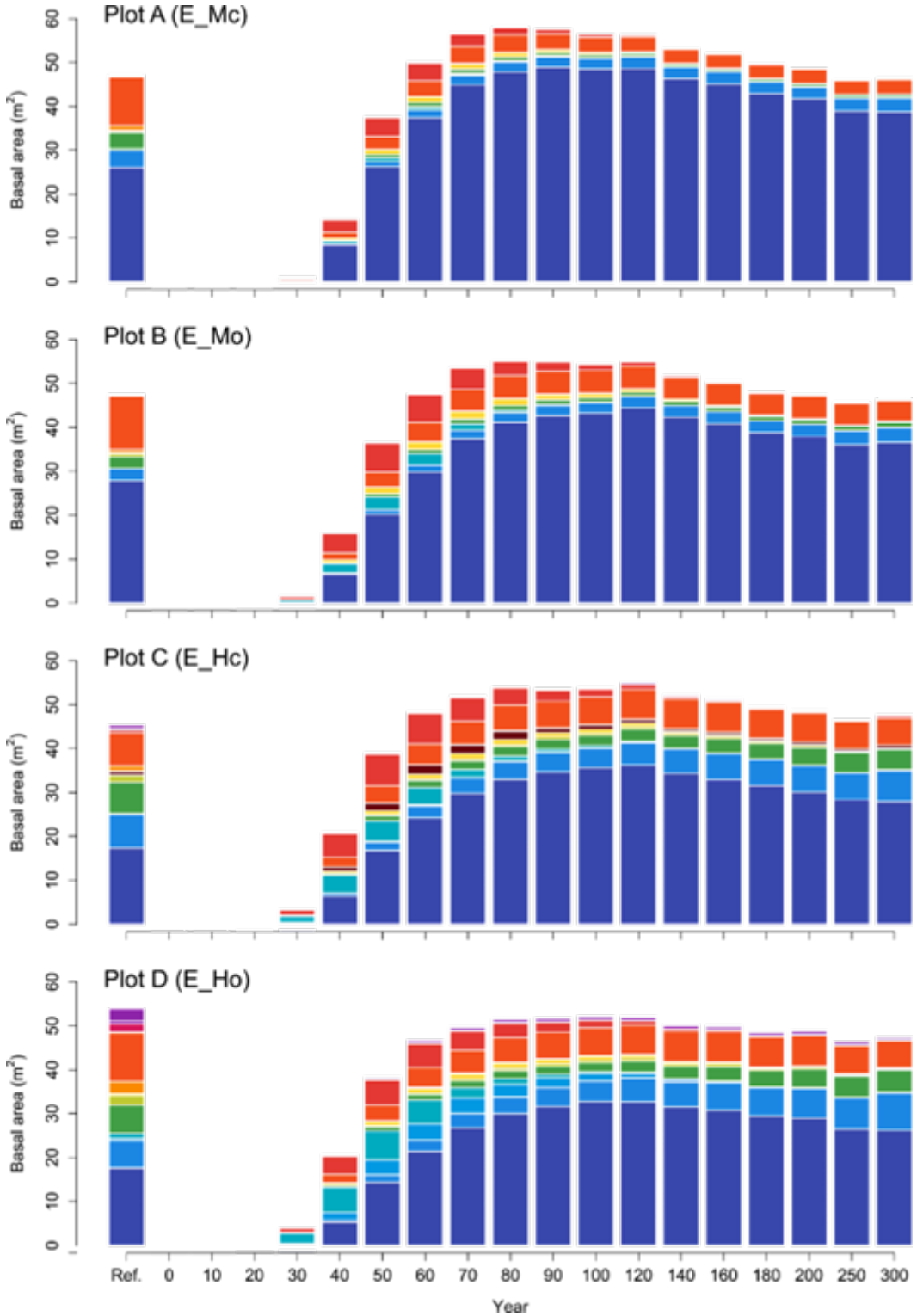

Figure 10.8: *Stand-scale evaluation of basal area and species composition trajectories simulated in iLand compared to inventory data from Kobayashi et al. (2023), Supplementary material.*

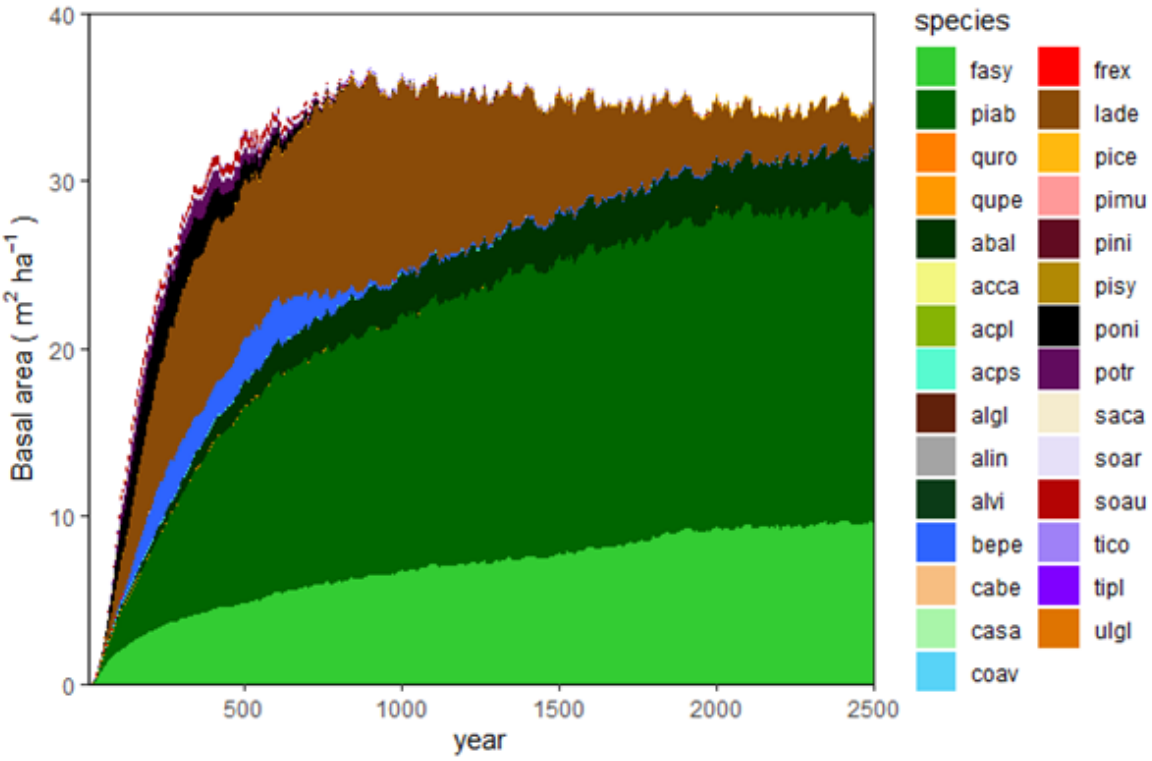

Figure 10.9: *Landscape-scale potential natural vegetation evaluation, starting from bare ground with seed inputs via a seed belt around the edges of the landscape with species composition representative of surrounding PNV forests, from Thom et al. (2022), Supplementary material.*

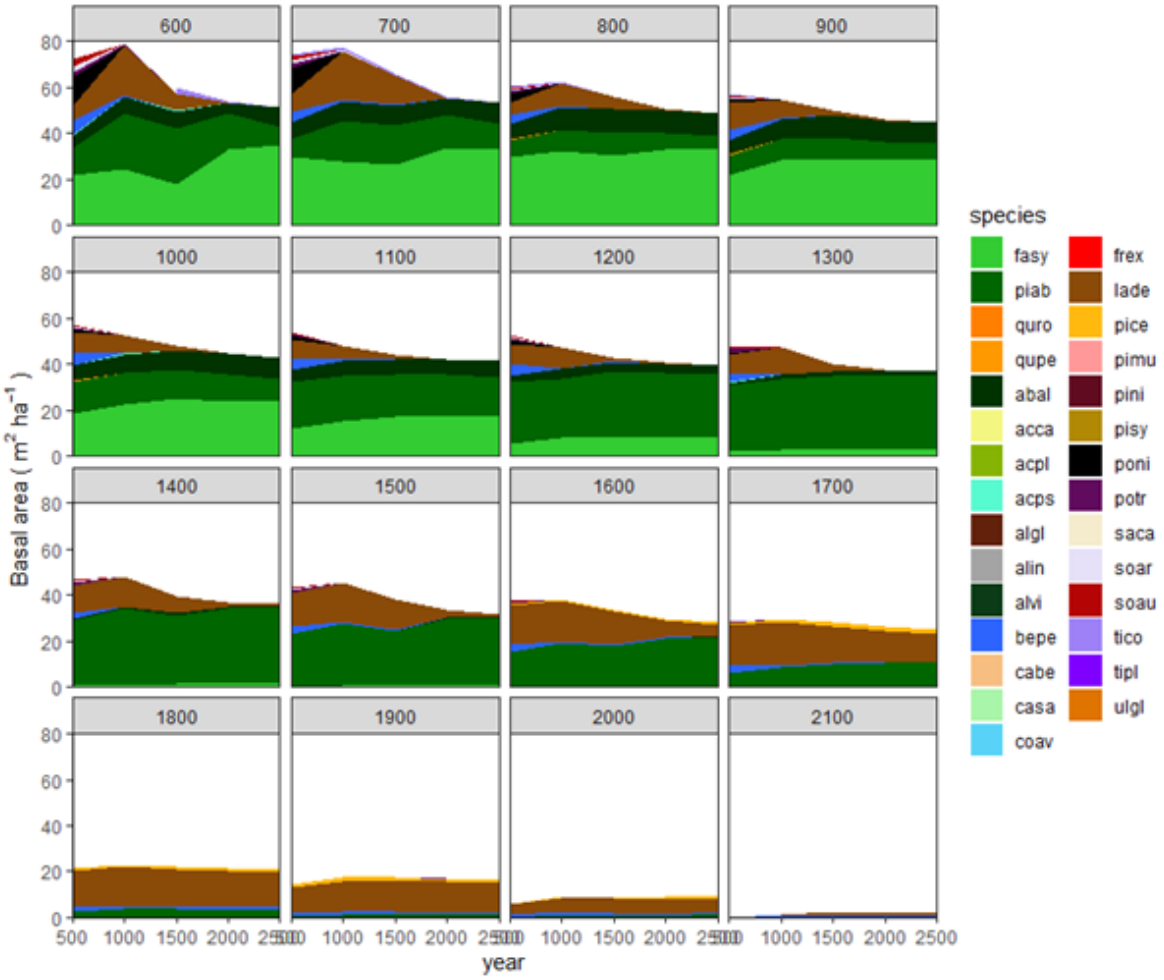

Figure 10.10: *Landscape-scale potential natural vegetation evaluation from the same simulations as the above figure, with trajectories shown for different elevation belts, from Thom et al. (2022), Supplementary material.*

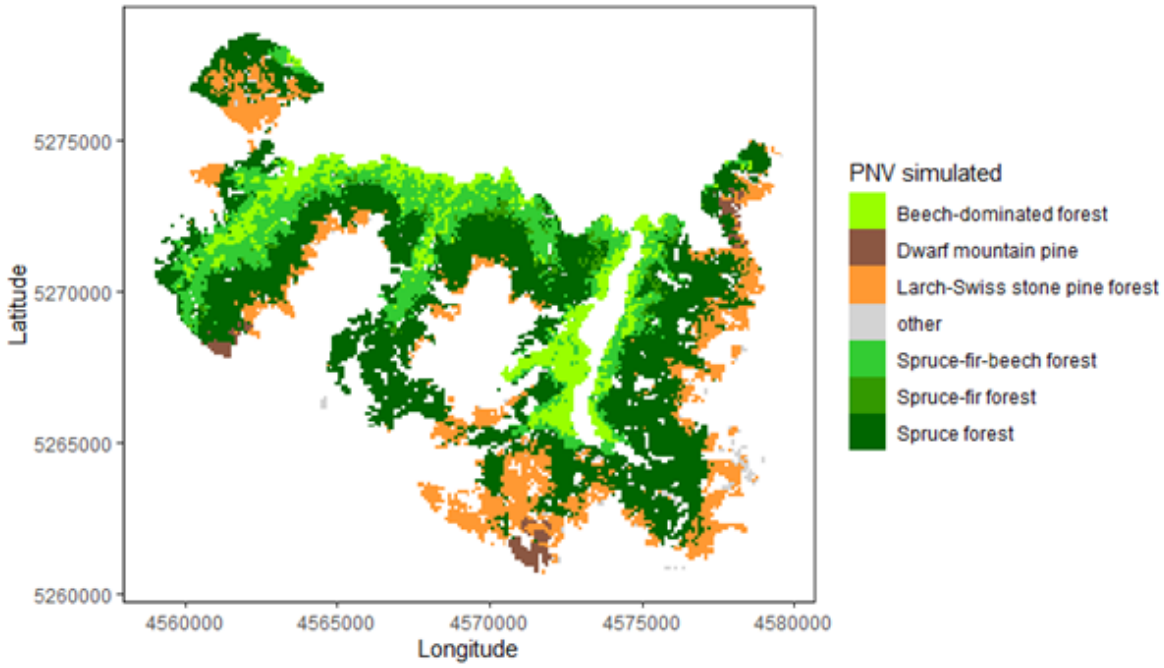

Figure 10.11: *Landscape-scale potential natural vegetation evaluation, map of forest types after 2500 simulation years, from Thom et al. (2022), Supplementary material.*

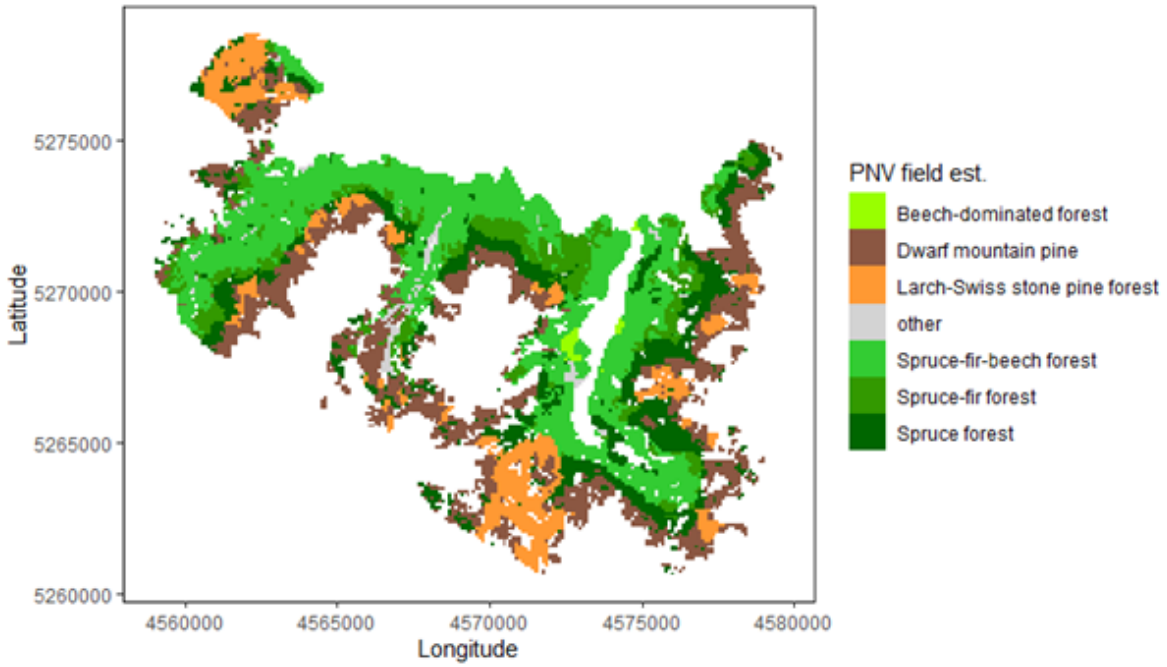

Figure 10.12: *Map of expected PNV forest types for comparison with PNV simulation, from Thom et al. (2022), Supplementary material.*

Suggested setup

For stand-scale evaluations:

- Stands should be selected based on having environmental conditions that are representative for a given dominant species, forest type, or species mix. These could be all or a subset of stands used for productivity evaluations described above. Alternatively, stands could be selected by using other spatial data (e.g., a forest type map, an elevational gradient) to identify representative locations within a study region or iLand landscape.
- Initialize stands either with seedlings or starting from bare ground. When starting from bare ground, seeds can come from a surrounding seed belt with user-defined species composition (e.g., approximated or field-derived composition for the given forest type or specific stand) to simulate post-disturbance or -clearing successional trajectories. Alternatively, bare ground simulations can be run with equal, very low probability of seed

inputs for all potentially present species and run for 100s to 1000s of years to simulate equilibrium species composition and dominance for a given set of environmental conditions.

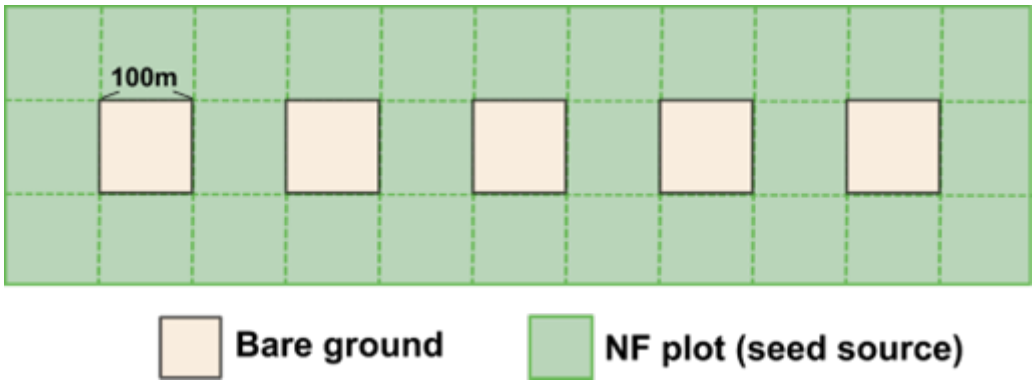

Figure 10.13: *Stand-scale simulation setup showing simulated stands and surrounding seed belt from Kobayashi et al. (2023), Supplementary material.*

Landscape-scale setups require compiling landscape extent, biophysical drivers, and vegetation conditions as described in the Setting up landscapes chapter. It is also important to consider disturbances at this stage, because the PNV for a landscape may be strongly shaped by past disturbances. These could be natural disturbances such as bark beetle outbreaks or fire, or disturbances due to human management. The PNV should be run starting from bare ground, meaning no trees, saplings, or seedlings on the landscape.

- The first step is to create a seed belt (see Figure 10.14) around the edge of the landscape that represents the surrounding forest. See the iLand wiki external seeds page for more information. The seed belt should be adjacent to the forested edges of the landscape and should take topography and other landscape features into consideration (e.g., you might expect seeds to enter from the bottom of a valley but not from the top of a mountain, you would not expect seeds to arrive from a lake). The species composition of the seed belt should take into consideration the forest type that you expect to be there, or that is present today, but it is essential that seeds from all species present in the landscape are present in at least some proportion. Otherwise, they will never be able to migrate in.
- The landscape PNV should then be run under historical climate for enough years for species to migrate into the landscape and achieve some sort of equilibrium relative to the drivers (which would include climate, soils, and disturbances). This is usually at least 1000 years, but can often be longer (e.g., 2500 years for Berchtesgaden examples included above). If there is a lot of variability, such as in disturbance regimes, PNVs

can be run for multiple replicates or the results from the last ~100 years or so can be averaged.

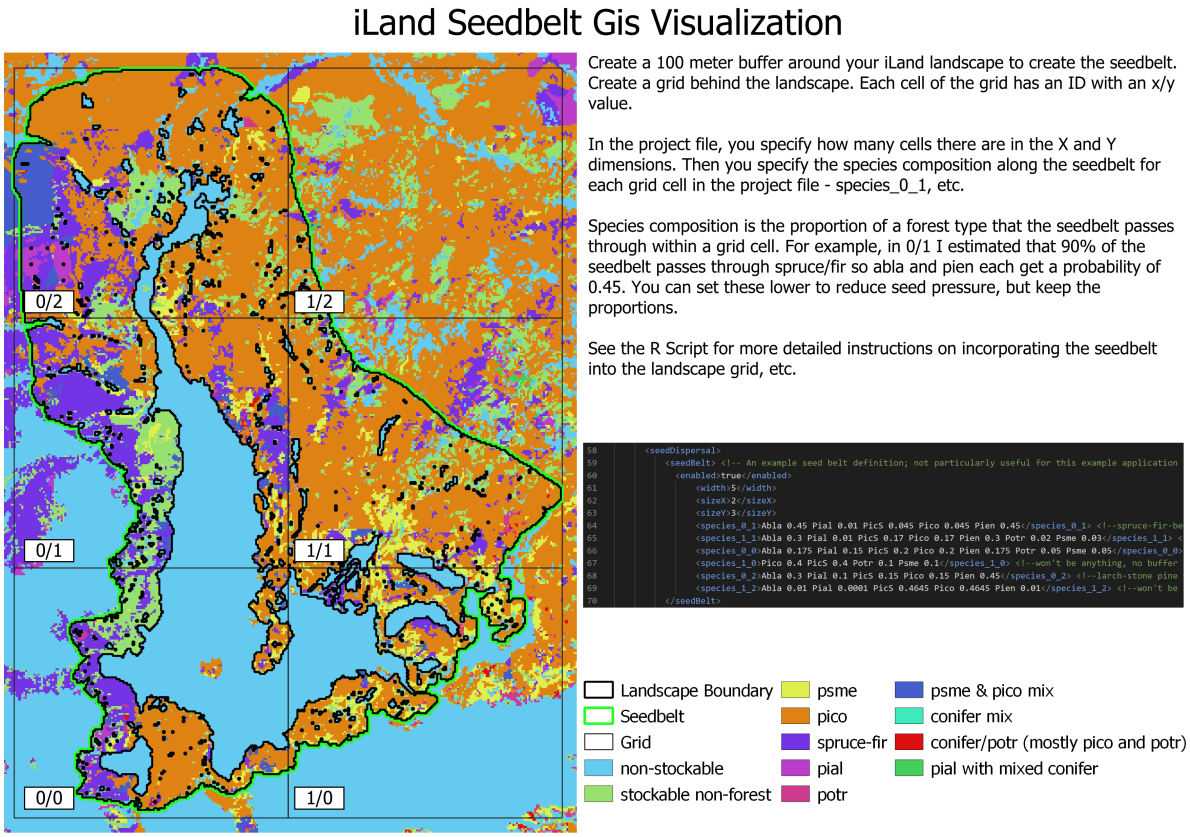

Figure 10.14: *An overview of the process of creating a seed belt and assigning species composition for the Grand Teton landscape, landscape from Hansen et al. (2020), seed belt figure by Timon Keller.*

Calibration recommendations

First, the `sapReferenceRatio` parameter must be iteratively calibrated (see the comment above on this topic).

The stand-scale setup is well suited for calibrating multiple species parameters. A suggested progression is to first simulate monospecific stands with initial seedlings or external seeds to calibrate species-specific sapling survival and growth. When running these simulations, ask:

- Are saplings surviving? If not, consider adjusting survival-related parameters (`sapMaxStress`,`SapStressThreshold`).
- Are seedlings establishing? If not, consider adjusting establishment-related parameters. Look at the establishment debug output to get a sense of what factors might be most limiting. Note that some parameters set thresholds that give a yes/no for establishment, whereas others control establishment probability. For more information, see the iLand wiki.
- Are forested conditions being maintained over time and generally covering the range of expected densities and structures? Use the run with no external seed input to focus on whether seed supply and forested conditions are being maintained over time. If subsequent generations after the initial cohorts are not maintaining the forest, consider parameters associated with seed supply (`maturityYears`,`seedYearInterval`,`nonSeedYearFraction`,`fecundity`). If stand densities and structures are low, consider adjusting `sapReinekesR`, which determines how many trees will be recruited as individuals when sapling cohorts pass the 4 m threshold. Do not spend too much time worrying about the sapling densities themselves, as these rely on this same multiplier. It is more important and ecologically meaningful for the model to get the tree densities right.
- Are saplings growing fast enough? If not, consider adjusting `sapHeightGrowthPotential`. However, this is usually parameterized with with empirical data, so it generally should align with expectations. Comparison data on sapling age versus height can help facilitate the calibration of this parameter.

Next, use stand-scale simulations starting from bare ground, with seeds available from multiple species via either the seed belt or external seed, to calibrate parameters that influence interspecific competition and relative dominance. Parameters to focus on calibrating include:

- Regeneration parameters, especially `sapReinekesR`, `fecundity`, and other seed supply parameters (`maturityYears`,`seedYearInterval`,`nonSeedYearFraction`,`fecundity`).
- Seedling establishment parameters (see establishment and species parameter wiki pages) if a given species is not establishing, or sapling growth/mortality parameters (`sapHeightGrowthPotential`, `sapMaxStress`,`SapStressThreshold`) if establishment is occurring but the species is not "winning" the race to become a tree.
- If saplings are being recruited but trees are not competitive, consider parameters that may affect relative tree growth along environmental gradient such as response to plant-available nitrogen (`respNitrogenClass`), minimum and optimum growth temperature (`respTempMin` and `respTempMax`), shade tolerance (`lightResponseClass`), and soil water response (`psiMin`). You could also consider some of the mortality parameters, such as `probIntrinsic` and `probStress`. Whenever making changes to any of these, rerun the productivity and mortality experiments from above to check that they still correspond well.

Evaluation strategy

Stand-scale regeneration simulations can be used to evaluate single-species behavior if comparison data is available, such as sapling growth over time or seedling establishment relative to distance from seed source (Hansen et al. 2018).

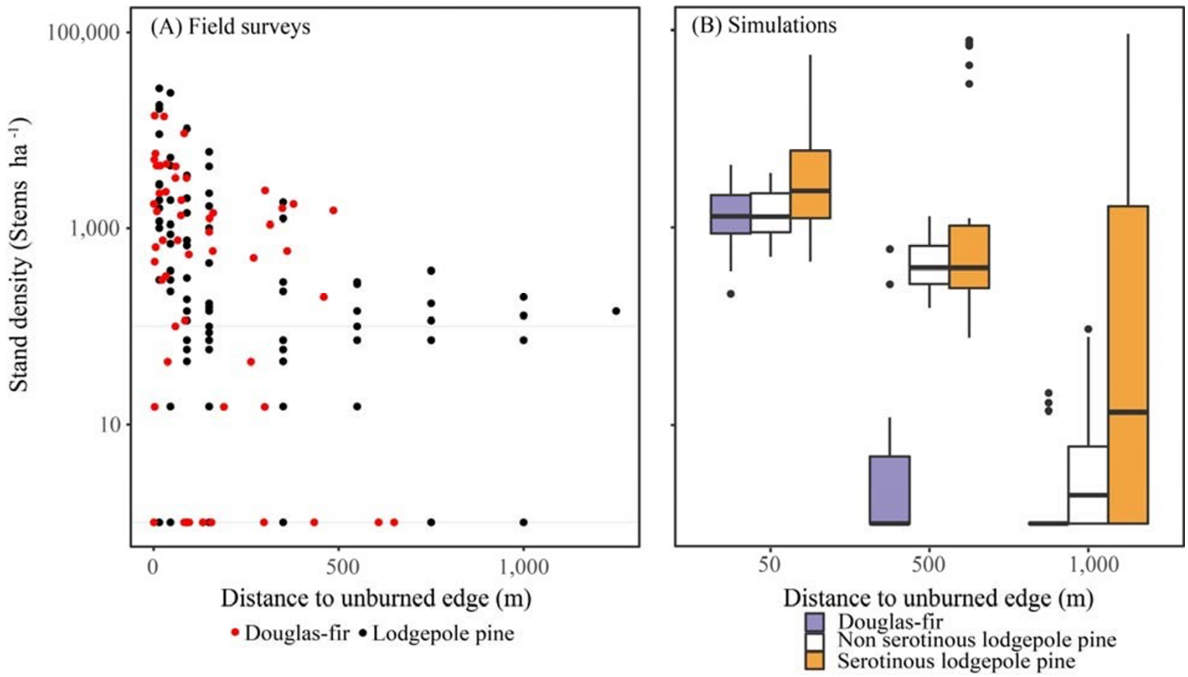

Figure 10.15: *Evaluation of stem densities at different distances from seed source, from Hansen et al. (2018), Supplementary material.*

Multispecies evaluations can include quantitative comparisons (e.g., goodness of fit), qualitative checks for general consistency with the distribution of field observations (e.g., comparison with stand density, basal area, and importance value over time or at specific stand ages; comparison of relative basal area shares by species over time with field observations), and/or expert assessment based on ecological expectations (e.g., evaluation of equilibrium or PNV simulations along a forest type or elevational gradient). At the end of a PNV simulation, a simulated forest type map can be created for comparison with a map of current forest types or with expectations (e.g., if management history has altered current forest types from their expected PNV conditions).

10.3 Other evaluation tests

Other possible evaluation tests include comparisons with other models (Braziunas et al. 2018); equilibrium simulations (Albrich, Rammer, and Seidl 2020); and tests of specific modules such as management (Albrich et al. 2018), disturbances (e.g., comparison against remote sensing products) (Hansen et al. 2020; Thom et al. 2022), or carbon and nitrogen cycling (Rupert Seidl, Spies, et al. 2012; Albrich et al. 2018; Hansen et al. 2020).

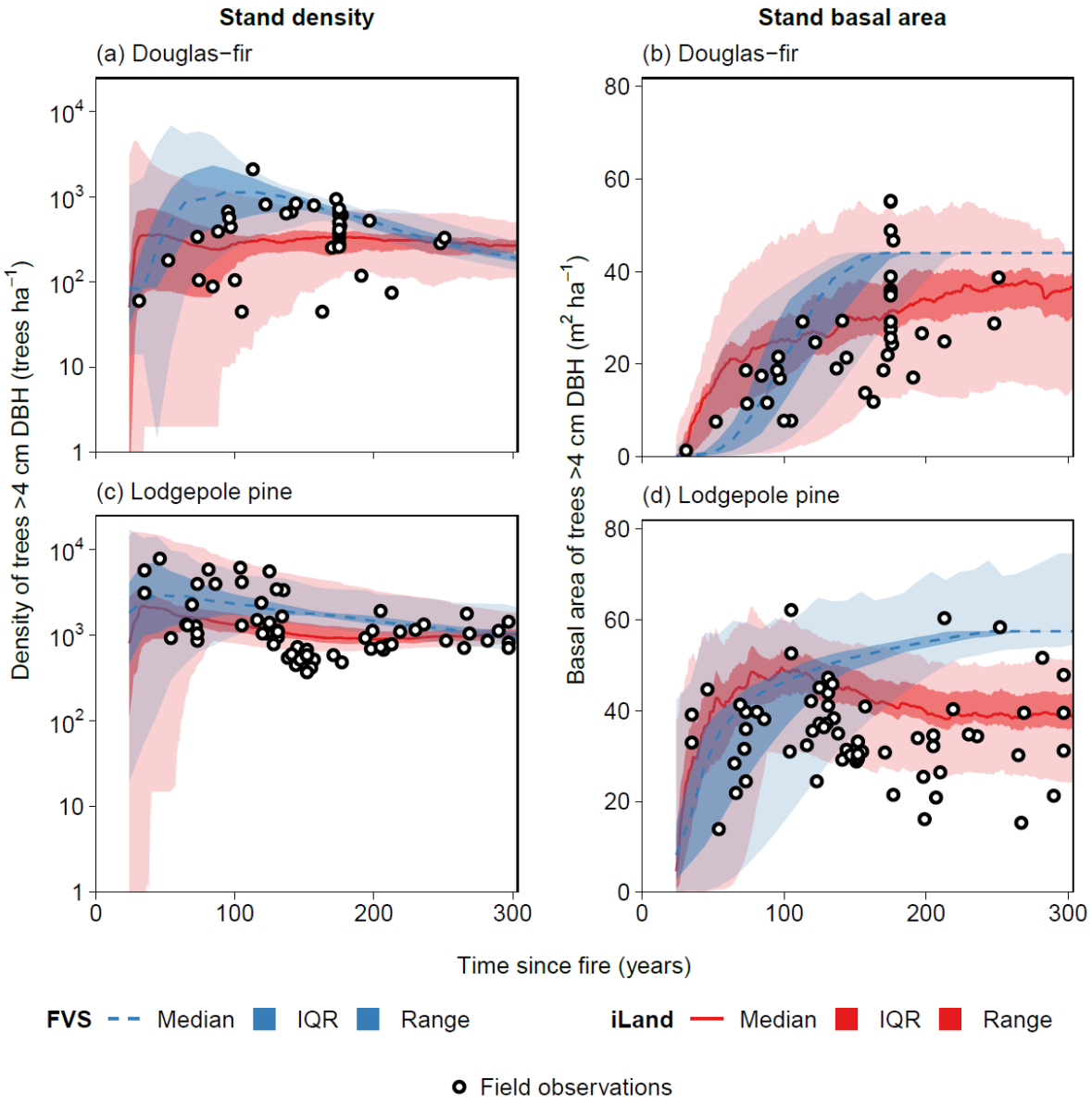

Figure 10.16: *Evaluation of forest trajectories simulated in iLand compared to the Forest Vegetation Simulator, an indiviual-tree growth and yield model, from Braziunas et al. (2018), Supplementary material.*

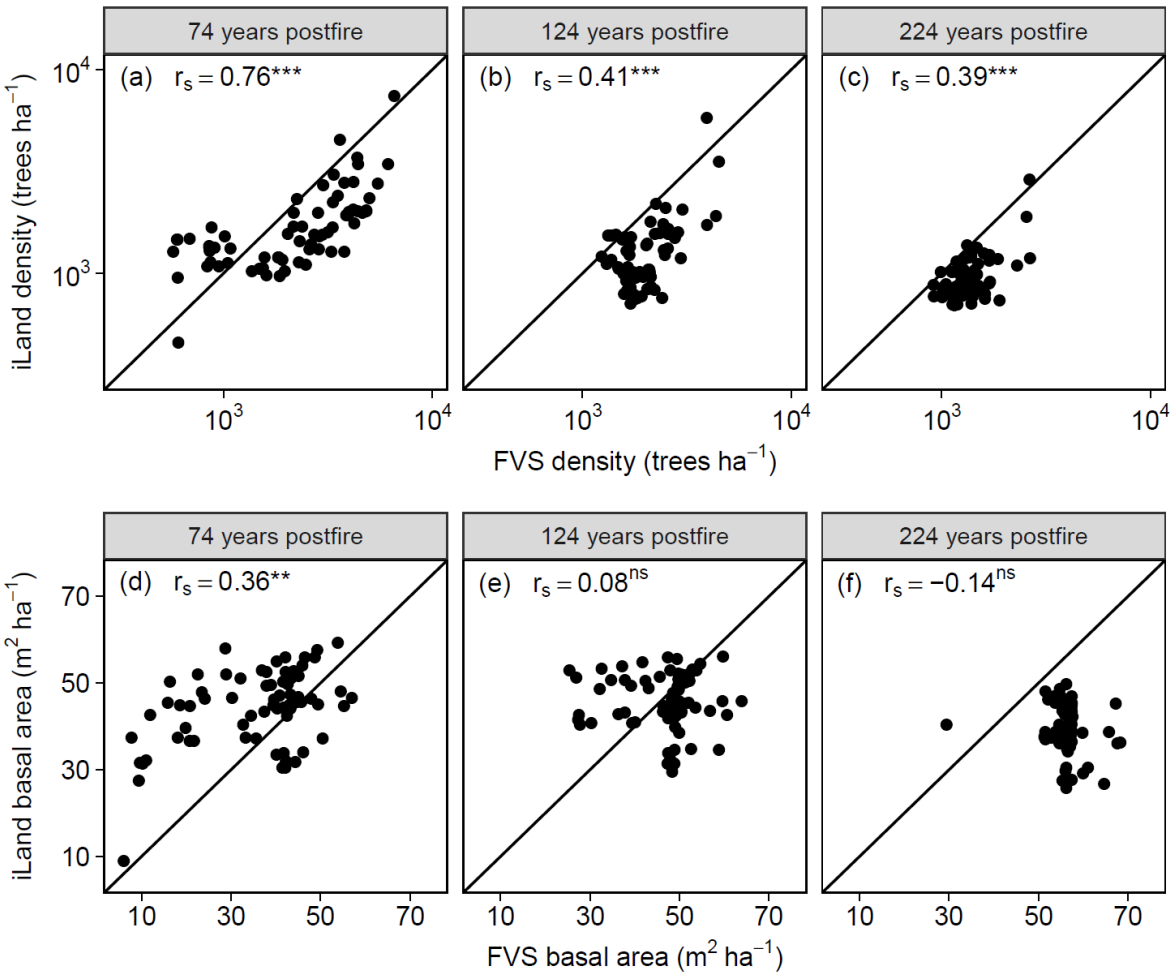

Figure 10.17: *Evaluation of post-fire lodgepole pine stand density and basal area simulated in iLand compared to the Forest Vegetation Simulator, an indiviual-tree growth and yield model, from Braziunas et al. (2018), Supplementary material.*

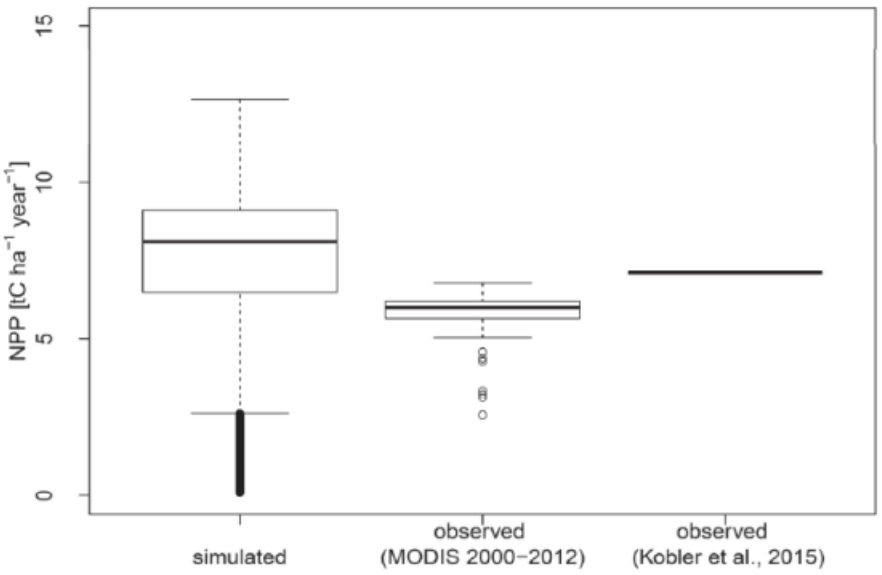

Figure 10.18: *Carbon cycle evaluation: Comparison of net primary productivity (NPP) simulated in iLand with remotely sensed data and stand-scale observations, from Albrich et al. (2018), Supplementary material.*

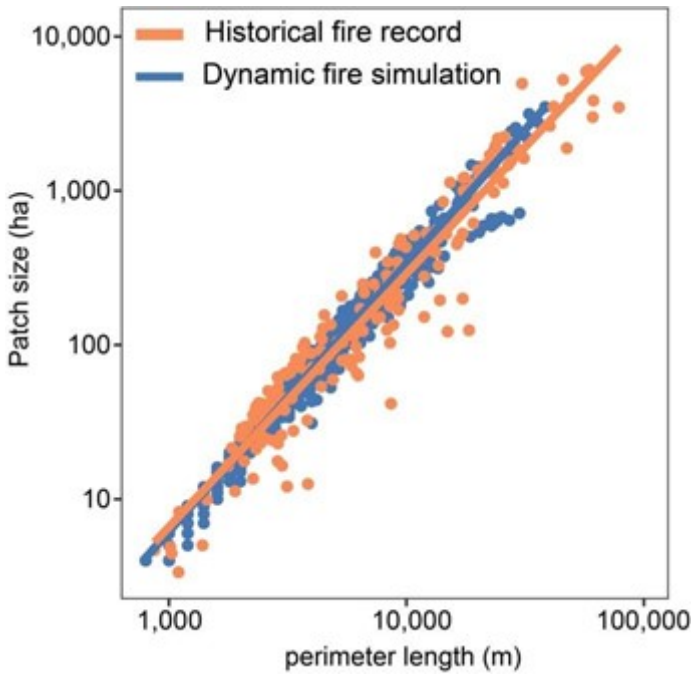

Figure 10.19: *Fire module evaluation: Comparison of fire size versus perimeter length simulated in iLand (blue) with historical record (orange), from Hansen et al. (2020), Supplementary material.*

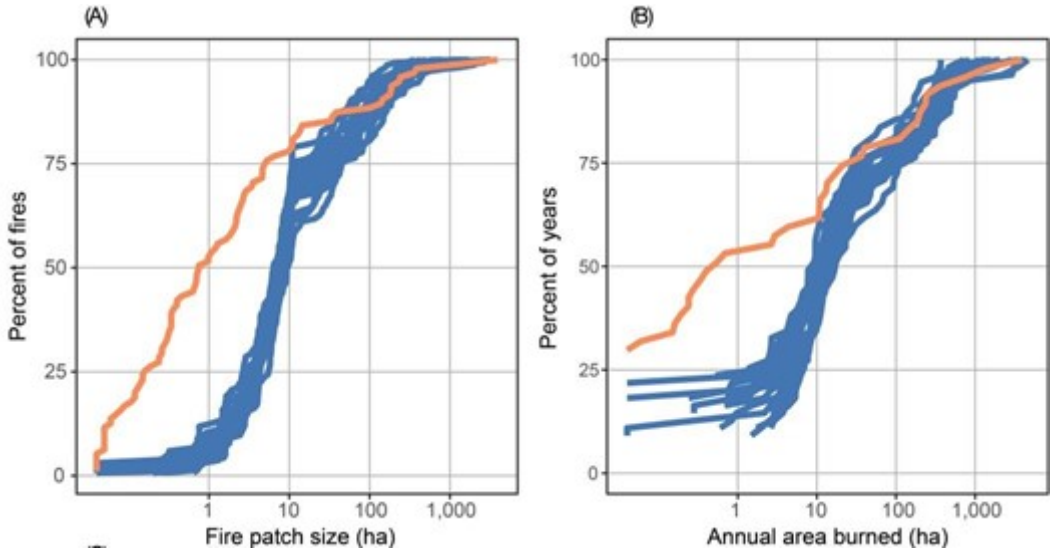

Figure 10.20: *Fire module evaluation: Comparison of cumulative frequency distribution of fire sizes simulated in iLand (blue) with historical record (orange), from Hansen et al. (2020), Supplementary material.*

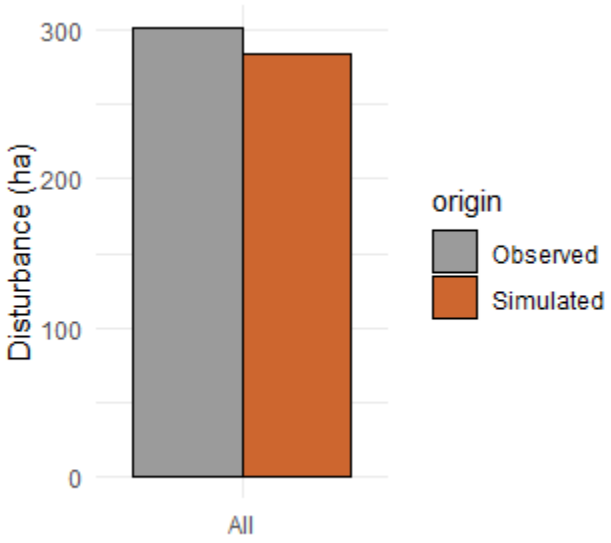

Figure 10.21: *Disturbance module evaluation: Comparison of cumulative area disturbed by wind or bark beetles over 18 years simulated in iLand with the historical record, from Thom et al. (2022), Supplementary material.*

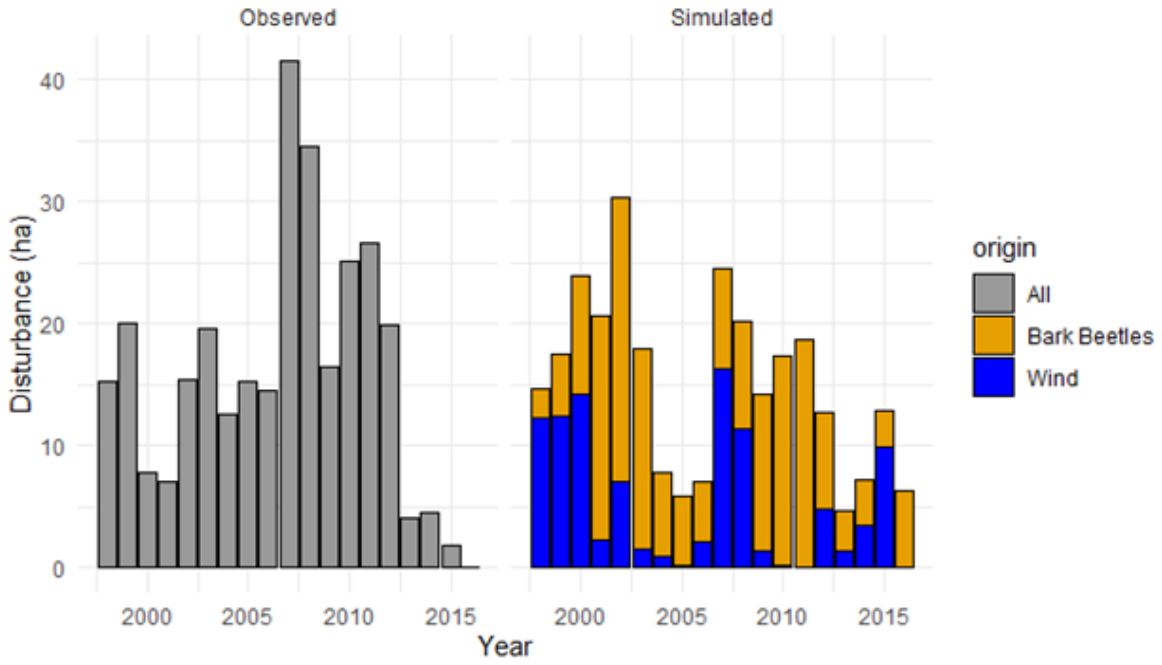

Figure 10.22: *Disturbance module evaluation: Comparison of annual area disturbed by wind or bark beetles simulated in iLand with the historical record, from Thom et al. (2022), Supplementary material.*

10.4 Concluding thoughts: Making parameters robust and adaptable through ongoing evaluation

Parameterized and calibrated species must strike a balance between accuracy within a specific region and generality to replicate patterns across diverse environments. As a goal, you want to be able to use the same species parameters across a wide range of environmental conditions, in new landscapes, and throughout a region or biome. You should re-evaluate whether species are behaving as expected every time you apply them in a new area or if you are adding new species to the pool. A useful approach is to create an automated pipeline allowing you to re-run the evaluations referenced above for your existing and new species pools, using the same set of stands and landscapes used in initial calibrations. This will allow you to identify effects of new changes on species behavior and interactions. By continuing to add evaluations for new regions and new landscapes, this should also make the species parameter sets more robust and reliable over time and across broader extents. It can also help identify when regional variants for species species, or entire species sets, are necessary. For example, the iLand team has developed the "Permanent Evaluation and Testing Suite" (PETS) framework for Central

European tree species, which enables standardized evaluation of changes in species parameters for all simulated forests with available data for model evaluation.

11 Managing simulation experiments

The iLand viewer (`iland.exe`) is an excellent tool for designing, initializing, evaluating, debugging, and visualizing your iLand landscape and simulation experiment. However, once you are ready to scale up from a single simulation run to a set of simulation scenarios (e.g., different climate, disturbance, or management scenarios) and multiple replications, it is much more efficient to automate this process. To do this, you can use the iLand console (`ilandc.exe`), which runs on both Windows and Linux platforms, and a shell script written in, e.g., Windows CMD or in bash. The critical first step of this process is making sure you are using the **same version of the iLand viewer and iLand console**. To check this, open `iland.exe` and go to `Help > About iLand` and note both the `SVN-Revision number` and `build date`. Then navigate to the iLand executable folder and enter `ilandc.exe` in a command line window and compare.

11.1 Command line parameters

The iLand console requires two inputs, the project file and the number of simulation years, and can take any number of additional options.

Listing 11.1 A generic example of running ilandc in the command-line. *years* and *additional-options* are placeholders that have to be replaced by actual values, see Listing 11.2.

```
# command-line input
ilandc.exe project-file.xml <years> <additional-options>
```

Additional inputs include any settings in the project file and should be specified following the hierarchical structure of the XML below the `<project>` level. For example, a different input climate database could be specified with `database.climate.out=new_climate.sqlite` or a different sequence of prescribed disturbance events could be specified with `model.world.timeEventsFile=timeevents_2.txt`.
Additional inputs can also include a javascript function run on model creation or simulation end.

For additional examples and more detail, see the iLand console wiki page.

Listing 11.2 A more specific example of running ilandc in bash (for the general syntax see Listing 11.1). The simulation setup is defined in `site_1.xml` and the simulation will run for 1000 years. Additional options are given that substitute corresponding values in `site_1.xml`.

```
#!/bin/bash
# "\" indicates line continuation
# make sure there is NO space after "\"
iland/ilandc.exe site_1.xml 1000 \
system.database.out=output_site_1.sqlite \
system.logging.logFile=log/log_site_1.txt \
model.site.availableNitrogen=50 \
model.site.soilDepth=100 \
model.site.pctSand=40 \
model.site.pctSilt=40 \
model.site.pctClay=20 \
model.climate.tableName=site_1
```

You can check the console output and/or the iLand log file to ensure that XML settings are being modified as expected.

11.2 Running multiple scenarios and replicates

The ability to modify any settings in the project file from the command line makes it straightforward to run many different simulation scenarios and multiple replicates in sequence or in parallel on a local machine or server. Scenarios can be specified in a file to run in sequence, as input arguments to a shell script to enable running in parallel, or as a combination of the two.

Here is an example of a `.csv` file with different climate and disturbance scenarios, each with 10 replicates.

```
gcm,rcp,wind_speed_mult,nreps
ICHEC-EC-EARTH,rcp45,0,10
ICHEC-EC-EARTH,rcp85,0,10
ICHEC-EC-EARTH,rcp45,15,10
ICHEC-EC-EARTH,rcp85,15,10
MPI-M-MPI-ESM-LR,rcp45,0,10
MPI-M-MPI-ESM-LR,rcp85,0,10
MPI-M-MPI-ESM-LR,rcp45,15,10
MPI-M-MPI-ESM-LR,rcp85,15,10
```

These can be used as inputs in a script to sequentially run iLand. Example bash script:

```
#!/bin/bash
# bash script to run iland for a list of scenarios
# usage: bash scripts/run_iland_local.sh

#####
# variables and arguments
#####

# set variables
# set this path to location of iland exe
path="/c/Users/user1/iland_executable/"
simulation_years="80"
csv_name="iland/main_file.csv" # csv filename
xml="cc_wind.xml"

#####
# loop through csv and run iland for each scenario
#####

# read in line by line, assign to variable
sed 1d $csv_name | while IFS=, read -r gcm rcp wind nreps
do
    for rep in $(seq 1 $nreps)
    do
        echo "running gcm $gcm with wind speed $wind rep $rep"

        # run iland with scenario settings
        ${path}ilandc.exe iland/${xml} $simulation_years \
        system.database.out=output_${gcm}_${wind}_${rep}.sqlite \
        system.database.climate=${gcm}.sqlite \
        model.world.timeEventsFile=timeevents_${rcp}_${wind}_${rep}.txt

    done
done
```

To run this script in bash, enter:

```
bash scripts/run_iland_local.sh
```

The script could alternatively include arguments, such as replicate number, that would make

it easy to run multiple simulations in parallel. For example, the below script allows running all simulation scenarios for only a specified range of replicates:

```
#!/bin/bash
# bash script to run iland for a list of scenarios
# usage: bash scripts/run_iland_local.sh [start_rep] [end_rep]

#####
# variables and arguments
#####

# arguments
start_rep=$1
end_rep=$2

# set variables
# set this path to location of iland exe
path="/c/Users/user1/iland_executable/"
simulation_years="80"
csv_name="iland/main_file.csv" # csv filename
xml="cc_wind.xml"

#####
# loop through csv and run iland for each scenario
#####

# read in line by line, assign to variable
sed 1d $csv_name | while IFS=, read -r gcm rcp wind nreps
do
    for rep in $(seq $start_rep $end_rep)
    do
        echo "running gcm $gcm with wind speed $wind rep $rep"

        # run iland with scenario settings
        ${path}ilandc.exe iland/${xml} $simulation_years \
        system.database.out=output_${gcm}_${wind}_${rep}.sqlite \
        system.database.climate=${gcm}.sqlite \
        model.world.timeEventsFile=timeevents_${rcp}_${wind}_${rep}.txt

    done
done
```

To run this script in bash for a single replicate (e.g., replicate 4), enter:

```
bash scripts/run_iland_local.sh 4 4
```

11.3 Pre-processing outputs

The same automation and scripting approach can be used to pre-process iLand outputs, for example using an `.R` script. This may be desirable if detailed iLand outputs (i.e., large file sizes) are needed, but then the same set of steps are applied to generate the dataset used for one's analysis. For example, Braziunas et al. (2021) applied a custom calculation to estimate fire intensity from iLand outputs. This required detailed information on stand structure, species composition, and sapling cohorts by resource unit, as well as fire spread rasters for each fire event. By adding a pre-processing step, they were able to automate the computationally intensive process of calculating fire intensity, delete the large iLand output files, and save a much smaller output file for each simulation replicate. Scripts associated with this project are on GitHub.

11.4 Running on a cluster

To further decrease the time required to complete simulations, iLand can be run on a compute cluster, which enables running many scenarios and replicates in parallel across multiple machines. This process requires compiling iLand on the compute cluster and preparing other files and scripts based on the cluster management software. If supported, pre-processing of outputs can also be included in compute cluster runs.

For additional information and examples, see the iLand wiki cluster page and this GitHub for an example with HTCondor software.

Listing 11.3 Example bash script with pre-processing iLand output

```
#!/bin/bash
# bash script to run iland for a list of scenarios
# usage: bash scripts/run_iland_local.sh [reps]

#####
# arguments and other variables
#####

# arguments
nreps=$1

# other variables
# set this path to location of iland exe
path="/c/Users/user1/iland_executable/"
simulation_years="120"
csv_name="iland/main_file.csv" # csv filename

#####
# loop through csv and run iland for each scenario
#####

# read in line by line, assign to variable
sed 1d $csv_name | while IFS=, read -r forest_type scen gcm kbdi
do
    for rep in $(seq 1 $nreps)
    do

        echo "running forest type $forest_type management scenario $scen \
        gcm $gcm"

        # management javascript file requires two inputs,
        # which change with each management scenario.
        # Here, I switch out these inputs for a given scenario
        and rep, each of
        # which has a unique clustered or dispersed management
        configuration generated
        # from a neutral landscape model.
        cp iland/gis/mgmt/${scen}_${rep}.txt iland/gis/mgmt/scenario_map.txt
        cp iland/gis/mgmt/${scen}_${rep}.csv iland/gis/mgmt/scenario_rids.csv

        # run each scenario
        ${path}ilandc.exe iland/${forest_type}_${scen}.xml $simulation_years \
        model.world.environmentFile=gis/${forest_type}_\
        ${gcm}_environment.txt \
        system.database.out=${forest_type}_${gcm}_${scen}_\
        output_${rep}.sqlite \
        modules.fire.KBDIref=$kbdi

        # reduce outputs with R code
        Rscript.exe iland/scripts/iland_output_prep.R
        Rscript.exe iland/scripts/fire_intensity.R
```

12 Developing iLand

12.1 Introduction

We believe that the results of research - especially when publicly funded - should also be available to the public. Furthermore, we believe that a fundamental principle of science is that research should be replicable by peers (which gets increasingly harder the more complex the applied tools become, and the more restricted traditional publication outlets become).

That's why we chose to release the full iLand software suite under the GNU GPL (General Public licence). The GNU General Public Licence (GPL) is one of the most commonly used licenses for open-source projects. The GPL grants and guarantees a wide range of rights to developers and users working with open-source software. Basically, it allows users to legally copy, distribute and modify software. This means you can:

Copy the software.
Copy it onto your own computers or servers, pretty much anywhere you want. There's no limit to the number of copies you can make.

Make whatever modifications to the software you want.
If you want to add or remove functionality, go ahead. If you want to use a portion of the code in another project, you can. The only catch is that the other project must also be released under the GPL.

One dedicated goal is to use as much open-source/free software as possible to not restrict the potential use of iLand by licenses etc. This includes:

- cross platform toolkit for (not only) the user interface (Qt)
- IDE (Qt creator), and compiler (Gnu C++)
- the SQLite database
- tools to organize source code (subversion, git) and documentation (doxygen, YUIdoc)

12.2 Getting iLand source code

The source code of iLand is available either as .ZIP package file bundled with the iLand download, or from our GitHub repository.

To get a copy of the iLand source code using git, type:

```
git clone https://github.com/edfm-tum/iland-model.git
```

This will get you the newest "stable" release of the model. To switch to the latest version currently under development, you can switch to the `dev` branch:

```
cd iland-model  # Navigate into the repository directory
git checkout dev # Switch to the "dev" branch
```

It is generally recommended to use the stable version (unless communicated otherwise).

12.3 Setting up Qt and basic requirements

To compile iLand on your target machine, or to develop iLand you need the Qt framework and other tools such as compilers.

Let us assume, that we use a Windows-based sytem and that the iLand source code is located in `c:\dev\iLand`. (see also below for other operating systems).

- **Get and install Qt**
  In order to be able to modify and compile iLand, you will need Qt and the QtCreator IDE. Both packages are freely available at http://qt.io/. The current release is compiled with Qt 6.5 for Windows (MSVC 64Bit). Note that it is also possible (but not tested) to work with another development system such as Microsoft Visual Studio.
- **Get a compiler**
  iLand is known to work with different compilers (Intel icc, msvc, gcc), but the most typical setup is either using Microsoft MSVC or the open source GCC compiler suite. The Microsoft compilers can be freely downloaded and used, and work very well on Windows (e.g., fast compile times), but are typically slightly complicated to install.
  https://doc.qt.io/qt-6/windows.html
  https://doc.qt.io/qt-6/linux.html

- **Setup in Qt Creator**
  After installing the Qt Creator IDE, start it and open the "iland.pro" project in the *src/iland* folder (be sure to preserve file paths when extracting the source code from a ZIP file). iLand uses no external libraries (except the Qt libraries) which should simplify the building process considerably – a click to "build" should suffice (in principle). iLand is split in three projects: `plugins` (disturbance modules), `iland` (the model with graphical user interface), and `ilandc` (the console version of the model). Both `iland` and `ilandc` depend on `plugins`.

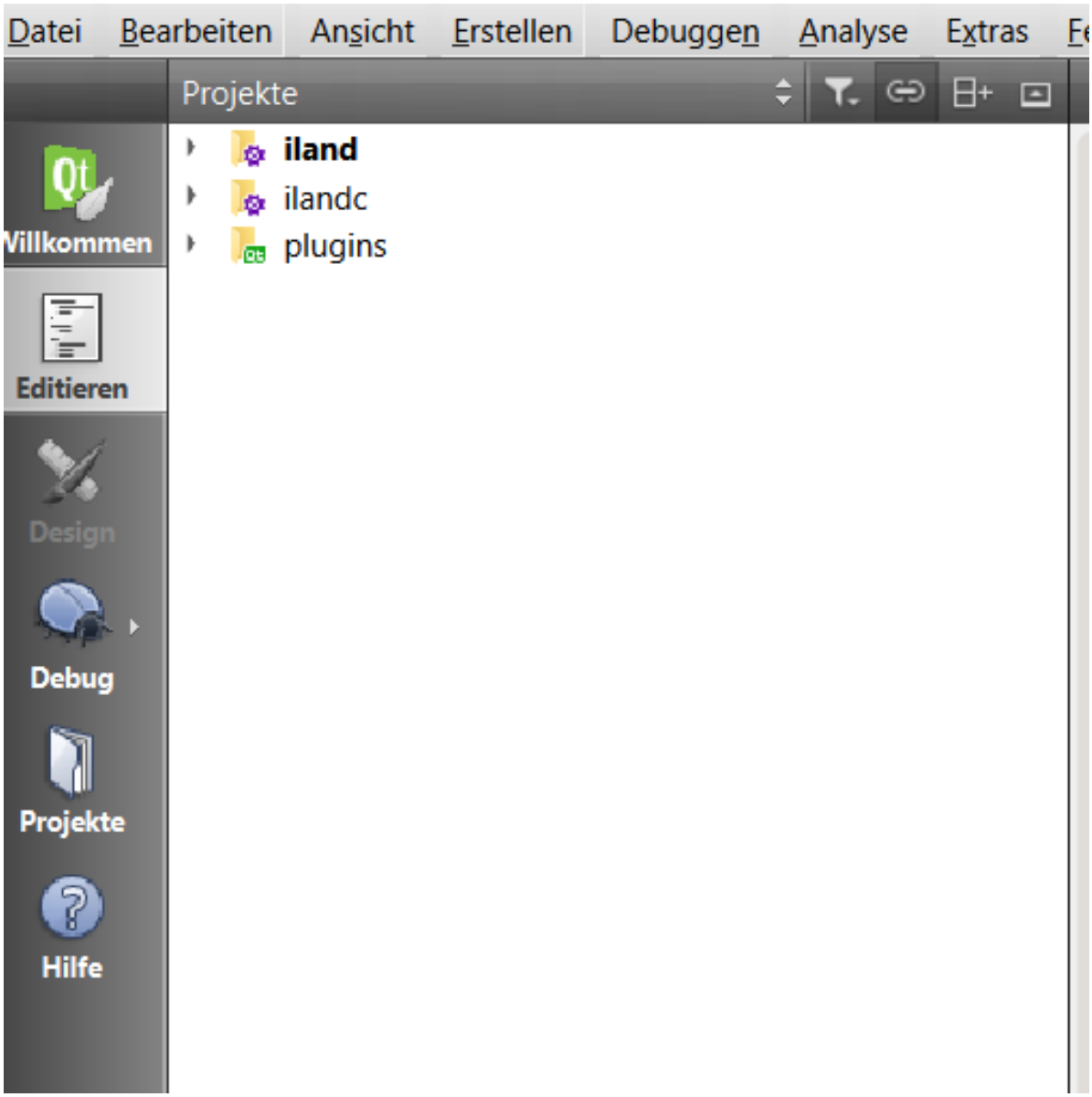

Figure 12.1: Load iland, ilandc and plugins in Qt Creator

- There are different ways how one can manage build locations. When you open a `.pro`-file for the first time, Creator asks for the "Kit" (or Kits) that you want to use. After selecting a Kit (e.g., Qt 6.5 MSVC) Creator creates build folders for each project (`iland`, `ilandc`, `plugins`) and configuration (debug / release builds). Using just the defaults should work - when you build the project `iland` (and when you built `plugins` earlier or set `plugins` as a dependency of `iland`), you should end up with a built executable (located in a folder such as

`C:\dev\iland-model\src\build-iland-Desktop_Qt_6_5_0_MSVC2019_64bit-Release\release`).

- Alternatively, you can create a common `build` directory, typically on the same level than the `src` folder (e.g., `c:\dev\iLand\build`). Now you need to change the build paths in Qt Creator. (Project settings, build path): use the common build-folder created earlier and build the subprojects (`plugins`, `iland`, `ilandc`) to the folders `\<build-folder\>/plugins`, `\<build-folder\>/iland`, `\<build-folder\>/ilandc` (e.g. `c:\dev\iland\build\iland`). The compiled executable file will then be saved to e.g. `c:/build/iland/release/iland.exe`. Make sure to also compile all sub projects with the same compiler version (see Figure 12.2). In this setting, the paths look a bit nicer, but your build build files are overwritten when switching between debug and release build.

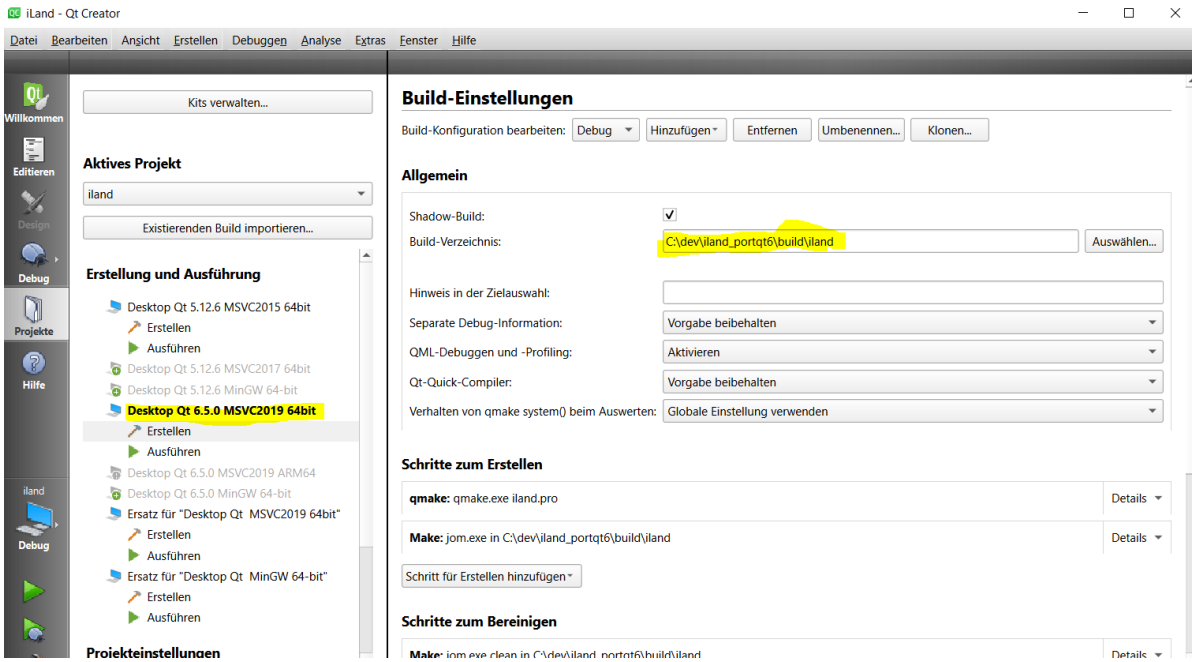

Figure 12.2: Use the "Project" tab to set build options. Change the path for every projects (iland, ilandc, plugins) (Note that the path differs from the text)

- Select a "Release" configuration when you just want to run the model, or "Debug" if you want to use breakpoints and step-by-step-debugging (Note that a debug build runs typically much slower!).
- Check the Qt documentation for more details on how to work with Qt and Qt creator!

12.4 Compiling iLand for different operating systems

12.4.1 Building iLand on systems with graphical user interface

iLand is supported on platforms that run Qt, which is Windows, Linux and Mac operating systems. For a Linux system with a graphical user interface (e.g., Ubuntu) the approach is very similar to using Windows (see above): install Qt and Qt Creator and build from within the IDE. Typically, you'd use `gcc` compilers on that platform. Mac systems *should* work, but we never actually tried it. If you have experience with compiling and running iLand on Mac, we'd be happy to hear about it!

12.4.2 Compiling iLand on Linux clusters

This is applicably mostly on large server infrastructure, for example high performance computing clusters. In such an environment you typically only want to build the console version of iLand.

While almost all HPC cluster installations use Linux, some subtle differences may apply. But see also this example for a Windows cluster environment!

In order to build iLand, you need:

- Qt installed on the cluster (Qt6) - check the cluster documentation how third-party packages are used!
- a suitable compiler - typically compilers are already available, check the cluster documentation which ones and how they are selected!
- The iLand code (from GitHub or a ZIP)

Example for a `build_iland.sh` bash script file to (re-) build iLand. In this case the "module" commands load specific software packages (here: intel compiler, qt). Note that there are likely differences depending on the cluster environment! This script builds the `ilandc` executable.

```
# build iLand on Linux cluster
# using Intel compiler (the -spec linux-icc-64 -> this might differ)
# Note: run in a build folder (on the same level as "src" of iLand)

module load intel
module load qt

mkdir plugins
mkdir ilandc
```

```
cd plugins
qmake ../src/plugins/plugins.pro -spec linux-icc-64
make

cd ../ilandc
qmake ../src/ilandc/ilandc.pro -spec linux-icc-64
make

cd ..
```

12.4.3 Building iLand on a standard Linux server

This is a build script to build iLand for Ubuntu. In this particular case, a local Qt installation is used (`/home/werner/Qt/6.5.3`). This script builds both the console and the UI version.

```
echo "Building iLand..."

QTDIR=/home/werner/Qt/6.5.3/gcc_64/bin
PATH=$QTDIR:$PATH

mkdir build
cd build
mkdir plugins
mkdir iland
mkdir ilandc

cd plugins
qmake ../../src/plugins/plugins.pro
make

cd ../iland
qmake ../../src/iland/iland.pro
make

cd ../ilandc
qmake ../../src/ilandc/ilandc.pro
make

cd ..
```

12.5 High level source code overview

The source code is organized into several sub directories:

- `core` - this includes the core logic of iLand. If you look for how a specific ecological process is implemented, than start here
- `abe` - code for the agent base forest management engine
- `bite` - code for the BITE module
- `output` - code that creates the different output tables of iLand.
- `tools` - Infrastructure and helping code for iLand (for example, classes to work with spatial grids)

Many objects/classes/elements of the model share a similar structure. Very often, there is a `setup()` function, and an `execute()` (or `run()` or `calculateXXX()`) function. The former is used to create model structures (and typically here values from the project file are used), and the latter when the model runs.

Many elements are organized hierarchically - this is particularly true for the classes representing spatial entities. For instance, the `Model` class represents the full forest landscape; it holds a grid (and a list) of `ResourceUnit`s. This class represents a 100x100m cell. The `ResourceUnit` contains a list of `Tree` objects (yes, you guessed correctly), and structures that represent information on species level (e.g., `ResourceUnitSpecies`) and links to other objects (such as `WaterCycle`). For example, the function `WaterCycle::setup()` initializes the water cycle for a resource unit, which is influenced by a number of settings (e.g, percentage of sand in the soil, `model.site.pctSand)` . On the other hand, the `WaterCycle::run()` function is called for each year and resource unit during a simulation and calculates soil-water related processes (such as evapotranspiration).

The entry point for the simulation of a single year in iLand is the `Model::runYear()` function (in `core/model.cpp`). All other steps are invoked from there directly or indirectly. However, the specific route to a specific line of code can be complicated. Here it is often useful to either use breakpoints or search for code references (see below).

There is a single class for each available output of the model. These are found in the `output` folder of the source code tree. For example, the `waterout.h` and `waterout.cpp` files include code and definition of the `WaterOut` class. Each of these output classes defines the columns (typically in the constructor, for example `WaterOut::WaterOut()`) and documentation for the output (the content of the https://iland-model.org/outputs is *generated* by iLand - this is what the "Output table description" action in the "Misc" menu is doing). The `exec()` function of each output creates the actual data, typically be extracting information from other model parts.

12.6 Tips for getting started

- Look at the names of the code files first - for example, an individual tree in iLand is represented by a "Tree" object which is defined in `tree.h` and `tree.cpp`.
- Use strings from error messages or the log file to quickly find the relevant parts of the code. Note that messages are often composed of multiple parts (e.g. file names) - be sure to look for the right parts of a string! This is also a quick way to find where a specific setting (in the project file) is used in the model.
- Use the tools provided by Qt Creator: in particular, Use the "find references" (`Ctrl+Shift+U`) - this allows you to quickly find from where a particular function is called (or a variable used)!
- Use the debugger and breakpoints: for this you need to compile iLand in "Debug" mode. Having done that, you can set breakpoints to stop the model at a specific point. From there, you can run the model step-by-step and view values of variables.

i Example - make something "editable"

Consider you want to make something that is currently "hard coded" editable from the project file (for example, you want to run some sort of sensitivity analysis, or you want to enable / disable a certain process).
The strategy would be to first locate the relevant part of the code (see above):

```
const double factor = 0.05; // hard coded factor
// ... some code that uses factor
```

Now, change to read a value from the project file:

```
    // find an appropriate spot in the project file - but you can also use
    // the "user" section:
    QString setting_key = "user.my_factor";
    double factor = GlobalSettings::instance()->settings().valueDouble(setting_key);
    // ... unchanged code that uses factor
```

Consider that reading from the settings is relatively slow; if this is a time-critical part of the model, it is better to read the setting once (for example, in the `setup()` function) and save to a member variable of the class!

Example - extend an output

Consider you want some extra information in one of the outputs that is currently not included.
The strategy would be to first think about *which* output to extend - is it the right spatial and temporal resolution? Next would be to locate the correct output class. Within each output class are two crucial spots:
Change the definition of the output:

```
// in the constructor of the output you'll find a statement like this:
columns() << OutputColumn::year() <<
....

// each column is created by an object that defines the column name, a column description a
<< OutputColumn("maxSnowCover", "Permafrost: maximum snow height (m) in a year.", OutDouble

// to extend, add a definition, for example after "maxSnowCover":
columns() << OutputColumn("myOutput", "this is …", OutDouble);

// that's it!
```

Change the `exec()` function to alter what information is written to the output table.

```
// look for statements as:
*this << <some data>

// in our case, we need to take care of the right sequence, so look for "maxSnowCover"
msd += wc->permafrost()->stats.maxSnowDepth;

// add your data and make sure it is in the right order and *before* writeRow()
*this << my_numeric_value;
...
writeRow(); // indicates the end of one row
```

This is also straightforward - the difficult past is often to have the information that you want to write somewhere available (it might be very well feasible to copy what iLand is doing anyway - for example, you could extend a data structure that already keeps stats on a process which you are interested in).

References

Albrich, Katharina, Werner Rammer, and Rupert Seidl. 2020. "Climate Change Causes Critical Transitions and Irreversible Alterations of Mountain Forests." *Global Change Biology* 26 (7): 4013–27. https://doi.org/10.1111/gcb.15118.

Albrich, Katharina, Werner Rammer, Dominik Thom, and Rupert Seidl. 2018. "Trade-Offs Between Temporal Stability and Level of Forest Ecosystem Services Provisioning Under Climate Change." *Ecological Applications* 28 (7): 1884–96. https://doi.org/https://doi.org/10.1002/eap.1785.

Albrich, Katharina, Rupert Seidl, Werner Rammer, and Dominik Thom. 2022. "From Sink to Source: Changing Climate and Disturbance Regimes Could Tip the 21st Century Carbon Balance of an Unmanaged Mountain Forest Landscape." *Forestry: An International Journal of Forest Research*.

Braziunas, Kristin H., Winslow D. Hansen, Rupert Seidl, Werner Rammer, and Monica G. Turner. 2018. "Looking Beyond the Mean: Drivers of Variability in Postfire Stand Development of Conifers in Greater Yellowstone." *Forest Ecology and Management* 430 (December): 460–71. https://doi.org/10.1016/j.foreco.2018.08.034.

Braziunas, Kristin H., Werner Rammer, Pieter De Frenne, Joan Díaz-Calafat, Per-Ola Hedwall, Cornelius Senf, Dominik Thom, Florian Zellweger, and Rupert Seidl. In prep. "Microclimate Temperature Effects Propagate Across Scales in Forest Ecosystems," In prep.

Braziunas, Kristin H., Rupert Seidl, Werner Rammer, and Monica G. Turner. 2021. "Can We Manage a Future with More Fire? Effectiveness of Defensible Space Treatment Depends on Housing Amount and Configuration." *Landscape Ecology* 36: 309–30. https://doi.org/10.1007/s10980-020-01162-x.

Burns, R. M., and B. H. Honkala,. 1990. *Silvics of North America*. Vol. 654. Agriculture Handbook. Washington, DC: US Department of Agriculture, Forest Service.

Dobor, L., T. Hlasny, W. Rammer, S. Zimova, I. Barka, and R. Seidl. 2020. "Spatial Configuration Matters When Removing Windfelled Trees to Manage Bark Beetle Disturbances in Central European Forest Landscapes." *Journal of Environmental Management* 254 (January). https://doi.org/10.1016/j.jenvman.2019.109792.

Grimm, Volker, Eloy Revilla, Uta Berger, Florian Jeltsch, Wolf M. Mooij, Steven F. Railsback, Hans-hermann Hermann Thulke, et al. 2005. "Pattern-Oriented Modeling of Agent-Based Complex Systems: Lessons from Ecology." *Science (New York, N.Y.)* 310 (5750): 98791. https://doi.org/10.1126/science.1116681.

Hansen, Winslow D., Diane Abendroth, Werner Rammer, Rupert Seidl, and Monica G. Turner. 2020. "Can Wildland Fire Management Alter 21st-Century Subalpine Fire and Forests

in Grand Teton National Park, Wyoming, USA?" *Ecological Applications* 30 (2): 1–15. https://doi.org/10.1002/eap.2030.

Hansen, Winslow D., Kristin H. Braziunas, Werner Rammer, Rupert Seidl, and Monica G. Turner. 2018. "It Takes a Few to Tango: Changing Climate and Fire Regimes Can Cause Regeneration Failure of Two Subalpine Conifers." *Ecology* 99 (4): 966–77. https://doi.org/https://doi.org/10.1002/ecy.2181.

Hansen, Winslow D., Adrianna Foster, Benjamin Gaglioti, Rupert Seidl, and Werner Rammer. 2023. "The Permafrost and Organic LayEr Module for Forest Models (POLE-FM) 1.0." *Geoscientific Model Development* 16 (7): 20112036. https://doi.org/10.5194/gmd-16-2011-2023.

Holzer, Dominik, Kai Bödeker, Werner Rammer, and Thomas Knoke. 2024. "Evaluating Dynamic Tree-Species-Shifting and Height Development Caused by Ungulate Browsing in Forest Regeneration Using a Process-Based Modeling Approach." *Ecological Modelling* 493 (July): 110741. https://doi.org/10.1016/j.ecolmodel.2024.110741.

Honkaniemi, Juha, Werner Rammer, and Rupert Seidl. 2021. "From Mycelia to Mastodons–a General Approach for Simulating Biotic Disturbances in Forest Ecosystems." *Environmental Modelling & Software* 138: 104977. https://doi.org/10.1016/j.envsoft.2021.104977.

Kätterer, Thomas, and Olof Andrén. 2001. "The ICBM Family of Analytically Solved Models of Soil Carbon, Nitrogen and Microbial Biomass Dynamics — Descriptions and Application Examples." *Ecological Modelling* 136 (2): 191–207. https://doi.org/10.1016/S0304-3800(00)00420-8.

Kattge, Jens, Gerhard Bönisch, Sandra Díaz, Sandra Lavorel, Iain Colin Prentice, Paul Leadley, Susanne Tautenhahn, et al. 2020. "TRY Plant Trait Database – Enhanced Coverage and Open Access." *Global Change Biology* 26 (1): 119–88. https://doi.org/10.1111/gcb.14904.

Kobayashi, Yuta, Rupert Seidl, Werner Rammer, Kureha F. Suzuki, and Akira S. Mori. 2023. "Identifying Effective Tree Planting Schemes to Restore Forest Carbon and Biodiversity in Shiretoko National Park, Japan." *Restoration Ecology* 31 (1): e13681. https://doi.org/10.1111/rec.13681.

Niinemets, Ü, F Valladares, and Centro De Ciencias Medioambientales. 2006. "Tolerance to Shade, Drought, and Waterlogging of Temperate Northern Hemisphere Trees and Shrubs." *Ecological Monographs* 76 (4): 521547. http://www.esajournals.org/doi/full/10.1890/0012-9615(2006)076[0521:TTSDAW]2.0.CO;2 http://www.esajournals.org/doi/full/10.1890/0012-9615(2006)076\%5B0521:TTSDAW\%5D2.0.CO;2.

Nitschke, Craig R., and John L. Innes. 2008. "A Tree and Climate Assessment Tool for Modelling Ecosystem Response to Climate Change." *Ecological Modelling* 210 (3): 263–77. https://doi.org/https://doi.org/10.1016/j.ecolmodel.2007.07.026.

Poggio, Laura, Luis M De Sousa, Niels H Batjes, Gerard Heuvelink, Bas Kempen, Eloi Ribeiro, and David Rossiter. 2021. "SoilGrids 2.0: Producing Soil Information for the Globe with Quantified Spatial Uncertainty." *Soil* 7 (1): 217–40. https://doi.org/10.5194/soil-7-217-2021.

Purves, Jeremy W. AND Pacala, Drew W. AND Lichstein. 2007. "Crown Plasticity and Competition for Canopy Space: A New Spatially Implicit Model Parameterized for 250

North American Tree Species." *PLOS ONE* 2 (9): 1–11. https://doi.org/10.1371/journal.pone.0000870.

Rammer, Werner, Dominik Thom, Martin Baumann, Kristin H. Braziunas, Christina Dollinger, Jonas Kerber, Johannes Mohr, and Rupert Seidl. 2024. "The Individual-Based Forest Landscape and Disturbance Model iLand: Progress and Outlook." *Ecological Modelling*.

Riley, Karin L, Isaac C Grenfell, and Mark A Finney. 2016. "Mapping Forest Vegetation for the Western United States Using Modified Random Forests Imputation of FIA Forest Plots." *Ecosphere* 7 (10): e01472.

Seidl, R., and W. Rammer. 2017. "Climate Change Amplifies the Interactions Between Wind and Bark Beetle Disturbances in Forest Landscapes." *Landscape Ecology* 32 (7): 1485–98. https://doi.org/10.1007/s10980-016-0396-4.

Seidl, Rupert, Werner Rammer, Robert M. Scheller, and Thomas A. Spies. 2012. "An Individual-Based Process Model to Simulate Landscape-Scale Forest Ecosystem Dynamics." *Ecological Modelling* 231 (April): 87100. https://doi.org/10.1016/j.ecolmodel.2012.02.015.

Seidl, Rupert, Thomas A. Spies, Werner Rammer, E. Ashley Steel, Robert J. Pabst, and Keith Olsen. 2012. "Multi-Scale Drivers of Spatial Variation in Old-Growth Forest Carbon Density Disentangled with Lidar and an Individual-Based Landscape Model." *Ecosystems* 15 (8): 1321–35. https://doi.org/10.1007/s10021-012-9587-2.

Thom, Dominik, Werner Rammer, Katharina Albrich, Kristin H. Braziunas, Laura Dobor, Christina Dollinger, Winslow D. Hansen, et al. 2024. "Parameters of 150 Temperate and Boreal Tree Species and Provenances for an Individual-Based Forest Landscape and Disturbance Model." *Data in Brief*, June, 110662. https://doi.org/10.1016/j.dib.2024.110662.

Thom, Dominik, Werner Rammer, Rita Garstenauer, and Rupert Seidl. 2018. "Legacies of Past Land Use Have a Stronger Effect on Forest Carbon Exchange Than Future Climate Change in a Temperate Forest Landscape." *Biogeosciences* 15 (18): 5699–5713. https://doi.org/10.5194/bg-15-5699-2018.

Thom, Dominik, Werner Rammer, Patrick Laux, Gerhard Smiatek, Harald Kunstmann, Sebastian Seibold, and Rupert Seidl. 2022. "Will Forest Dynamics Continue to Accelerate Throughout the 21st Century in the Northern Alps?" *Global Change Biology* 28 (10). https://doi.org/10.1111/gcb.16133.

Turner, Monica G., Kristin H. Braziunas, Winslow D. Hansen, Tyler J. Hoecker, Werner Rammer, Zak Ratajczak, A. Leroy Westerling, and Rupert Seidl. 2022. "The Magnitude, Direction, and Tempo of Forest Change in Greater Yellowstone in a Warmer World with More Fire." *Ecological Monographs* 92 (1): e01485. https://doi.org/10.1002/ecm.1485.